
FLINT Documentation

Release 3.5.0-dev

The FLINT development team

Mar 26, 2026

CONTENTS

1	Introduction	3
1.1	Introduction	3
1.1.1	What is FLINT?	3
1.1.2	Maintainers and Authors	3
1.1.3	Requirements	3
1.1.4	Structure of FLINT	4
1.1.5	License	4
1.2	Building, testing and installing	4
1.2.1	Quick start	4
1.2.2	Library and install paths	5
1.2.3	Testing FLINT	5
1.2.4	Static or dynamic library only	6
1.2.5	AVX2 instructions	6
1.2.6	TLS, reentrancy and single mode	6
1.2.7	ABI and architecture support	7
1.2.8	CMake build for Windows users	7
1.2.9	Uninstalling FLINT	7
1.2.10	Assertion checking	7
1.2.11	Linking and running code	8
1.3	Bug reporting	8
1.3.1	Reporting bugs	8
1.4	Contributing to FLINT	9
1.4.1	Code conventions	9
1.4.2	Test code	9
1.5	Contributors	10
1.5.1	Contributors	10
1.6	Examples	10
1.6.1	Example programs	10
1.7	Memory management	11
1.7.1	Memory allocation functions	11
1.7.2	Global caches and cleanup	11
1.7.3	Temporary allocation	11
1.8	Portability	12
1.8.1	Portable FLINT types	12
1.9	Threading	12
1.9.1	Multithreaded FLINT	12
1.9.2	Writing threaded functions in FLINT	13
1.9.3	Functional parallel programming helpers	13
2	General utilities	15
2.1	flint.h – global definitions	15
2.1.1	Macros	15
2.1.2	Integer types	16
2.1.3	Allocation Functions	17

2.1.4	Random Numbers	17
2.1.5	Thread functions	17
2.1.6	Input/Output	18
2.1.7	Exceptions	21
2.1.8	Sorting and searching	22
2.2	profiler.h – performance profiling	22
2.2.1	Timer based on clock	22
2.2.2	Timer based on cycle counter	23
2.2.3	Memory usage	25
2.3	thread_pool.h – thread pool	25
2.3.1	Thread pool	25
2.4	mpoly.h – support functions for multivariate polynomials	26
2.4.1	Orderings	26
2.4.2	Monomial arithmetic	27
2.4.3	Monomial comparison	27
2.4.4	Monomial divisibility	28
2.4.5	Basic manipulation	28
2.4.6	Setting and getting monomials	29
2.4.7	Packing and unpacking monomials	30
2.4.8	Chunking	31
2.4.9	Chained heap functions	31
2.5	machine_vectors.h – SIMD-accelerated operations on fixed-length vectors	31
2.5.1	Types	31
2.5.2	Printing	32
2.5.3	Access and conversions	32
2.5.4	Permutations	33
2.5.5	Comparisons	33
2.5.6	Arithmetic and basic operations	33
2.5.7	Modular arithmetic	35
3	Generic rings	37
3.1	gr.h – generic structures and their elements	37
3.1.1	Introduction	37
3.1.2	Context operations	40
3.1.3	Element operations	41
3.2	gr.h (continued) – implementing rings	50
3.2.1	Example	50
3.2.2	Method table	51
3.2.3	Placeholder and trivial methods	51
3.2.4	Required methods	51
3.2.5	Testing rings	52
3.3	gr.h (continued) – builtin domains and types	53
3.3.1	Coercions	53
3.3.2	Domain properties	53
3.3.3	Debugging	54
3.3.4	Groups	54
3.3.5	Basic rings and fields	54
3.3.6	Residue rings and finite fields	55
3.3.7	Number fields and algebraic numbers	55
3.3.8	Real and complex numbers	55
3.3.9	Extended number sets	56
3.3.10	Floating-point arithmetic	56
3.3.11	Vectors	57
3.3.12	Matrices	57
3.3.13	Polynomial rings	57
3.3.14	Power series	57
3.3.15	Fraction fields	58
3.3.16	Symbolic expressions	58

3.4	gr_generic.h – basic algorithms and fallback implementations for generic elements . . .	59
3.4.1	Generic string parsing	59
3.4.2	Generic arithmetic	60
3.4.3	Generic special functions	62
3.4.4	Generic vector methods	62
3.4.5	Generic polynomial methods	65
3.5	gr_special.h – special arithmetic and transcendental functions	66
3.5.1	Mathematical constants	66
3.5.2	Elementary functions	66
3.5.3	Factorials and gamma functions	67
3.5.4	Combinatorial numbers	68
3.5.5	Error function and exponential integrals	69
3.5.6	Orthogonal polynomials	69
3.5.7	Bessel, Airy and Coulomb functions	70
3.5.8	Hypergeometric functions	70
3.5.9	Riemann zeta, polylogarithms and Dirichlet L-functions	71
3.5.10	Elliptic integrals	71
3.5.11	Elliptic, modular and theta functions	72
3.6	gr_vec.h – vectors over generic rings	73
3.6.1	Types and basic operations	73
3.6.2	Arithmetic	74
3.6.3	Sums and products	76
3.6.4	Dot products	76
3.6.5	Sorting and searching	77
3.6.6	Other functions	77
3.7	gr_mat.h – dense matrices over generic rings	78
3.7.1	Type compatibility	78
3.7.2	Types, macros and constants	78
3.7.3	Memory management	79
3.7.4	Window matrices	79
3.7.5	Input and output	79
3.7.6	Comparisons	80
3.7.7	Assignment and special values	80
3.7.8	Basic row, column and entry operations	80
3.7.9	Entrywise operations	81
3.7.10	Norms	82
3.7.11	Addition and scalar arithmetic	82
3.7.12	Matrix multiplication	83
3.7.13	Powering	84
3.7.14	Polynomial evaluation	84
3.7.15	Diagonal and triangular matrices	84
3.7.16	Gaussian elimination	84
3.7.17	Solving	85
3.7.18	Determinant and trace	86
3.7.19	Permanent	87
3.7.20	Rank	87
3.7.21	Row echelon form	87
3.7.22	Nullspace	88
3.7.23	Inverse and adjugate	88
3.7.24	Characteristic polynomial	88
3.7.25	Minimal polynomial	89
3.7.26	Companion matrix	90
3.7.27	Similarity transformations	90
3.7.28	Eigenvalues	90
3.7.29	Jordan decomposition	91
3.7.30	Matrix functions	91
3.7.31	Hessenberg form	92
3.7.32	Random matrices	92

3.7.33	Orthogonal matrices	92
3.7.34	QR decomposition	93
3.7.35	Special matrices	94
3.7.36	Helper functions for reduction	94
3.7.37	LLL	95
3.7.38	Linear ODEs	95
3.7.39	Test functions	96
3.8	gr_poly.h – dense univariate polynomials over generic rings	97
3.8.1	Supported coefficient domains	97
3.8.2	Type compatibility	97
3.8.3	Weak normalization	97
3.8.4	Types, macros and constants	97
3.8.5	Memory management	98
3.8.6	Basic manipulation	98
3.8.7	Arithmetic	99
3.8.8	Multiplication algorithms	100
3.8.9	Scalar arithmetic	101
3.8.10	Powering	102
3.8.11	Shifting	102
3.8.12	Scalar division	103
3.8.13	Division with remainder	103
3.8.14	Division with remainder with full precomputed inverse	104
3.8.15	Power series division	105
3.8.16	Exact division	106
3.8.17	Square roots	107
3.8.18	Evaluation	107
3.8.19	Newton basis	108
3.8.20	Multipoint evaluation and interpolation	109
3.8.21	Composition	110
3.8.22	Power series composition and reversion	111
3.8.23	Derivative and integral	112
3.8.24	Monic polynomials	112
3.8.25	GCD	112
3.8.26	Resultant	113
3.8.27	Squarefree factorization	114
3.8.28	Shift equivalence	114
3.8.29	Roots	116
3.8.30	Power series special functions	116
3.8.31	Modular arithmetic and composition	118
3.8.32	Test functions	120
3.9	gr_mpoly.h – sparse multivariate polynomials over generic rings	122
3.9.1	Types, macros and constants	122
3.9.2	Context object methods	123
3.9.3	Memory management	123
3.9.4	Basic manipulation	123
3.9.5	Generators	124
3.9.6	Conversions	124
3.9.7	Comparisons	124
3.9.8	Random generation	125
3.9.9	Input and output	125
3.9.10	Coefficient and exponent access	125
3.9.11	Arithmetic	125
3.9.12	Derivative and integral	126
3.9.13	Other operations	126
3.9.14	Container operations	126
3.10	gr_series.h – formal power series over generic rings	128
3.10.1	Context constructors	128
3.10.2	Truncated power series	128

3.10.3	Power series	130
4	Integers	135
4.1	ulong_extras.h – arithmetic and number-theoretic functions for single-word integers . .	135
4.1.1	Simple example	135
4.1.2	Random functions	136
4.1.3	Basic arithmetic	137
4.1.4	Miscellaneous	137
4.1.5	Basic arithmetic with precomputed inverses	137
4.1.6	Greatest common divisor	140
4.1.7	Jacobi and Kronecker symbols	141
4.1.8	Modular arithmetic	141
4.1.9	Modular arithmetic with fixed operand	142
4.1.10	Divisibility testing	144
4.1.11	Prime number generation and counting	144
4.1.12	Primality testing	146
4.1.13	Chinese remaindering	148
4.1.14	Square root and perfect power testing	148
4.1.15	Factorisation	149
4.1.16	Arithmetic functions	153
4.1.17	Factorials	153
4.1.18	Primitive roots and discrete logarithms	153
4.1.19	Elliptic curve method for factorization of ulong	154
4.2	mpz.h – integers	155
4.2.1	Simple example	155
4.2.2	Types, macros and constants	156
4.2.3	Memory management	157
4.2.4	Random generation	157
4.2.5	Conversion	158
4.2.6	Input and output	161
4.2.7	Basic properties and manipulation	162
4.2.8	Comparison	163
4.2.9	Basic arithmetic	163
4.2.10	Greatest common divisor	167
4.2.11	Modular arithmetic	168
4.2.12	Bit packing and unpacking	169
4.2.13	Logic Operations	170
4.2.14	Chinese remaindering	170
4.2.15	Primality testing	172
4.2.16	Special functions	175
4.3	mpz_vec.h – vectors of integers	175
4.3.1	Memory management	175
4.3.2	Randomisation	175
4.3.3	Bit sizes and norms	175
4.3.4	Input and output	176
4.3.5	Conversions	177
4.3.6	Assignment and basic manipulation	177
4.3.7	Comparison	177
4.3.8	Sorting	178
4.3.9	Addition and subtraction	178
4.3.10	Scalar multiplication and division	178
4.3.11	Sums and products	179
4.3.12	Reduction mod p	179
4.3.13	Gaussian content	180
4.3.14	Dot product	180
4.4	mpz_factor.h – integer factorisation	180
4.4.1	Types, macros and constants	180
4.4.2	Factoring integers	180

4.4.3	Input and output	182
4.4.4	Elliptic curve (ECM) method	183
4.5	fmpz_mat.h – matrices over the integers	184
4.5.1	Simple example	184
4.5.2	Types, macros and constants	185
4.5.3	Memory management	185
4.5.4	Basic assignment and manipulation	185
4.5.5	Window	186
4.5.6	Random matrix generation	186
4.5.7	Input and output	188
4.5.8	Comparison	188
4.5.9	Transpose	189
4.5.10	Concatenate	189
4.5.11	Modular reduction and reconstruction	189
4.5.12	Addition and subtraction	190
4.5.13	Matrix-scalar arithmetic	190
4.5.14	Matrix multiplication	191
4.5.15	Inverse	192
4.5.16	Kronecker product	192
4.5.17	Content	192
4.5.18	Trace	193
4.5.19	Determinant	193
4.5.20	Permanent	194
4.5.21	Transforms	194
4.5.22	Characteristic polynomial	194
4.5.23	Minimal polynomial	194
4.5.24	Rank	195
4.5.25	Column partitioning	195
4.5.26	Nonsingular solving	195
4.5.27	Row reduction	197
4.5.28	Strong echelon form and Howell form	198
4.5.29	Nullspace	198
4.5.30	Echelon form	198
4.5.31	Hermite normal form	199
4.5.32	Smith normal form	200
4.5.33	Special matrices	200
4.5.34	Conversions	201
4.5.35	Cholesky Decomposition	201
4.5.36	LLL	201
4.5.37	Classical LLL	201
4.5.38	Modified LLL	202
4.6	fmpz_lll.h – LLL reduction	202
4.6.1	Parameter manipulation	202
4.6.2	Random parameter generation	202
4.6.3	Heuristic dot product	202
4.6.4	The various Babai’s	203
4.6.5	Shift	204
4.6.6	Varieties of LLL	204
4.6.7	ULLL	205
4.6.8	LLL-reducedness	206
4.6.9	Modified ULLL	206
4.6.10	Main LLL functions	206
4.7	fmpz_poly.h – univariate polynomials over the integers	207
4.7.1	Introduction	207
4.7.2	Simple example	207
4.7.3	Definition of the <code>fmpz_poly_t</code> type	207
4.7.4	Types, macros and constants	208
4.7.5	Memory management	208

4.7.6	Polynomial parameters	209
4.7.7	Assignment and basic manipulation	209
4.7.8	Randomisation	210
4.7.9	Getting and setting coefficients	211
4.7.10	Comparison	212
4.7.11	Addition and subtraction	212
4.7.12	Scalar absolute value, multiplication and division	213
4.7.13	Bit packing	214
4.7.14	Multiplication	214
4.7.15	FFT precached multiplication	217
4.7.16	Squaring	218
4.7.17	Powering	219
4.7.18	Shifting	220
4.7.19	Bit sizes and norms	221
4.7.20	Greatest common divisor	221
4.7.21	Discriminant	224
4.7.22	Gaussian content	224
4.7.23	Square-free	225
4.7.24	Euclidean division	225
4.7.25	Division with precomputed inverse	229
4.7.26	Divisibility testing	230
4.7.27	Division mod p	231
4.7.28	Power series division	231
4.7.29	Pseudo division	232
4.7.30	Derivative	234
4.7.31	Evaluation	234
4.7.32	Newton basis	236
4.7.33	Interpolation	236
4.7.34	Composition	237
4.7.35	Inflation and deflation	237
4.7.36	Taylor shift	238
4.7.37	Power series composition	238
4.7.38	Power series reversion	239
4.7.39	Square root	239
4.7.40	Power sums	241
4.7.41	Signature	241
4.7.42	Hensel lifting	241
4.7.43	Input and output	244
4.7.44	Modular reduction and reconstruction	245
4.7.45	Products	246
4.7.46	Subproduct trees	246
4.7.47	Roots	247
4.7.48	Minimal polynomials	247
4.7.49	Orthogonal polynomials	248
4.7.50	Fibonacci polynomials	249
4.7.51	Eulerian numbers and polynomials	249
4.7.52	Modular forms and q-series	250
4.7.53	CLD bounds	250
4.8	fmpz_poly_mat.h – matrices of polynomials over the integers	250
4.8.1	Simple example	250
4.8.2	Types, macros and constants	251
4.8.3	Memory management	251
4.8.4	Basic properties	251
4.8.5	Basic assignment and manipulation	252
4.8.6	Input and output	252
4.8.7	Random matrix generation	252
4.8.8	Special matrices	252
4.8.9	Basic comparison and properties	253

4.8.10	Norms	253
4.8.11	Transpose	253
4.8.12	Evaluation	253
4.8.13	Arithmetic	253
4.8.14	Row reduction	255
4.8.15	Trace	255
4.8.16	Determinant and rank	256
4.8.17	Inverse	256
4.8.18	Nullspace	256
4.8.19	Solving	256
4.9	fmpz_poly_factor.h – factorisation of polynomials over the integers	257
4.9.1	Types, macros and constants	257
4.9.2	Memory management	257
4.9.3	Manipulating factors	257
4.9.4	Input and output	258
4.9.5	Factoring algorithms	258
4.10	fmpz_mpoly.h – multivariate polynomials over the integers	259
4.10.1	Types, macros and constants	259
4.10.2	Context object	259
4.10.3	Memory management	259
4.10.4	Input/Output	260
4.10.5	Basic manipulation	260
4.10.6	Constants	261
4.10.7	Degrees	261
4.10.8	Coefficients	262
4.10.9	Comparison	263
4.10.10	Conversion	263
4.10.11	Container operations	263
4.10.12	Random generation	265
4.10.13	Addition/Subtraction	265
4.10.14	Scalar operations	266
4.10.15	Differentiation/Integration	267
4.10.16	Evaluation	267
4.10.17	Multiplication	268
4.10.18	Powering	268
4.10.19	Division	268
4.10.20	Greatest Common Divisor	269
4.10.21	Square Root	270
4.10.22	Univariate Functions	270
4.10.23	Internal Functions	271
4.10.24	Vectors	274
4.10.25	Ideals and Gröbner bases	275
4.10.26	Special polynomials	275
4.11	fmpz_mpoly_factor.h – factorisation of multivariate polynomials over the integers	276
4.11.1	Types, macros and constants	276
4.11.2	Memory management	276
4.11.3	Basic manipulation	276
4.11.4	Factorisation	277
4.12	long_extras.h – support functions for signed word arithmetic	277
4.12.1	Properties	277
4.12.2	Checked Arithmetic	277
4.12.3	Random functions	277
4.12.4	Modular arithmetic	278
4.13	longlong.h – support functions for multi-word arithmetic	278
4.13.1	Bit manipulation	278
4.13.2	Addition and subtraction	278
4.13.3	Multiplication	279
4.13.4	Division	279

4.14	mpn_extras.h – support functions for limb arrays	279
4.14.1	Macros	279
4.14.2	Utility functions	279
4.14.3	Addition and subtraction	280
4.14.4	Multiplication	280
4.14.5	Truncating multiplication	281
4.14.6	Divisibility	282
4.14.7	Division	283
4.14.8	Division and modular arithmetic with precomputed inverses	283
4.14.9	Preconditioned modular multiplication	285
4.14.10	GCD	286
4.14.11	Random Number Generation	286
4.15	aprc1.h – APRCL primality testing	286
4.15.1	Primality test functions	287
4.15.2	Configuration functions	287
4.15.3	Cyclotomic arithmetic	288
4.16	arith.h – arithmetic and special functions	291
4.16.1	Harmonic numbers	291
4.16.2	Stirling numbers	292
4.16.3	Bell numbers	293
4.16.4	Bernoulli numbers and polynomials	294
4.16.5	Euler numbers and polynomials	295
4.16.6	Multiplicative functions	295
4.16.7	Landau’s function	296
4.16.8	Number of partitions	296
4.16.9	Sums of squares	297
4.17	fft.h – Schoenhage-Strassen FFT	298
4.17.1	Split/combine FFT coefficients	298
4.17.2	Test helper functions	298
4.17.3	Arithmetic modulo a generalised Fermat number	298
4.17.4	Generic butterflies	299
4.17.5	Radix 2 transforms	299
4.17.6	Matrix Fourier Transforms	302
4.17.7	Negacyclic multiplication	304
4.17.8	Integer multiplication	305
4.17.9	Convolution	305
4.17.10	FFT Precaching	305
4.18	fft_small.h – FFT modulo word-size primes	306
4.18.1	Integer multiplication	306
4.18.2	Polynomial arithmetic	306
4.18.3	Preconditioned polynomial arithmetic	307
4.19	qsieve.h – Quadratic sieve	307
4.20	radix.h – multiprecision arithmetic in general radix	309
4.20.1	Radix objects	309
4.20.2	Low-level natural number arithmetic	310
4.20.3	Radix conversion	313
4.20.4	String conversion	313
4.20.5	Memory-managed integers	313
5	Rational numbers	317
5.1	fmpq.h – rational numbers	317
5.1.1	Types, macros and constants	317
5.1.2	Memory management	318
5.1.3	Canonicalisation	318
5.1.4	Basic assignment	318
5.1.5	Comparison	319
5.1.6	Conversion	319
5.1.7	Input and output	321

5.1.8	Random number generation	321
5.1.9	Arithmetic	322
5.1.10	Modular reduction and rational reconstruction	324
5.1.11	Rational enumeration	324
5.1.12	Continued fractions	325
5.1.13	Special functions	326
5.1.14	Dedekind sums	326
5.2	fmpq_vec.h – vectors over rational numbers	327
5.2.1	Memory management	327
5.2.2	Randomisation	327
5.2.3	Bit sizes and heights	327
5.2.4	Comparison	327
5.2.5	Sorting	327
5.2.6	Conversions	328
5.2.7	Dot product	328
5.2.8	Input and output	328
5.3	fmpq_mat.h – matrices over the rational numbers	328
5.3.1	Types, macros and constants	328
5.3.2	Memory management	329
5.3.3	Entry access	329
5.3.4	Basic assignment	329
5.3.5	Addition, scalar multiplication	330
5.3.6	Input and output	330
5.3.7	Random matrix generation	331
5.3.8	Window	331
5.3.9	Concatenate	331
5.3.10	Special matrices	331
5.3.11	Basic comparison and properties	331
5.3.12	Integer matrix conversion	332
5.3.13	Modular reduction and rational reconstruction	333
5.3.14	Matrix multiplication	333
5.3.15	Kronecker product	334
5.3.16	Trace	334
5.3.17	Determinant	334
5.3.18	Permanent	334
5.3.19	Nonsingular solving	334
5.3.20	Inverse	335
5.3.21	Echelon form	335
5.3.22	Gram-Schmidt Orthogonalisation	336
5.3.23	Transforms	336
5.3.24	Characteristic polynomial	336
5.3.25	Minimal polynomial	336
5.4	fmpq_poly.h – univariate polynomials over the rational numbers	336
5.4.1	Types, macros and constants	337
5.4.2	Memory management	337
5.4.3	Polynomial parameters	338
5.4.4	Accessing the numerator and denominator	338
5.4.5	Random testing	339
5.4.6	Assignment, swap, negation	339
5.4.7	Getting and setting coefficients	341
5.4.8	Comparison	341
5.4.9	Addition and subtraction	342
5.4.10	Scalar multiplication and division	343
5.4.11	Multiplication	345
5.4.12	Powering	345
5.4.13	Shifting	346
5.4.14	Euclidean division	346
5.4.15	Powering	347

5.4.16	Divisibility testing	347
5.4.17	Power series division	348
5.4.18	Greatest common divisor	348
5.4.19	Discriminant	349
5.4.20	Derivative and integral	350
5.4.21	Square roots	350
5.4.22	Power sums	351
5.4.23	Transcendental functions	351
5.4.24	Orthogonal polynomials	354
5.4.25	Evaluation	354
5.4.26	Interpolation	355
5.4.27	Composition	355
5.4.28	Power series composition	356
5.4.29	Power series reversion	357
5.4.30	Gaussian content	358
5.4.31	Square-free	359
5.4.32	Input and output	359
5.5	fmpz_mpoly_factor.h – factorisation of multivariate polynomials over the rational numbers	360
5.5.1	Types, macros and constants	360
5.5.2	Memory management	360
5.5.3	Basic manipulation	360
5.5.4	Factorisation	361
5.6	fmpz_mpoly.h – multivariate polynomials over the rational numbers	361
5.6.1	Types, macros and constants	361
5.6.2	Context object	361
5.6.3	Memory management	362
5.6.4	Input/Output	362
5.6.5	Basic manipulation	363
5.6.6	Constants	363
5.6.7	Degrees	364
5.6.8	Coefficients	364
5.6.9	Comparison	365
5.6.10	Container operations	365
5.6.11	Random generation	367
5.6.12	Addition/Subtraction	367
5.6.13	Scalar operations	368
5.6.14	Differentiation/Integration	369
5.6.15	Evaluation	369
5.6.16	Multiplication	369
5.6.17	Powering	370
5.6.18	Division	370
5.6.19	Greatest Common Divisor	370
5.6.20	Square Root	371
5.6.21	Univariate Functions	371
5.7	fmpz_poly_q.h – rational functions over the rational numbers	372
5.7.1	Simple example	373
5.7.2	Types, macros and constants	373
5.7.3	Memory management	373
5.7.4	Randomisation	374
5.7.5	Assignment	374
5.7.6	Comparison	374
5.7.7	Addition and subtraction	375
5.7.8	Scalar multiplication and division	375
5.7.9	Multiplication and division	375
5.7.10	Powering	375
5.7.11	Derivative	376
5.7.12	Evaluation	376

5.7.13	Input and output	376
5.8	mpz_mpoly_q.h – multivariate rational functions over $\mathbb{Q}$	376
5.8.1	Types and macros	377
5.8.2	Memory management	377
5.8.3	Assignment	377
5.8.4	Canonicalisation	377
5.8.5	Properties	378
5.8.6	Special values	378
5.8.7	Input and output	378
5.8.8	Random generation	379
5.8.9	Comparisons	379
5.8.10	Arithmetic	379
5.8.11	Content	380
6	Integers mod n	381
6.1	nmmod.h – integers mod n (word-size n)	381
6.1.1	Generic rings	381
6.1.2	Modular reduction and arithmetic	382
6.1.3	Montgomery arithmetic	383
6.1.4	Discrete Logarithms via Pohlig-Hellman	385
6.2	nmmod_vec.h – vectors over integers mod n (word-size n)	385
6.2.1	Memory management	385
6.2.2	Random functions	386
6.2.3	Basic manipulation and comparison	386
6.2.4	Printing	386
6.2.5	Arithmetic operations	387
6.2.6	Dot products	387
6.3	nmmod_mat.h – matrices over integers mod n (word-size n)	389
6.3.1	Types, macros and constants	389
6.3.2	Memory management	389
6.3.3	Basic properties and manipulation	390
6.3.4	Window	390
6.3.5	Concatenate	390
6.3.6	Printing	391
6.3.7	Random matrix generation	391
6.3.8	Comparison	392
6.3.9	Transposition and permutations	392
6.3.10	Addition and subtraction	392
6.3.11	Matrix-scalar arithmetic	392
6.3.12	Matrix multiplication	393
6.3.13	Matrix Exponentiation	394
6.3.14	Trace	394
6.3.15	Determinant and rank	394
6.3.16	Inverse	394
6.3.17	Triangular solving	394
6.3.18	Nonsingular square solving	395
6.3.19	LU decomposition	396
6.3.20	Reduced row echelon form	396
6.3.21	Nullspace	397
6.3.22	Transforms	397
6.3.23	Characteristic polynomial	397
6.3.24	Minimal polynomial	397
6.3.25	Strong echelon form and Howell form	397
6.4	nmmod_poly.h – univariate polynomials over integers mod n (word-size n)	398
6.4.1	Simple example	398
6.4.2	Types, macros and constants	399
6.4.3	Memory management	399
6.4.4	Polynomial properties	399

6.4.5	Assignment and basic manipulation	400
6.4.6	Randomization	400
6.4.7	Construction of irreducible polynomials	400
6.4.8	Getting and setting coefficients	401
6.4.9	Input and output	401
6.4.10	Comparison	402
6.4.11	Shifting	403
6.4.12	Addition and subtraction	403
6.4.13	Scalar multiplication and division	404
6.4.14	Bit packing and unpacking	404
6.4.15	KS2/KS4 Reduction	405
6.4.16	Multiplication	405
6.4.17	Preconditioned modular multiplication	408
6.4.18	Powering	409
6.4.19	Division	411
6.4.20	Divisibility testing	414
6.4.21	Derivative and integral	414
6.4.22	Evaluation	414
6.4.23	Multipoint evaluation	415
6.4.24	Interpolation	417
6.4.25	Composition	418
6.4.26	Taylor shift	419
6.4.27	Modular composition	419
6.4.28	Greatest common divisor	422
6.4.29	Discriminant	426
6.4.30	Power series composition	426
6.4.31	Power series reversion	426
6.4.32	Square roots	426
6.4.33	Power sums	427
6.4.34	Transcendental functions	428
6.4.35	Special polynomials	430
6.4.36	Products	430
6.4.37	Subproduct trees	430
6.4.38	Geometric progression	431
6.4.39	Inflation and deflation	431
6.4.40	Chinese Remaindering	431
6.4.41	Berlekamp-Massey Algorithm	432
6.5	nmod_poly_mat.h – matrices of univariate polynomials over integers mod n (word-size n)	433
6.5.1	Types, macros and constants	434
6.5.2	Memory management	434
6.5.3	Truncate, shift	434
6.5.4	Basic properties	434
6.5.5	Basic assignment and manipulation	434
6.5.6	Window	435
6.5.7	Concatenate	435
6.5.8	Input and output	435
6.5.9	Random matrix generation	435
6.5.10	Special matrices	436
6.5.11	Basic comparison and properties	436
6.5.12	Transposition	436
6.5.13	Norms	437
6.5.14	Evaluation	437
6.5.15	Arithmetic	437
6.5.16	Row reduction	438
6.5.17	Trace	439
6.5.18	Determinant and rank	439
6.5.19	Inverse	439

6.5.20	Nullspace	439
6.5.21	Solving	440
6.6	nmod_poly_factor.h – factorisation of univariate polynomials over integers mod n (word-size n)	440
6.6.1	Types, macros and constants	440
6.6.2	Factorisation	440
6.7	nmod_mpoly.h – multivariate polynomials over integers mod n (word-size n)	443
6.7.1	Types, macros and constants	443
6.7.2	Context object	443
6.7.3	Memory management	443
6.7.4	Input/Output	444
6.7.5	Basic manipulation	444
6.7.6	Constants	445
6.7.7	Degrees	445
6.7.8	Coefficients	446
6.7.9	Comparison	446
6.7.10	Container operations	446
6.7.11	Random generation	448
6.7.12	Addition/Subtraction	448
6.7.13	Scalar operations	448
6.7.14	Differentiation	449
6.7.15	Evaluation	449
6.7.16	Multiplication	450
6.7.17	Powering	450
6.7.18	Division	450
6.7.19	Greatest Common Divisor	451
6.7.20	Square Root	452
6.7.21	Univariate Functions	452
6.7.22	Internal Functions	453
6.8	nmod_mpoly_factor.h – factorisation of multivariate polynomials over integers mod n (word-size n)	453
6.8.1	Types, macros and constants	453
6.8.2	Memory management	454
6.8.3	Basic manipulation	454
6.8.4	Factorisation	454
6.9	mpn_mod.h – integers mod n (packed multi-word n)	455
6.9.1	Types, macros and constants	455
6.9.2	Context objects	455
6.9.3	Basic operations and arithmetic	456
6.9.4	Vector functions	457
6.9.5	Matrix algorithms	457
6.9.6	Polynomial algorithms	458
6.10	fmpz_mod.h – arithmetic modulo integers	460
6.10.1	Types, macros and constants	460
6.10.2	Context object	460
6.10.3	Conversions	460
6.10.4	Arithmetic	460
6.10.5	Discrete Logarithms via Pohlig-Hellman	461
6.11	fmpz_mod_vec.h – vectors over integers mod n	462
6.11.1	Conversions	462
6.11.2	Arithmetic	462
6.11.3	Scalar Multiplication	462
6.11.4	Dot Product	462
6.11.5	Multiplication	462
6.12	fmpz_mod_mat.h – matrices over integers mod n	463
6.12.1	Types, macros and constants	463
6.12.2	Element access	463
6.12.3	Memory management	463

6.12.4	Random generation	464
6.12.5	Windows and concatenation	464
6.12.6	Input and output	465
6.12.7	Comparison	465
6.12.8	Set and transpose	465
6.12.9	Conversions	465
6.12.10	Addition and subtraction	465
6.12.11	Scalar arithmetic	466
6.12.12	Matrix multiplication	466
6.12.13	Trace	467
6.12.14	Gaussian elimination	467
6.12.15	Strong echelon form and Howell form	467
6.12.16	Inverse	467
6.12.17	LU decomposition	468
6.12.18	Triangular solving	468
6.12.19	Solving	468
6.12.20	Transforms	469
6.12.21	Characteristic polynomial	469
6.12.22	Minimal polynomial	469
6.13	fmpz_mod_poly.h – polynomials over integers mod n	469
6.13.1	Simple example	470
6.13.2	Types, macros and constants	470
6.13.3	Memory management	470
6.13.4	Randomisation	471
6.13.5	Attributes	472
6.13.6	Assignment and basic manipulation	472
6.13.7	Conversion	473
6.13.8	Comparison	473
6.13.9	Getting and setting coefficients	474
6.13.10	Shifting	474
6.13.11	Addition and subtraction	474
6.13.12	Scalar multiplication and division	475
6.13.13	Multiplication	476
6.13.14	Products	477
6.13.15	Division	480
6.13.16	Divisibility testing	483
6.13.17	Power series inversion	483
6.13.18	Power series division	483
6.13.19	Greatest common divisor	484
6.13.20	Minpoly	487
6.13.21	Resultant	488
6.13.22	Discriminant	488
6.13.23	Derivative	488
6.13.24	Evaluation	489
6.13.25	Multipoint evaluation	489
6.13.26	Composition	490
6.13.27	Square roots	490
6.13.28	Modular composition	491
6.13.29	Subproduct trees	494
6.13.30	Radix conversion	494
6.13.31	Input and output	495
6.13.32	Inflation and deflation	496
6.13.33	Berlekamp-Massey Algorithm	496
6.14	fmpz_mod_poly_factor.h – factorisation of polynomials over integers mod n	497
6.14.1	Types, macros and constants	497
6.14.2	Factorisation	498
6.14.3	Root Finding	500
6.15	fmpz_mod_mpoly.h – polynomials over the integers mod n	500

6.15.1	Types, macros and constants	500
6.15.2	Context object	501
6.15.3	Memory management	501
6.15.4	Input/Output	501
6.15.5	Basic manipulation	502
6.15.6	Constants	502
6.15.7	Degrees	503
6.15.8	Coefficients	503
6.15.9	Comparison	504
6.15.10	Container operations	504
6.15.11	Random generation	506
6.15.12	Addition/Subtraction	507
6.15.13	Scalar operations	507
6.15.14	Differentiation	508
6.15.15	Evaluation	508
6.15.16	Multiplication	509
6.15.17	Powering	509
6.15.18	Division	509
6.15.19	Greatest Common Divisor	510
6.15.20	Square Root	510
6.15.21	Univariate Functions	511
6.15.22	Vectors	512
6.15.23	Ideals and Gröbner bases	513
6.15.24	Converting to/from other polynomial types	514
6.15.25	Internal Functions	514
6.16	fmpz_mod_mpoly_factor.h – factorisation of multivariate polynomials over the in- tegers mod n	514
6.16.1	Types, macros and constants	514
6.16.2	Memory management	514
6.16.3	Basic manipulation	515
6.16.4	Factorisation	515
6.17	fmpz_mod_mpoly_q.h – multivariate rational functions over $\mathbb{Z}/m\mathbb{Z}$	515
6.17.1	Types and macros	516
6.17.2	Memory management	516
6.17.3	Assignment	516
6.17.4	Canonicalisation	516
6.17.5	Properties	517
6.17.6	Special values	517
6.17.7	Input and output	517
6.17.8	Random generation	518
6.17.9	Comparisons	518
6.17.10	Arithmetic	518
7	Groups and other structures	521
7.1	perm.h – permutations	521
7.1.1	Memory management	521
7.1.2	Assignment	521
7.1.3	Composition	521
7.1.4	Parity	521
7.1.5	Randomisation	522
7.2	qfb.h – binary quadratic forms	522
7.2.1	Introduction	522
7.2.2	Memory management	522
7.2.3	Hash table	522
7.2.4	Basic manipulation	523
7.2.5	Comparison	523
7.2.6	Input/output	523
7.2.7	Computing with forms	523

7.3	dirichlet.h – Dirichlet characters	525
7.3.1	Dirichlet characters	525
7.3.2	Multiplicative group modulo q	525
7.3.3	Character type	526
7.3.4	Character properties	527
7.3.5	Character evaluation	528
7.3.6	Character operations	528
7.4	dlog.h – discrete logarithms mod along primes	529
7.4.1	Types, macros and constants	529
7.4.2	Single evaluation	529
7.4.3	Precomputations	529
7.4.4	Vector evaluations	530
7.4.5	Internal discrete logarithm strategies	530
7.5	bool_mat.h – matrices over booleans	533
7.5.1	Types, macros and constants	533
7.5.2	Memory management	533
7.5.3	Conversions	533
7.5.4	Input and output	534
7.5.5	Value comparisons	534
7.5.6	Random generation	534
7.5.7	Special matrices	534
7.5.8	Transpose	535
7.5.9	Arithmetic	535
7.5.10	Special functions	535
8	Number fields and algebraic numbers	537
8.1	nf.h – number fields	537
8.2	nf_elem.h – number field elements	537
8.2.1	Initialisation	537
8.2.2	Conversion	538
8.2.3	Basic manipulation	539
8.2.4	Comparison	539
8.2.5	I/O	539
8.2.6	Arithmetic	539
8.2.7	Representation matrix	541
8.2.8	Modular reduction	541
8.3	fmpz.h – Gaussian integers	542
8.3.1	Types, macros and constants	542
8.3.2	Basic manipulation	542
8.3.3	Input and output	542
8.3.4	Random number generation	542
8.3.5	Properties	542
8.3.6	Units	543
8.3.7	Norms	543
8.3.8	Arithmetic	543
8.3.9	Division	543
8.3.10	GCD	543
8.3.11	Primality testing	544
8.4	qqbar.h – algebraic numbers represented by minimal polynomials	544
8.4.1	Types and macros	544
8.4.2	Memory management	545
8.4.3	Assignment	545
8.4.4	Properties	545
8.4.5	Conversions	546
8.4.6	Special values	546
8.4.7	Input and output	547
8.4.8	Random generation	547
8.4.9	Comparisons	548

8.4.10	Complex parts	548
8.4.11	Integer parts	549
8.4.12	Arithmetic	549
8.4.13	Powers and roots	550
8.4.14	Numerical enclosures	551
8.4.15	Numerator and denominator	551
8.4.16	Conjugates	551
8.4.17	Polynomial evaluation	551
8.4.18	Polynomial roots	552
8.4.19	Roots of unity and trigonometric functions	553
8.4.20	Guessing and simplification	554
8.4.21	Symbolic expressions and conversion to radicals	554
8.4.22	Internal functions	557
9	Real and complex numbers	559
9.1	Feature overview	559
9.2	Using ball arithmetic	560
9.2.1	Ball semantics	560
9.2.2	Binary and decimal	560
9.2.3	Quality of enclosures	561
9.2.4	Predicates	562
9.2.5	A worked example: the sine function	563
9.2.6	More on precision and accuracy	565
9.2.7	Polynomial time guarantee	566
9.3	Technical conventions and potential issues	567
9.3.1	Integer overflow	567
9.3.2	Aliasing	567
9.3.3	Thread safety and caches	568
9.3.4	Use of hardware floating-point arithmetic	568
9.3.5	Interface changes	569
9.3.6	General note on correctness	569
9.4	Arb example programs	569
9.4.1	pi.c	569
9.4.2	pi_agm.c	570
9.4.3	zeta_zeros.c	570
9.4.4	bernoulli.c	571
9.4.5	class_poly.c	571
9.4.6	hilbert_matrix.c	571
9.4.7	keiper_li.c	572
9.4.8	logistic.c	573
9.4.9	real_roots.c	574
9.4.10	poly_roots.c	576
9.4.11	zeta_zeros.c	578
9.4.12	complex_plot.c	579
9.4.13	lvalue.c	580
9.4.14	lcentral.c	581
9.4.15	integrals.c	582
9.4.16	fpwrap.c	583
9.4.17	functions_benchmark.c	583
9.5	mag.h – fixed-precision unsigned floating-point numbers for bounds	583
9.5.1	Types, macros and constants	584
9.5.2	Memory management	584
9.5.3	Special values	585
9.5.4	Assignment and conversions	585
9.5.5	Comparisons	586
9.5.6	Input and output	586
9.5.7	Random generation	587
9.5.8	Arithmetic	587

9.5.9	Fast, unsafe arithmetic	588
9.5.10	Powers and logarithms	588
9.5.11	Special functions	589
9.6	nfloat.h – packed floating-point numbers with n-word precision	590
9.6.1	Types, macros and constants	591
9.6.2	Context objects	592
9.6.3	Basic operations and arithmetic	593
9.6.4	Vector functions	595
9.6.5	Matrix functions	595
9.6.6	Internal functions	596
9.6.7	Complex numbers	596
9.6.8	Packed fixed-point arithmetic	598
9.6.9	Directed rounding (experimental)	600
9.7	arf.h – arbitrary-precision floating-point numbers	601
9.7.1	Types, macros and constants	601
9.7.2	Memory management	602
9.7.3	Special values	602
9.7.4	Assignment, rounding and conversions	603
9.7.5	Comparisons and bounds	605
9.7.6	Magnitude functions	606
9.7.7	Shallow assignment	606
9.7.8	Random number generation	607
9.7.9	Input and output	607
9.7.10	Addition and multiplication	608
9.7.11	Summation	609
9.7.12	Dot products	609
9.7.13	Division	609
9.7.14	Square roots	610
9.7.15	Complex arithmetic	610
9.7.16	Low-level methods	610
9.8	acf.h – complex floating-point numbers	611
9.8.1	Types, macros and constants	611
9.8.2	Memory management	611
9.8.3	Basic manipulation	612
9.8.4	Arithmetic	612
9.8.5	Approximate arithmetic	612
9.9	arb.h – real numbers	612
9.9.1	Types, macros and constants	613
9.9.2	Memory management	614
9.9.3	Assignment and rounding	614
9.9.4	Assignment of special values	615
9.9.5	Input and output	616
9.9.6	Random number generation	617
9.9.7	Radius and interval operations	617
9.9.8	Comparisons	620
9.9.9	Arithmetic	622
9.9.10	Dot product	624
9.9.11	Powers and roots	625
9.9.12	Exponentials and logarithms	626
9.9.13	Trigonometric functions	627
9.9.14	Inverse trigonometric functions	628
9.9.15	Hyperbolic functions	628
9.9.16	Inverse hyperbolic functions	629
9.9.17	Constants	629
9.9.18	Lambert W function	630
9.9.19	Gamma function and factorials	630
9.9.20	Zeta function	631
9.9.21	Bernoulli numbers and polynomials	632

9.9.22	Polylogarithms	632
9.9.23	Other special functions	633
9.9.24	Internals for computing elementary functions	634
9.9.25	Vector functions	637
9.10	acb.h – complex numbers	638
9.10.1	Types, macros and constants	638
9.10.2	Memory management	639
9.10.3	Basic manipulation	639
9.10.4	Input and output	640
9.10.5	Random number generation	640
9.10.6	Precision and comparisons	641
9.10.7	Complex parts	643
9.10.8	Arithmetic	643
9.10.9	Dot product	645
9.10.10	Mathematical constants	646
9.10.11	Powers and roots	646
9.10.12	Exponentials and logarithms	647
9.10.13	Trigonometric functions	647
9.10.14	Inverse trigonometric functions	648
9.10.15	Hyperbolic functions	648
9.10.16	Inverse hyperbolic functions	648
9.10.17	Lambert W function	649
9.10.18	Rising factorials	649
9.10.19	Gamma function	650
9.10.20	Zeta function	651
9.10.21	Polylogarithms	651
9.10.22	Arithmetic-geometric mean	651
9.10.23	Other special functions	652
9.10.24	Piecewise real functions	652
9.10.25	Vector functions	653
9.11	arb_poly.h – polynomials over the real numbers	655
9.11.1	Types, macros and constants	655
9.11.2	Memory management	655
9.11.3	Basic manipulation	655
9.11.4	Conversions	656
9.11.5	Input and output	657
9.11.6	Random generation	657
9.11.7	Comparisons	657
9.11.8	Bounds	657
9.11.9	Arithmetic	657
9.11.10	Composition	660
9.11.11	Evaluation	660
9.11.12	Product trees	662
9.11.13	Multipoint evaluation	662
9.11.14	Interpolation	663
9.11.15	Differentiation	663
9.11.16	Transforms	664
9.11.17	Powers and elementary functions	665
9.11.18	Lambert W function	668
9.11.19	Gamma function and factorials	668
9.11.20	Zeta function	669
9.11.21	Root-finding	670
9.11.22	Other special polynomials	670
9.12	acb_poly.h – polynomials over the complex numbers	671
9.12.1	Types, macros and constants	671
9.12.2	Memory management	671
9.12.3	Basic properties and manipulation	671
9.12.4	Input and output	673

9.12.5	Random generation	673
9.12.6	Comparisons	673
9.12.7	Conversions	673
9.12.8	Bounds	674
9.12.9	Arithmetic	674
9.12.10	Composition	676
9.12.11	Evaluation	677
9.12.12	Product trees	677
9.12.13	Multipoint evaluation	678
9.12.14	Interpolation	678
9.12.15	Differentiation	679
9.12.16	Transforms	679
9.12.17	Elementary functions	680
9.12.18	Lambert W function	683
9.12.19	Gamma function	683
9.12.20	Power sums	684
9.12.21	Zeta function	684
9.12.22	Other special functions	685
9.12.23	Root-finding	686
9.13	arb_fmpz_poly.h – extra methods for integer polynomials	688
9.13.1	Evaluation	688
9.13.2	Utility methods	689
9.13.3	Polynomial roots	689
9.13.4	Special polynomials	689
9.14	acb_dft.h – Discrete Fourier transform	690
9.14.1	Main DFT functions	690
9.14.2	DFT on products	691
9.14.3	Convolution	691
9.14.4	FFT algorithms	692
9.15	arb_mat.h – matrices over the real numbers	694
9.15.1	Types, macros and constants	694
9.15.2	Memory management	694
9.15.3	Conversions	695
9.15.4	Random generation	695
9.15.5	Input and output	695
9.15.6	Comparisons	695
9.15.7	Special matrices	696
9.15.8	Transpose	697
9.15.9	Norms	697
9.15.10	Arithmetic	697
9.15.11	Scalar arithmetic	698
9.15.12	Vector arithmetic	699
9.15.13	Gaussian elimination and solving	699
9.15.14	Cholesky decomposition and solving	701
9.15.15	Characteristic polynomial and companion matrix	702
9.15.16	Special functions	703
9.15.17	Sparsity structure	703
9.15.18	Component and error operations	703
9.15.19	Eigenvalues and eigenvectors	704
9.15.20	LLL reduction	704
9.16	acb_mat.h – matrices over the complex numbers	704
9.16.1	Types, macros and constants	704
9.16.2	Memory management	705
9.16.3	Conversions	705
9.16.4	Random generation	706
9.16.5	Input and output	706
9.16.6	Comparisons	706
9.16.7	Special matrices	707

9.16.8	Transpose	707
9.16.9	Norms	708
9.16.10	Arithmetic	708
9.16.11	Scalar arithmetic	709
9.16.12	Vector arithmetic	709
9.16.13	Gaussian elimination and solving	710
9.16.14	Characteristic polynomial and companion matrix	712
9.16.15	Special functions	712
9.16.16	Component and error operations	713
9.16.17	Eigenvalues and eigenvectors	713
9.17	acb_hypgeom.h – hypergeometric functions of complex variables	715
9.17.1	Rising factorials	716
9.17.2	Gamma function	716
9.17.3	Convergent series	717
9.17.4	Asymptotic series	719
9.17.5	Generalized hypergeometric function	719
9.17.6	Confluent hypergeometric functions	719
9.17.7	Error functions and Fresnel integrals	720
9.17.8	Bessel functions	721
9.17.9	Modified Bessel functions	722
9.17.10	Airy functions	723
9.17.11	Coulomb wave functions	724
9.17.12	Incomplete gamma and beta functions	725
9.17.13	Exponential and trigonometric integrals	727
9.17.14	Gauss hypergeometric function	728
9.17.15	Orthogonal polynomials and functions	730
9.17.16	Dilogarithm	732
9.18	arb_hypgeom.h – hypergeometric functions of real variables	732
9.18.1	Rising factorials	733
9.18.2	Gamma function	733
9.18.3	Binomial coefficients	734
9.18.4	Generalized hypergeometric function	734
9.18.5	Confluent hypergeometric functions	734
9.18.6	Gauss hypergeometric function	735
9.18.7	Error functions and Fresnel integrals	735
9.18.8	Incomplete gamma and beta functions	736
9.18.9	Exponential and trigonometric integrals	738
9.18.10	Bessel functions	739
9.18.11	Airy functions	739
9.18.12	Coulomb wave functions	740
9.18.13	Orthogonal polynomials and functions	740
9.18.14	Dilogarithm	741
9.18.15	Hypergeometric sums	741
9.19	acb_elliptic.h – elliptic integrals and functions of complex variables	742
9.19.1	Complete elliptic integrals	742
9.19.2	Legendre incomplete elliptic integrals	743
9.19.3	Carlson symmetric elliptic integrals	744
9.19.4	Weierstrass elliptic functions	745
9.20	acb_modular.h – modular forms of complex variables	746
9.20.1	The modular group	747
9.20.2	Modular transformations	748
9.20.3	Addition sequences	748
9.20.4	Jacobi theta functions	749
9.20.5	Dedekind eta function	752
9.20.6	Modular forms	752
9.20.7	Elliptic integrals and functions	753
9.20.8	Class polynomials	753
9.21	acb_theta.h – Riemann theta functions	753

9.21.1	Main user functions	754
9.21.2	Example of usage	755
9.21.3	The Siegel modular group	756
9.21.4	The Siegel half space	758
9.21.5	Theta characteristics	759
9.21.6	Toolbox for derivatives	760
9.21.7	Ellipsoids	761
9.21.8	Error bounds in summation algorithms	762
9.21.9	Context structures in summation algorithms	763
9.21.10	Summation algorithms	765
9.21.11	AGM steps	767
9.21.12	Quasilinear algorithms on reduced input	768
9.21.13	Reduction and main functions	773
9.21.14	Dimension 2 specifics	774
9.21.15	Tests	777
9.21.16	Profiling	781
9.22	acb_dirichlet.h – Dirichlet L-functions, Riemann zeta and related functions	782
9.22.1	Roots of unity	782
9.22.2	Truncated L-series and power sums	782
9.22.3	Riemann zeta function	783
9.22.4	Riemann-Siegel formula	783
9.22.5	Hurwitz zeta function	784
9.22.6	Hurwitz zeta function precomputation	784
9.22.7	Lerch transcendent	785
9.22.8	Stieltjes constants	785
9.22.9	Dirichlet character evaluation	786
9.22.10	Dirichlet character Gauss, Jacobi and theta sums	786
9.22.11	Discrete Fourier transforms	788
9.22.12	Dirichlet L-functions	789
9.22.13	Hardy Z-functions	790
9.22.14	Gram points	791
9.22.15	Riemann zeta function zeros	791
9.22.16	Riemann zeta function zeros (Platt’s method)	792
9.23	bernoulli.h – support for Bernoulli numbers	794
9.23.1	Generation of Bernoulli numbers	794
9.23.2	Caching	795
9.23.3	Bounding	795
9.23.4	Isolated Bernoulli numbers	795
9.24	hypgeom.h – support for hypergeometric series	796
9.24.1	Strategy for error bounding	796
9.24.2	Types, macros and constants	797
9.24.3	Memory management	797
9.24.4	Error bounding	797
9.24.5	Summation	798
9.25	partitions.h – computation of the partition function	798
9.26	arb_calc.h – calculus with real-valued functions	799
9.26.1	Types, macros and constants	799
9.26.2	Debugging	800
9.26.3	Subdivision-based root finding	800
9.26.4	Newton-based root finding	801
9.27	acb_calc.h – calculus with complex-valued functions	802
9.27.1	Types, macros and constants	802
9.27.2	Integration	803
9.27.3	Local integration algorithms	805
9.27.4	Integration (old)	806
9.28	arb_fpwrap.h – floating-point wrappers of Arb mathematical functions	807
9.28.1	Option and return flags	807
9.28.2	Types	808

9.28.3	Functions	808
9.28.4	Calling from C	815
9.28.5	Interfacing from Python	816
9.28.6	Interfacing from Julia	817
9.29	fmpz_extras.h – extra methods for FLINT integers	817
9.29.1	Memory-related methods	818
9.29.2	Convenience methods	818
9.29.3	Inlined arithmetic	818
9.29.4	Low-level conversions	819
9.30	General formulas and bounds	819
9.30.1	Error propagation	819
9.30.2	Sums and series	820
9.30.3	Complex analytic functions	820
9.30.4	Euler-Maclaurin formula	821
9.31	Algorithms for mathematical constants	821
9.31.1	Pi	821
9.31.2	Logarithms of integers	821
9.31.3	Euler’s constant	822
9.31.4	Catalan’s constant	822
9.31.5	Apery’s constant	822
9.31.6	Khinchin’s constant	822
9.31.7	Glaisher’s constant	823
9.31.8	Reciprocal Fibonacci constant	823
9.32	Algorithms for the gamma function	823
9.32.1	The Stirling series	823
9.32.2	Rational arguments	824
9.33	Algorithms for the Hurwitz zeta function	824
9.33.1	Euler-Maclaurin summation	824
9.33.2	Parameter Taylor series	824
9.34	Algorithms for polylogarithms	825
9.34.1	Computation for small z	825
9.34.2	Expansion for general z	826
9.35	Algorithms for hypergeometric functions	826
9.35.1	Convergent series	826
9.35.2	Convergent series of power series	827
9.35.3	Asymptotic series for the confluent hypergeometric function	827
9.35.4	Asymptotic series for Airy functions	828
9.35.5	Corner case of the Gauss hypergeometric function	829
9.36	Algorithms for the arithmetic-geometric mean	830
9.36.1	Functional equation	830
9.36.2	AGM iteration	830
9.36.3	First derivative	830
9.36.4	Higher derivatives	831
10 Exact real and complex numbers		833
10.1	Introduction	833
10.1.1	Exact numbers in Calcium	833
10.1.2	FAQ	834
10.2	Calcium example programs	836
10.2.1	elementary.c	836
10.2.2	binet.c	837
10.2.3	machin.c	838
10.2.4	swinnerton_dyer_poly.c	838
10.2.5	huge_expr.c	839
10.2.6	hilbert_matrix.c	840
10.2.7	dft.c	840
10.3	calcium.h – global definitions	843
10.3.1	Version	843

10.3.2	Triple-valued logic	843
10.3.3	FLINT, Arb and Antic extras	843
10.3.4	Input and output	843
10.4	ca.h – exact real and complex numbers	844
10.4.1	Introduction: numbers	844
10.4.2	Introduction: special values	844
10.4.3	Number objects	845
10.4.4	Context objects	845
10.4.5	Memory management for numbers	846
10.4.6	Symbolic expressions	846
10.4.7	Printing	846
10.4.8	Special values	848
10.4.9	Assignment and conversion	848
10.4.10	Conversion of algebraic numbers	849
10.4.11	Random generation	849
10.4.12	Representation properties	850
10.4.13	Value predicates	851
10.4.14	Comparisons	852
10.4.15	Field structure operations	852
10.4.16	Arithmetic	852
10.4.17	Powers and roots	855
10.4.18	Complex parts	855
10.4.19	Exponentials and logarithms	857
10.4.20	Trigonometric functions	857
10.4.21	Special functions	859
10.4.22	Numerical evaluation	859
10.4.23	Rewriting and simplification	860
10.4.24	Factorization	860
10.4.25	Context options	861
10.4.26	Internal representation	863
10.5	ca_vec.h – vectors of real and complex numbers	864
10.5.1	Types, macros and constants	864
10.5.2	Memory management	864
10.5.3	Length	864
10.5.4	Assignment	865
10.5.5	Special vectors	865
10.5.6	Input and output	865
10.5.7	List operations	865
10.5.8	Arithmetic	865
10.5.9	Comparisons and properties	866
10.5.10	Internal representation	866
10.6	ca_poly.h – dense univariate polynomials over the real and complex numbers	867
10.6.1	Types, macros and constants	867
10.6.2	Memory management	867
10.6.3	Assignment and simple values	868
10.6.4	Random generation	868
10.6.5	Input and output	868
10.6.6	Degree and leading coefficient	869
10.6.7	Comparisons	869
10.6.8	Arithmetic	869
10.6.9	Evaluation and composition	870
10.6.10	Derivative and integral	871
10.6.11	Power series division	871
10.6.12	Elementary functions	871
10.6.13	Greatest common divisor	871
10.6.14	Roots and factorization	872
10.6.15	Vectors of polynomials	872
10.7	ca_mat.h – matrices over the real and complex numbers	873

10.7.1	Types, macros and constants	873
10.7.2	Memory management	873
10.7.3	Assignment and conversions	874
10.7.4	Random generation	874
10.7.5	Input and output	874
10.7.6	Special matrices	874
10.7.7	Comparisons and properties	875
10.7.8	Conjugate and transpose	875
10.7.9	Arithmetic	875
10.7.10	Powers	876
10.7.11	Polynomial evaluation	876
10.7.12	Gaussian elimination and LU decomposition	877
10.7.13	Solving and inverse	878
10.7.14	Rank and echelon form	878
10.7.15	Determinant and trace	879
10.7.16	Characteristic polynomial	880
10.7.17	Eigenvalues and eigenvectors	880
10.7.18	Jordan canonical form	880
10.7.19	Matrix functions	881
10.8	ca_ext.h – real and complex extension numbers	881
10.8.1	Type and macros	882
10.8.2	Memory management	883
10.8.3	Structure	883
10.8.4	Input and output	883
10.8.5	Numerical evaluation	884
10.8.6	Cache	884
10.9	ca_field.h – extension fields	885
10.9.1	Type and macros	885
10.9.2	Memory management	886
10.9.3	Input and output	887
10.9.4	Ideal	887
10.9.5	Structure operations	887
10.9.6	Cache	887
10.10	fexpr.h – flat-packed symbolic expressions	888
10.10.1	Introduction	888
10.10.2	Types and macros	889
10.10.3	Memory management	890
10.10.4	Size information	890
10.10.5	Comparisons	890
10.10.6	Atoms	891
10.10.7	Input and output	892
10.10.8	LaTeX output	892
10.10.9	Function call structure	892
10.10.10	Composition	893
10.10.11	Subexpressions and replacement	893
10.10.12	Arithmetic expressions	894
10.10.13	Vectors	895
10.11	fexpr_builtin.h – builtin symbols	896
10.11.1	C helper functions	896
10.11.2	Variables and iteration	896
10.11.3	Booleans and logic	897
10.11.4	Tuples, lists and sets	898
10.11.5	Numbers and arithmetic	899
10.11.6	Operators and calculus	903
10.11.7	Matrices and linear algebra	905
10.11.8	Polynomials, series and rings	905
10.11.9	Special functions	906

11	Finite fields	915
11.1	fq.h – finite fields	915
11.1.1	Types, macros and constants	915
11.1.2	Context Management	915
11.1.3	Memory management	916
11.1.4	Basic arithmetic	917
11.1.5	Roots	918
11.1.6	Output	918
11.1.7	Randomisation	919
11.1.8	Assignments and conversions	919
11.1.9	Comparison	920
11.1.10	Special functions	920
11.1.11	Bit packing	921
11.2	fq_default.h – unified finite fields	921
11.2.1	Types, macros and constants	921
11.2.2	Context Management	921
11.2.3	Memory management	923
11.2.4	Predicates	923
11.2.5	Basic arithmetic	923
11.2.6	Roots	924
11.2.7	Output	924
11.2.8	Randomisation	925
11.2.9	Assignments and conversions	925
11.2.10	Comparison	926
11.2.11	Special functions	926
11.3	fq_vec.h – vectors over finite fields	927
11.3.1	Memory management	927
11.3.2	Randomisation	927
11.3.3	Input and output	927
11.3.4	Assignment and basic manipulation	927
11.3.5	Comparison	927
11.3.6	Addition and subtraction	928
11.3.7	Scalar multiplication and division	928
11.3.8	Dot products	928
11.4	fq_mat.h – matrices over finite fields	928
11.4.1	Types, macros and constants	928
11.4.2	Memory management	928
11.4.3	Basic properties and manipulation	929
11.4.4	Conversions	929
11.4.5	Concatenate	930
11.4.6	Printing	930
11.4.7	Window	930
11.4.8	Random matrix generation	930
11.4.9	Comparison	931
11.4.10	Transpose	931
11.4.11	Addition and subtraction	931
11.4.12	Matrix multiplication	932
11.4.13	Inverse	932
11.4.14	LU decomposition	932
11.4.15	Reduced row echelon form	933
11.4.16	Triangular solving	933
11.4.17	Solving	933
11.4.18	Transforms	934
11.4.19	Characteristic polynomial	934
11.4.20	Minimal polynomial	934
11.5	fq_default_mat.h – matrices over finite fields	934
11.5.1	Types, macros and constants	934
11.5.2	Memory management	934

11.5.3	Basic properties and manipulation	935
11.5.4	Conversions	935
11.5.5	Concatenate	936
11.5.6	Printing	936
11.5.7	Window	936
11.5.8	Random matrix generation	937
11.5.9	Comparison	937
11.5.10	Transpose	938
11.5.11	Addition and subtraction	938
11.5.12	Matrix multiplication	938
11.5.13	Inverse	938
11.5.14	LU decomposition	938
11.5.15	Reduced row echelon form	939
11.5.16	Triangular solving	939
11.5.17	Solving	939
11.5.18	Transforms	940
11.5.19	Characteristic polynomial	940
11.5.20	Minimal polynomial	940
11.6	fq_poly.h – univariate polynomials over finite fields	940
11.6.1	Types, macros and constants	940
11.6.2	Memory management	940
11.6.3	Polynomial parameters	941
11.6.4	Randomisation	942
11.6.5	Assignment and basic manipulation	942
11.6.6	Getting and setting coefficients	943
11.6.7	Comparison	943
11.6.8	Addition and subtraction	943
11.6.9	Scalar multiplication and division	944
11.6.10	Multiplication	944
11.6.11	Squaring	947
11.6.12	Powering	948
11.6.13	Shifting	950
11.6.14	Norms	950
11.6.15	Euclidean division	950
11.6.16	Greatest common divisor	952
11.6.17	Divisibility testing	954
11.6.18	Derivative	954
11.6.19	Square root	954
11.6.20	Evaluation	955
11.6.21	Composition	955
11.6.22	Output	957
11.6.23	Inflation and deflation	958
11.7	fq_default_poly.h – univariate polynomials over finite fields	958
11.7.1	Types, macros and constants	958
11.7.2	Memory management	958
11.7.3	Polynomial parameters	959
11.7.4	Randomisation	959
11.7.5	Assignment and basic manipulation	960
11.7.6	Getting and setting coefficients	960
11.7.7	Comparison	961
11.7.8	Addition and subtraction	961
11.7.9	Scalar multiplication and division	962
11.7.10	Multiplication	962
11.7.11	Squaring	962
11.7.12	Powering	963
11.7.13	Shifting	963
11.7.14	Norms	963
11.7.15	Euclidean division	963

11.7.16	Greatest common divisor	964
11.7.17	Divisibility testing	964
11.7.18	Derivative	964
11.7.19	Square root	964
11.7.20	Evaluation	965
11.7.21	Composition	965
11.7.22	Output	965
11.7.23	Inflation and deflation	966
11.8	fq_poly_factor.h – factorisation of univariate polynomials over finite fields	966
11.8.1	Types, macros and constants	966
11.8.2	Memory Management	966
11.8.3	Basic Operations	966
11.8.4	Irreducibility Testing	967
11.8.5	Factorisation	967
11.8.6	Root Finding	969
11.9	fq_default_poly_factor.h – factorisation of univariate polynomials over finite fields	969
11.9.1	Types, macros and constants	969
11.9.2	Memory Management	969
11.9.3	Basic Operations	969
11.9.4	Irreducibility Testing	970
11.9.5	Factorisation	970
11.9.6	Root Finding	971
11.10	fq_embed.h – Computing isomorphisms and embeddings of finite fields	971
11.11	fq_nmod.h – finite fields (word-size characteristic)	973
11.11.1	Types, macros and constants	973
11.11.2	Context Management	973
11.11.3	Memory management	974
11.11.4	Basic arithmetic	975
11.11.5	Roots	976
11.11.6	Output	976
11.11.7	Randomisation	976
11.11.8	Assignments and conversions	977
11.11.9	Comparison	978
11.11.10	Special functions	978
11.11.11	Bit packing	979
11.12	fq_nmod_vec.h – vectors over finite fields (word-size characteristic)	979
11.12.1	Memory management	979
11.12.2	Randomisation	979
11.12.3	Input and output	979
11.12.4	Assignment and basic manipulation	980
11.12.5	Comparison	980
11.12.6	Addition and subtraction	980
11.12.7	Scalar multiplication and division	980
11.12.8	Dot products	981
11.13	fq_nmod_mat.h – matrices over finite fields (word-size characteristic)	981
11.13.1	Types, macros and constants	981
11.13.2	Memory management	981
11.13.3	Basic properties and manipulation	981
11.13.4	Conversions	982
11.13.5	Concatenate	982
11.13.6	Printing	983
11.13.7	Window	983
11.13.8	Random matrix generation	983
11.13.9	Comparison	984
11.13.10	Transpose	984
11.13.11	Addition and subtraction	984
11.13.12	Matrix multiplication	985
11.13.13	Inverse	985

11.13.14	LU decomposition	985
11.13.15	Reduced row echelon form	986
11.13.16	Triangular solving	986
11.13.17	Solving	986
11.13.18	Transforms	987
11.13.19	Characteristic polynomial	987
11.13.20	Minimal polynomial	987
11.14	fq_nmod_poly.h – univariate polynomials over finite fields (word-size characteristic)	987
11.14.1	Types, macros and constants	987
11.14.2	Memory management	987
11.14.3	Polynomial parameters	988
11.14.4	Randomisation	989
11.14.5	Assignment and basic manipulation	989
11.14.6	Getting and setting coefficients	990
11.14.7	Comparison	990
11.14.8	Addition and subtraction	991
11.14.9	Scalar multiplication and division	991
11.14.10	Multiplication	992
11.14.11	Squaring	995
11.14.12	Powering	996
11.14.13	Shifting	998
11.14.14	Norms	998
11.14.15	Euclidean division	998
11.14.16	Greatest common divisor	1001
11.14.17	Divisibility testing	1002
11.14.18	Derivative	1002
11.14.19	Square root	1003
11.14.20	Evaluation	1003
11.14.21	Composition	1003
11.14.22	Output	1006
11.14.23	Inflation and deflation	1007
11.15	fq_nmod_poly_factor.h – factorisation of univariate polynomials over finite fields (word-size characteristic)	1007
11.15.1	Types, macros and constants	1007
11.15.2	Memory Management	1008
11.15.3	Basic Operations	1008
11.15.4	Irreducibility Testing	1009
11.15.5	Factorisation	1009
11.15.6	Root Finding	1011
11.16	fq_nmod_embed.h – Computing isomorphisms and embeddings of finite fields	1011
11.17	fq_nmod_mpoly.h – multivariate polynomials over finite fields of word-sized characteristic	1012
11.17.1	Types, macros and constants	1012
11.17.2	Context object	1013
11.17.3	Memory management	1013
11.17.4	Input/Output	1013
11.17.5	Basic manipulation	1014
11.17.6	Constants	1014
11.17.7	Degrees	1015
11.17.8	Coefficients	1015
11.17.9	Comparison	1016
11.17.10	Container operations	1016
11.17.11	Random generation	1018
11.17.12	Addition/Subtraction	1018
11.17.13	Scalar operations	1018
11.17.14	Differentiation	1019
11.17.15	Evaluation	1019
11.17.16	Multiplication	1019

11.17.17	Powering	1020
11.17.18	Division	1020
11.17.19	Greatest Common Divisor	1020
11.17.20	Square Root	1021
11.17.21	Univariate Functions	1021
11.18	fq_nmod_mpoly_factor.h – factorisation of multivariate polynomials over finite fields of word-sized characteristic	1022
11.18.1	Types, macros and constants	1022
11.18.2	Memory management	1022
11.18.3	Basic manipulation	1022
11.18.4	Factorisation	1023
11.19	fq_zech.h – finite fields (Zech logarithm representation)	1023
11.19.1	Types, macros and constants	1023
11.19.2	Context Management	1024
11.19.3	Memory management	1025
11.19.4	Basic arithmetic	1026
11.19.5	Roots	1027
11.19.6	Output	1027
11.19.7	Randomisation	1027
11.19.8	Assignments and conversions	1028
11.19.9	Comparison	1029
11.19.10	Special functions	1029
11.19.11	Bit packing	1030
11.20	fq_zech_vec.h – vectors over finite fields (Zech logarithm representation)	1030
11.20.1	Memory management	1030
11.20.2	Randomisation	1030
11.20.3	Input and output	1030
11.20.4	Assignment and basic manipulation	1030
11.20.5	Comparison	1031
11.20.6	Addition and subtraction	1031
11.20.7	Scalar multiplication and division	1031
11.20.8	Dot products	1031
11.21	fq_zech_mat.h – matrices over finite fields (Zech logarithm representation)	1031
11.21.1	Types, macros and constants	1031
11.21.2	Memory management	1032
11.21.3	Basic properties and manipulation	1032
11.21.4	Conversions	1032
11.21.5	Concatenate	1033
11.21.6	Printing	1033
11.21.7	Window	1033
11.21.8	Random matrix generation	1033
11.21.9	Comparison	1034
11.21.10	Transpose	1034
11.21.11	Addition and subtraction	1035
11.21.12	Matrix multiplication	1035
11.21.13	LU decomposition	1036
11.21.14	Reduced row echelon form	1036
11.21.15	Triangular solving	1036
11.21.16	Solving	1036
11.21.17	Transforms	1037
11.21.18	Characteristic polynomial	1037
11.21.19	Minimal polynomial	1037
11.22	fq_zech_poly.h – univariate polynomials over finite fields (Zech logarithm representation)	1037
11.22.1	Types, macros and constants	1037
11.22.2	Memory management	1037
11.22.3	Polynomial parameters	1038
11.22.4	Randomisation	1039
11.22.5	Assignment and basic manipulation	1039

11.22.6	Getting and setting coefficients	1040
11.22.7	Comparison	1040
11.22.8	Addition and subtraction	1041
11.22.9	Scalar multiplication and division	1041
11.22.10	Multiplication	1042
11.22.11	Squaring	1044
11.22.12	Powering	1045
11.22.13	Shifting	1047
11.22.14	Norms	1048
11.22.15	Euclidean division	1048
11.22.16	Greatest common divisor	1050
11.22.17	Divisibility testing	1051
11.22.18	Derivative	1052
11.22.19	Square root	1052
11.22.20	Evaluation	1052
11.22.21	Composition	1053
11.22.22	Output	1055
11.22.23	Inflation and deflation	1056
11.23	fq_zech_poly_factor.h – factorisation of univariate polynomials over finite fields (Zech logarithm representation)	1057
11.23.1	Types, macros and constants	1057
11.23.2	Memory Management	1057
11.23.3	Basic Operations	1057
11.23.4	Irreducibility Testing	1058
11.23.5	Factorisation	1058
11.23.6	Root Finding	1060
11.24	fq_zech_embed.h – Computing isomorphisms and embeddings of finite fields	1060
12	p-adic numbers	1063
12.1	padic.h – p-adic numbers	1063
12.1.1	Introduction	1063
12.1.2	Data structures	1063
12.1.3	Context	1064
12.1.4	Memory management	1064
12.1.5	Randomisation	1065
12.1.6	Assignments and conversions	1065
12.1.7	Comparison	1066
12.1.8	Arithmetic operations	1066
12.1.9	Exponential	1067
12.1.10	Logarithm	1068
12.1.11	Special functions	1069
12.1.12	Input and output	1070
12.2	padic_poly.h – polynomials over p-adic numbers	1070
12.2.1	Module documentation	1070
12.2.2	Memory management	1070
12.2.3	Polynomial parameters	1071
12.2.4	Randomisation	1071
12.2.5	Assignment and basic manipulation	1072
12.2.6	Getting and setting coefficients	1073
12.2.7	Comparison	1073
12.2.8	Addition and subtraction	1073
12.2.9	Scalar multiplication	1074
12.2.10	Multiplication	1074
12.2.11	Powering	1074
12.2.12	Series inversion	1075
12.2.13	Derivative	1075
12.2.14	Shifting	1075
12.2.15	Evaluation	1075

12.2.16	Composition	1076
12.2.17	Input and output	1076
12.2.18	Testing	1077
12.3	padic_mat.h – matrices over p-adic numbers	1077
12.3.1	Module documentation	1077
12.3.2	Macros	1077
12.3.3	Memory management	1078
12.3.4	Basic assignment	1078
12.3.5	Conversions	1079
12.3.6	Entries	1079
12.3.7	Comparison	1079
12.3.8	Input and output	1079
12.3.9	Random matrix generation	1080
12.3.10	Transpose	1080
12.3.11	Addition and subtraction	1080
12.3.12	Scalar operations	1080
12.3.13	Multiplication	1081
12.4	qadic.h – unramified extensions over p-adic numbers	1081
12.4.1	Data structures	1081
12.4.2	Context	1081
12.4.3	Memory management	1082
12.4.4	Properties	1083
12.4.5	Randomisation	1083
12.4.6	Assignments and conversions	1083
12.4.7	Comparison	1084
12.4.8	Basic arithmetic	1084
12.4.9	Square root	1085
12.4.10	Special functions	1085
12.4.11	Output	1088
13	Floating-point support code	1089
13.1	double_extras.h – support functions for double arithmetic	1089
13.1.1	Random functions	1089
13.1.2	Arithmetic	1089
13.1.3	Special functions	1089
13.2	d_vec.h – double precision vectors	1090
13.2.1	Memory management	1090
13.2.2	Randomisation	1090
13.2.3	Assignment and basic manipulation	1090
13.2.4	Comparison	1090
13.2.5	Arithmetic	1091
13.2.6	Dot product and norm	1091
13.3	d_mat.h – double precision matrices	1091
13.3.1	Memory management	1091
13.3.2	Basic assignment and manipulation	1091
13.3.3	Random matrix generation	1092
13.3.4	Input and output	1092
13.3.5	Comparison	1092
13.3.6	Transpose	1092
13.3.7	Matrix multiplication	1092
14	Interfaces	1093
14.1	flint_ctypes – Python interface	1093
14.1.1	Introduction	1093
14.1.2	API documentation	1094
15	References	1095
15.1	References	1095

16 Version history	1097
16.1 History and changes	1097
16.1.1 FLINT version history	1097
16.1.2 Antic version history	1160
16.1.3 Calcium version history	1161
16.1.4 Arb version history	1165
Bibliography	1197
Index	1207

Welcome to FLINT's documentation! FLINT is a C library for doing number theory.

- Website: <https://flintlib.org>
- Source code on GitHub: <https://github.com/flintlib/flint>
- Issue tracker: <https://github.com/flintlib/flint/issues>
- Mailing list: <https://groups.google.com/group/flint-devel>

FLINT is free software distributed under the GNU Lesser General Public License (LGPL), version 3 or later.

INTRODUCTION

1.1 Introduction

1.1.1 What is FLINT?

FLINT is a C library of functions for doing basic arithmetic in support of computational number theory and other areas of computer algebra. It is highly optimised and can be compiled on numerous platforms.

FLINT provides highly optimised implementations of basic rings, such as the integers, rationals, p -adics, finite fields, etc., and linear algebra and univariate and multivariate polynomials over most of these rings.

FLINT also has some multithreading capabilities. To this end, the library is threadsafe, with few exceptions noted in the appropriate place, and a number of key functions have multithreaded implementations.

1.1.2 Maintainers and Authors

FLINT is currently maintained by Fredrik Johansson of INRIA Bordeaux.

FLINT was originally designed by William Hart and David Harvey. Since then FLINT was rewritten as FLINT 2 by William Hart, Fredrik Johansson and Sebastian Pancratz. Many other substantial contributions have been made by other authors, e.g. Tom Bachmann, Mike Hansen, Daniel Schultz and Andy Novocin. There have been a great number of other contributors, listed on the main FLINT website and the contributors section of this documentation.

1.1.3 Requirements

FLINT should compile on any machine with GCC and a standard GNU toolchain, though GCC 4.8 and following are recommended.

FLINT is specially optimised for x86 (32 and 64 bit) machines. There is also limited optimisation for ARM machines.

As of version 3.0, FLINT requires GMP 6.2.1 or later, and MPFR 4.1.0 or later. Note that earlier, MPIR, a fork of GMP, was supported. However, as of FLINT 3.0, this support has been dropped.

It is also required that the platform provide a `uint64_t` type if a native 64 bit type is not available. Full C99 compliance is not required.

1.1.4 Structure of FLINT

FLINT is supplied as a set of modules, `fmpz`, `fmpz_poly`, etc., each of which can be linked to a C program making use of their functionality.

All of the functions in FLINT have a corresponding test function provided in an appropriately named test file. For example, the function `fmpz_poly_add` located in `src/fmpz_poly/add.c` has test code in the file `src/fmpz_poly/test/t-add.c`.

Some modules have a `profile` directory in which profile programs can be found.

Documentation exists in the `doc/source` directory in a series of `.rst` files.

1.1.5 License

FLINT is distributed under the GNU Lesser General Public License (LGPL) version 3 or later. There is a copy of the license included in the repository and distribution tarballs.

Note, however, that between FLINT version 2.6 and 3.1, it was distributed with LGPL version 2, and before that GPL version 2.

1.2 Building, testing and installing

1.2.1 Quick start

Building FLINT requires:

- GMP, at least version 6.2.1 (<https://gmplib.org/>)
- MPFR, at least version 4.1.0 (<https://mpfr.org/>)
- Either of the following build systems:
 - GNU Make together with GNU Autotools (Recommended)
 - CMake (Only supported for Windows users)

If building from a release on a typical Linux or Unix-like system (see below for instructions using CMake), FLINT can be configured, built and installed as follows:

```
./configure
make -j N
make install
```

where `N` is the number of jobs number allowed to run parallel. Typically, the fastest way to build is to let `N` be the number of threads your CPU plus one, which can be obtained in Bash through `$(expr $(nproc) + 1)`.

If building from scratch, that is, without a `configure` script, then `configure` needs to be generated first. For this GNU Autotools needs to be installed in order to run

```
./bootstrap.sh
```

After this is done, `configure` should be generated and user can proceed with configuring, building and installing FLINT.

By default, FLINT only builds a shared library, but a static library can be built by pushing `--enable-static` to `configure`.

We also recommend that you check that the library works as it should through `make check`, or `make -j N check` for a parallel check, before installing.

For a complete list of build settings, type


```
./configure --help
```

An example of a custom configuration command would be

```
./configure \
--enable-assert \
--enable-avx2 \
--with-gmp-include=/home/user1/builds/includes/ \
--with-gmp-lib=/home/user1/builds/lib/ \
--with-mpfr=/usr \
--prefix=/home/user1/installations/ \
CC=clang \
CFLAGS="-Wall -O3 -march=alderlake"
```

1.2.2 Library and install paths

If you intend to install the FLINT library and header files, you can specify where they should be placed by passing `--prefix=path` to `configure`, where `path` is the directory under which the `lib` and `include` directories exist into which you wish to place the FLINT files when it is installed.

If GMP and MPFR are not installed in the default search path of your compiler (e.g. `/usr/include/` and `/usr/lib/`), you must specify where they are by passing their location to `configure` `--with-gmp=ABSOLUTE_PATH` for GMP and `--with-mpfr=ABSOLUTE_PATH` for MPFR. Note that the FLINT build system can handle GMP and MPFR as installed at some location and as source builds (built from source but not installed). Though, to run the FLINT tests, GMP and MPFR needs to be properly installed.

1.2.3 Testing FLINT

The full FLINT test suite can be run using

```
make check
```

or in parallel on a multicore system using

```
make -j check
```

Here, `make -j N check` is typically the fastest way to build when `N` equals to the number of threads your system's CPU has plus one, that is, `make -j $(expr $(nproc) + 1) check` typically is the fastest way to check FLINT.

Number of test iterations

The number of test iterations can be changed with the `FLINT_TEST_MULTIPLIER` environment variable. For example, the following will only run 10% of the default iterations:

```
export FLINT_TEST_MULTIPLIER=0.1
make check
```

Conversely, `FLINT_TEST_MULTIPLIER=10` will stress test FLINT by performing 10x the default number of iterations.

Testing single modules

If you wish to simply check a single module of FLINT you can pass the option `MOD=modname` to `make check`. You can also pass a list of module names:

```
make check MOD=ulong_extras
make -j N check MOD="fft fmpz_mat"
```

Testing single functions

Testing a single function is also possible, although one cannot utilize `make` all the way through for this. For example, if you would like to test the function `fmpz_add` and `fmpz_sub` in the module `fmpz`, you run

```
# Build all tests
make tests
# Run the test executable for `fmpz' with `fmpz_add' and `fmpz_sub' as inputs
./build/fmpz/test/main fmpz_add fmpz_sub
```

Test coverage

To obtain coverage statistics for the FLINT test suite, assuming that `gcov` and `lcov` are installed, configure FLINT with `--enable-coverage`. Then run:

```
make -j N check
make coverage_html
```

This will place a coverage report in `build/coverage`.

1.2.4 Static or dynamic library only

By default FLINT only builds a shared libraries by default. If you need to build a static library, you can pass `--enable-static` to `configure`. With this, `--disable-shared` can be passed as well to disable the build of a shared library, which will reduce the building time.

1.2.5 AVX2 instructions

On x86-64 machines with AVX2 support, compiling FLINT with the `--enable-avx2` option can improve performance substantially, notably by enabling the small-prime FFT. Currently this option is not enabled by default.

1.2.6 TLS, reentrancy and single mode

FLINT uses thread local storage by default (`--enable-tls`). However, if reentrancy is required on systems that do not support this, one can pass `--disable-tls` and mutexes will be used instead (requires POSIX). As most modern systems support thread local storage, it is not recommended to build FLINT without TLS.

There are two modes in which FLINT may installed: the default “single” mode, which is faster, but makes use of thread local storage for its memory manager and to handle threading, and a slower but less complicated “reentrant” mode. The later is useful when debugging a program where tracing allocations is important.

If you wish to select the single mode, pass the `--disable-reentrant` option to `configure`, though note that this is the default. The reentrant mode is selected by passing the option `--enable-reentrant` to `configure`.

1.2.7 ABI and architecture support

On some systems, e.g. Sparc and some Macs, more than one ABI is available. FLINT chooses the ABI based on the CPU type available, however its default choice can be overridden by passing either `ABI=64` or `ABI=32` to configure.

To build on MinGW64 it is necessary to pass `ABI=64` to configure, as FLINT is otherwise unable to distinguish it from MinGW32.

In some cases, it is necessary to override the CPU/OS defaults. This can be done by specifying the build system triplet to `configure` via `--build=arch-vendor-os`.

It is also possible to override the default `CC`, `AR` and `CFLAGS` used by FLINT by passing `CC=full_path_to_compiler`, etc., to FLINT's `configure`.

1.2.8 CMake build for Windows users

For Windows users, we also provide a way to install FLINT using CMake. Note, however, that FLINT's CMake script only exists to provide Windows users a way to install FLINT. For UNIX-type systems, please use Autotools along with GNU Make instead, as described at the top of this page.

If you wish to install FLINT with CMake on Windows, simply type:

```
mkdir build && cd build
cmake .. -DBUILD_SHARED_LIBS=ON
cmake --build . --target install
```

1.2.9 Uninstalling FLINT

To uninstall FLINT with GNU make, type:

```
make uninstall
```

Now to use FLINT, simply include the appropriate header files for the FLINT modules you wish to use in your C program. Then compile your program, linking against the FLINT library, GMP, MPFR and pthreads with the options `-lflint` `-lmpfr` `-lgmp` `-lpthread`.

To clean up the local build files, use:

```
make clean
make distclean
```

1.2.10 Assertion checking

FLINT has an assert system. If you want a debug build you can pass `--enable-assert` to configure. However, this will slow FLINT considerably, so asserts should not be enabled (`--disable-assert`, the default) for deployment.

1.2.11 Linking and running code

Here is an example program to get started using FLINT:

```
#include "flint/flint.h"
#include "flint/arb.h"

int main()
{
    arb_t x;
    arb_init(x);
    arb_const_pi(x, 50 * 3.33);
    arb_printn(x, 50, 0); flint_printf("\n");
    flint_printf("Computed with FLINT-%s\n", flint_version);
    arb_clear(x);
}
```

Compile it with:

```
gcc test.c -lflint
```

You may also have to pass the flags `-lmpfr` and `-lgmp` to the compiler. If the FLINT header and library files are not in a standard location such as `/usr/local`, you may also have to provide flags such as:

```
-I/path/to/flint -L/path/to/flint
```

Finally, to run the program, make sure that the linker can find `libflint`. If it is installed in a nonstandard location, you can for example add this path to the `LD_LIBRARY_PATH` environment variable.

The output of the example program should be something like the following:

```
[3.1415926535897932384626433832795028841971693993751 +/- 4.43e-50]
Computed with flint-3.0.0
```

1.3 Bug reporting

1.3.1 Reporting bugs

The maintainers wishes to be made aware of any and all bugs. Please open an issue at the GitHub repository (<https://github.com/flintlib/flint>) or send an email with your bug report to the FLINT devel list <https://groups.google.com/group/flint-devel>.

If possible please include details of your system, how FLINT was compiled/installed, the versions of GMP and MPFR as well as precise details of how to replicate the bug.

Note that FLINT needs to be linked against version 6.2.1 or later of GMP, version 4.1.0 or later of MPFR. Version 4.8 or later of GCC is recommended for parallel builds.

1.4 Contributing to FLINT

1.4.1 Code conventions

Four steps are needed to add a new function:

- Add the function `module_foo()` in a new file `src/module/foo.c`.
- Add a corresponding test program in a new file `src/module/test/t-foo.c`.
- Add the function prototype to `src/module.h`.
- Document the function in `doc/source/module.rst`.

The build system takes care of everything else automatically.

Test code (see below) can be omitted if `module_foo()` is a trivial helper function, but it should at least be tested indirectly via another function in that case. Auxiliary functions needed to implement `module_foo()` but which have no use elsewhere should be declared as `static` in `src/module/foo.c`. If `module_foo()` is very short, it can be declared inline directly in `module.h` with the `MODULE_INLINE` macro.

Use the following checklist regarding code style:

- Try to keep names and function arguments consistent with existing code.
- Follow the conventions regarding types, aliasing rules, etc. described in *Technical conventions and potential issues* and in `code_conventions.md`.
- Use basic FLINT constants, types and functions: `FLINT_BITS`, `flint_malloc/flint_free`, `flint_abort`, `flint_printf`, etc.
- Complex macros should be avoided.
- Indentation is four spaces.
- Curly braces normally go on a new line.
- Binary operators are surrounded by spaces (but parentheses and brackets are not).
- Logically distinct chunks of code (variable declarations, initialization, precomputations, the main loop, cleanup, etc.) should be separated by a single blank line.
- Lines are up to 79 characters long, but this rule can be broken if it helps readability.
- Add correct copyright notices at the top of each file.

1.4.2 Test code

The easiest way to write a test program for a new function is to adapt the test code for an existing, similar function.

Most of the test code in FLINT uses the strategy of computing the same result in two or more different ways (for example, using functional equations, interchanging the order of parameter, or varying the precision and other algorithm parameters) and verifying that the results are consistent. It is also a good idea to test that aliasing works. Input data is usually generated randomly, but in some cases including precomputed reference values also makes sense.

Faster test code is better. A single test program should not take more than 1 seconds to run with the default number of iterations, and preferably no more than 0.1 seconds. Most functions can be tested effectively in a few milliseconds. Think of what the corner cases are and try to generate random input biased toward such cases. The `randtest()` functions attempt to generate corner cases automatically, but some thought may be needed to use them optimally. Try to ensure that the test code fails if you deliberately break the tested function in any way. It is also a good idea to run the test code once with `FLINT_TEST_MULTIPLIER=10.0` or higher. If a function's input space is too large to probe effectively for

corner cases with random input, that can be a hint that the function should be split into smaller logical parts that can be tested separately.

The test code must complete without errors when run with `valgrind`. The most common mistake leading to memory corruption or memory leaks is to miss or duplicate an `init()` or `clear()` call. Check that the `init()` and `clear()` calls exactly match the variable declarations in each code block, including the test code itself.

Profiling code is not needed in most cases, but it is often a good idea to run some benchmarks at least during the initial development of a new feature. The `TIMEIT_START`/`TIMEIT_STOP` and `SHOW_MEMORY_USAGE` macros in FLINT are useful for quick measurements.

1.5 Contributors

1.5.1 Contributors

FLINT has only been possible due to an extraordinary number of high quality contributions from a vast array of people.

A complete list of contributors is available on the FLINT website at: <https://flintlib.org/authors.html>

If you believe your name is missing from this list, please contact us immediately on the `flint-devel` list. The list is updated at the time of each new release of FLINT.

1.6 Examples

1.6.1 Example programs

FLINT comes with example programs to demonstrate current and future FLINT features. To build the example programs run:

```
make examples
```

The example programs are built in the `build/examples` directory.

For Arb and Calcium there are separate example pages *Arb example programs* and *Calcium example programs*. Below are some general examples.

- `partitions` Demonstrates the partition counting code, e.g. `build/examples/partitions 1000000000` will compute the number of partitions of 10^9 .
- `delta_qexp` Computes the n -th term of the delta function, e.g. `build/examples/delta_qexp 1000000` will compute the one million-th term of the q -expansion of delta.
- `crt` Demonstrates the integer Chinese Remainder code, e.g. `build/examples/crt 10382788` will build up the given integer from its value mod various primes.
- `multi_crt` Demonstrates the fast tree version of the integer Chinese Remainder code, e.g. `build/examples/multi_crt 100493287498239 13` will build up the given integer from its value mod the given number of primes.
- `stirling_matrix` Generates Stirling number matrices of the first and second kind and computes their product, which should come out as the identity matrix. The matrices are printed to standard output. For example `build/examples/stirling_matrix 10` does this with 10 by 10 matrices.
- `fmpz_poly_factor_zassenhaus` Demonstrates the factorisation of a small polynomial. A larger polynomial is also provided on disk and a small (obvious) change to the example program will read this file instead of using the hard coded polynomial.

- `padic` Gives examples of the usage of many functions in the `padic` module.
- `fmpz_poly_q` Gives a very simple example of the `fmpz_poly_q` module.
- `fmpq_poly` Gives a very simple example of the `fmpq_poly` module.

1.7 Memory management

1.7.1 Memory allocation functions

The file `flint.h` defines functions `flint_malloc`, `flint_realloc`, `flint_calloc` and `flint_free`. They have the same interface as the standard library functions, but may perform additional error checking.

By default the memory allocation functions wrap the system's `malloc`, `realloc`, `calloc` and `free`. The user can override this behaviour by calling `__flint_set_memory_functions` passing the `malloc`, `realloc`, `calloc` and `free` function pointers as parameters (see `flint.h` for the exact prototype). The current memory functions can be returned in a similar manner by calling `__flint_get_memory_functions` passing the address of pointers in which the function pointers can be stored.

Memory allocated with `flint_malloc` must be freed with `flint_free` and not with `free`.

1.7.2 Global caches and cleanup

FLINT may cache some data (such as allocated integers and tables of prime numbers) to speed up various computations. If FLINT is built in threadsafe mode, most caches are thread-local (some are always global and shared among the threads).

Data cached by the current thread can be freed by calling the `flint_cleanup()` function. The user can register additional cleanup functions to be invoked by `flint_cleanup()` by passing a pointer to a function with signature `void cleanup_function(void)` to `flint_register_cleanup_function()`.

The user should call `flint_cleanup_master()` exactly once right before exiting a program. This cleans up all caches in all threads and should result in a clean output with tools like `valgrind` if there are no memory leaks.

1.7.3 Temporary allocation

FLINT allows for temporary allocation of memory using `alloca` to allocate on the stack if the allocation is small enough.

The following program demonstrates how to use this facility to allocate two different arrays.

```
#include <gmp.h>
#include "flint.h"

void myfun(void)
{
    /* other variable declarations */
    nn_ptr a, b;
    TMP_INIT;

    /* arbitrary code */

    TMP_START; /* we are about to do some allocation */
}
```

(continues on next page)

(continued from previous page)

```
/* arbitrary code */

a = TMP_ALLOC(32*sizeof(ulong));
b = TMP_ALLOC(64*sizeof(ulong));

/* arbitrary code */

TMP_END; /* cleans up a and b */

/* arbitrary code */
}
```

It is very important to note that temporary allocations should not be made in recursive functions or in loop bodies, as many small allocations on the stack can exhaust the stack causing a stack overflow.

1.8 Portability

1.8.1 Portable FLINT types

For platform independence, FLINT provides two types `ulong` and `slong` to replace `unsigned long` and `long` respectively. These are guaranteed to be the same size as GMP's `mp_limb_t` and `mp_limb_signed_t` types, respectively.

A full list of types provided by FLINT is available in `code_conventions.md` in the top-level source tree.

As FLINT supports Windows 64 on which the FLINT `ulong` and `slong` types are 64 bits, whilst `unsigned long` and `long` are only 32 bits, it is necessary to have a special format specifier which is 64 bits on Windows 64 instead of the usual `%lu` and `%ld`.

For this purpose FLINT provides its own I/O functions, `flint_printf`, `flint_fprintf`, `flint_sprintf`, `flint_scanf`, `flint_fscanf` and `flint_sscanf`, which work exactly as the usual system versions, but which take the `%wu` and `%wd` format specifiers, which support FLINT `ulong` and `slong` types respectively.

Also, instead of using constants `123UL` and `123L`, FLINT provides the macros `UWORD(123)` and `WORD(123)` respectively for constants of type `ulong` and `slong` respectively.

The maximum and minimum values that can be represented by these types are given by `UWORD_MAX` and `WORD_MAX` respectively.

1.9 Threading

1.9.1 Multithreaded FLINT

FLINT provides a number of multithreaded functions, which use multiple threads by default if FLINT was built with at least `pthread`. (This functionality works best when thread local storage is also available on the system.)

By default, FLINT will just use one thread. To control the maximum number of threads FLINT uses, one can call the function `flint_set_num_threads(n)`, where n is the maximum number of threads to use.

One can also query the current thread limit by calling `flint_get_num_threads()`.

Each version of FLINT brings new functions that are threaded by default.

Many core algorithms such as the FFT (for large integer and polynomial operations, including some factoring algorithms), integer factoring and multivariate polynomial algorithms are threaded in FLINT.

1.9.2 Writing threaded functions in FLINT

FLINT uses a custom thread pool for threading. This involves creating a worker function, requesting threads from the thread pool, starting the threads, waiting for them to finish, then giving the threads back to the pool. Simple examples of this include `fmpz_mod_mat_mul_classical_threaded` and `fmpz_poly_taylor_shift_multi_mod`.

The user should not have to run specialised versions of functions to get threading. This means that user facing functions should generally not have `_threaded` appended to their name. Either there is a single function that does the job, and it happens to be threaded, or there is a best-of-breed function that calls the appropriate threaded function when this is the best strategy.

There are some instances where it may be desirable (e.g. for testing purposes, or because naming proves difficult) where one wants a `_threaded` in the name. But these cases should be rare.

In some cases, one does not want functions to request threads from the pool themselves, but to accept threads from another function which has already obtained them. Such functions will accept an array of thread pool handles and a number of threads. The naming convention for such functions is to append `_threaded_pool` to the end of their name. However, the usual distinctions between underscore and non-underscore functions should still apply.

Functions should request `flint_get_num_threads()` threads from the thread pool. The function should not exceed this number of threads in total. In general a thread that is woken should start zero additional workers. However, if this is not the desired behaviour, an option exists to the function for waking worker threads to alter how many threads it can start. In some cases it is also necessary to temporarily restrict the number of worker threads a given function can start. This is accomplished by calling `flint_set_num_workers()` and then once the function is called, calling `flint_reset_num_workers()`. Any threaded function which calls `flint_get_num_threads()` to determine how many threads to request from the thread pool will be appropriately restricted by such calls.

Note that if `flint_get_num_threads()` returns `n` then the number of workers that can be started is `n - 1` (in addition to the thread the function is already running in). For this reason our documentation often distinguishes number of workers and number of threads. Please refer to the thread pool interface and FLINT threading interface documentation to see the exact specification.

1.9.3 Functional parallel programming helpers

The following convenience function are defined in `thread_support.h`. They are currently experimental, and the interfaces might change in a future version.

slong `flint_get_num_available_threads()`

Returns the number of threads that are not currently in use.

`typedef void (*do_func_t)(slong i, void *args)`

`void flint_parallel_do(do_func_t f, void *args, slong n, int thread_limit, int flags)`

Evaluate `f(i, args)` for $0 \leq i < n - 1$ in parallel using up to `thread_limit` threads (including the master thread). If `thread_limit` is nonpositive, the number of threads defaults to `flint_get_num_threads()`.

The following `flags` are supported:

`FLINT_PARALLEL_UNIFORM` - assumes that the cost of function calls is roughly constant, so that scheduling uniformly into blocks is efficient.

`FLINT_PARALLEL_STRIDED` - assumes that the cost increases or decreases monotonically with `i`, so that strided scheduling is efficient.

`FLINT_PARALLEL_DYNAMIC` (not implemented) - use dynamic scheduling.

FLINT_PARALLEL_VERBOSE - print information.

```
typedef void (*bsplit_merge_func_t)(void*, void*, void*, void*)
```

```
typedef void (*bsplit_basecase_func_t)(void*, slong, slong, void*)
```

```
typedef void (*bsplit_init_func_t)(void*, void*)
```

```
typedef void (*bsplit_clear_func_t)(void*, void*)
```

```
void flint_parallel_binary_splitting(void *res, bsplit_basecase_func_t basecase,  
                                     bsplit_merge_func_t merge, size_t sizeof_res,  
                                     bsplit_init_func_t init, bsplit_clear_func_t clear, void  
                                     *args, slong a, slong b, slong basecase_cutoff, int  
                                     thread_limit, int flags)
```

Sets `res` to $f(a) \circ f(a+1) \circ \dots \circ f(b-1)$ computed using parallel binary splitting, using up to `thread_limits` threads (including the master thread). If `thread_limit` is nonpositive, the number of threads defaults to `flint_get_num_threads()`.

The function `basecase(res, a, b, args)` gets called when $b - a$ does not exceed `basecase_cutoff`, which must be at least 1.

The function `merge(res, x, y, args)` implements the associative operation $(x \circ y)$, writing the result to `res`. If called with `FLINT_PARALLEL_BSPLIT_LEFT_INPLACE` in `flags`, the same space will be used for `res` and `x`.

A result is assumed to fit in a structure of size `sizeof_res`. The functions `init(res, args)` and `clear(res, args)` initialize and clear intermediate result objects.

GENERAL UTILITIES

2.1 flint.h – global definitions

2.1.1 Macros

The file `flint.h` contains various useful macros.

__FLINT_VERSION

__FLINT_VERSION_MINOR

__FLINT_VERSION_PATCHLEVEL

The major, minor and patch for current version of FLINT.

__FLINT_RELEASE

Equivalent to $10000 * \text{__FLINT\_VERSION} + 100 * \text{__FLINT\_VERSION\_MINOR} + \text{__FLINT\_VERSION\_PATCHLEVEL}$.

FLINT_VERSION

A static text string giving the version number, e.g. `3.1.0` or `3.2.0-dev`.

FLINT_BITS

The constant defining how many bits per limb on the machine. We require this to be either 32 or 64. This constant is set during the configuration.

FLINT_D_BITS

A constant set at compile time to be the number of bits per double on the machine or one less than the number of bits per limb, whichever is smaller. This will have the value *31* on 32-bit systems and *53* on 64-bit systems. Numerous internal functions using precomputed inverses only support operands up to `FLINT_D_BITS` bits, hence the macro.

FLINT_ABS(x)

Returns the absolute value of x for primitive signed numerical types. It might fail for least negative values such as `INT_MIN` and `LONG_MIN`.

FLINT_UABS(x)

Returns the absolute value of x for primitive signed numerical types, casting the result to an *ulong*. The result is well-defined for least negative values.

FLINT_MIN(x, y)

FLINT_MAX(x, y)

Returns the minimum or maximum of x and y for primitive types. This macro is only safe to use when x and y are of the same type, to avoid problems with integer promotion.

FLINT_SWAP(T, x, y)

Swaps x and y , both of types T . For instance, with x and y of type `fmpz_poly_t`, one can write `FLINT_SWAP(fmpz_poly_struct, *x, *y)` to swap the content of x with the content of y .

FLINT_SGN(*x*)

Returns the sign of *x* where *x* is interpreted as a *slong*, that is, returns -1 if $x < 0$, 0 if $x = 0$ and 1 if $x > 0$.

2.1.2 Integer types

The *char*, *short* and *int* types are assumed to be two's complement types with exactly 8, 16 and 32 bits. Although this is not guaranteed prior to C23, it is true on all mainstream platforms prior to this.

Since the C types *long* and *unsigned long* do not have a standardised size in practice, FLINT defines *slong* and *ulong* types which are guaranteed to be 32 bits on a 32-bit system and 64 bits on a 64-bit system. They are also guaranteed to have the same size as GMP's *mp_limb_t*. GMP builds with a different limb size configuration are not supported at all.

type *ulong*

The *ulong* type is used for integer-valued coefficients that are known to be unsigned, and for values that require the full 32-bit or 64-bit range. In method names, a *ulong* parameter is denoted by *ui*, for example *arb_add_ui()*.

The constant *UWORD_MAX* gives the range of this type. This type can be printed with *flint_printf* using the format string *%wu*.

This is equivalent to GMP's *mp_limb_t*.

type *slong*

The *slong* type is used for precisions, loop indices, array sizes, and the like, even when those values are known to be nonnegative. It is also used for small integer-valued coefficients. In method names, an *slong* parameter is denoted by *si*, for example *arb_add_si()*.

This type can be printed with *flint_printf* using the format string *%wd* or *%{slong}*.

This is equivalent to GMP's *mp_limb_signed_t*. Furthermore, for UNIX-type systems it is also equivalent to *mp_size_t*.

UWORD_MIN**UWORD_MAX****WORD_MIN****WORD_MAX**

The minimum and maximum values that a *ulong* and *slong* can hold, respectively.

type *flint_bitcnt_t*

A bit offset within an array of limbs (always nonnegative).

type *nn_ptr*

Pointer to a writable array of limbs.

This is equivalent to GMP's *mp_ptr*.

type *nn_srcptr*

Pointer to a read-only array of limbs.

This is equivalent to GMP's *mp_srcptr*.

2.1.3 Allocation Functions

void **flint_malloc**(size_t size)

Allocate *size* bytes of memory.

void **flint_realloc**(void *ptr, size_t size)

Reallocate an area of memory previously allocated by *flint_malloc()*, *flint_realloc()*, or *flint_calloc()*.

void **flint_calloc**(size_t num, size_t size)

Allocate *num* objects of *size* bytes each, and zero the allocated memory.

void **flint_free**(void *ptr)

Free a section of memory allocated by *flint_malloc()*, *flint_realloc()*, or *flint_calloc()*.

2.1.4 Random Numbers

type **flint_rand_struct**

A structure holding the state of the FLINT pseudo random number generator.

type **flint_rand_t**

An array of length 1 of *flint_rand_struct*.

void **flint_rand_init**(*flint_rand_t* state)

void **flint_rand_clear**(*flint_rand_t* state)

Initialises or clears a *flint_rand_t*.

2.1.5 Thread functions

void **flint_set_num_threads**(int num_threads)

Set up a thread pool of **num_threads** - 1 worker threads (in addition to the master thread) and set the maximum number of worker threads the master thread can start to **num_threads** - 1.

This function may only be called globally from the master thread. It can also be called at a global level to change the size of the thread pool, but an exception is raised if the thread pool is in use (threads have been woken but not given back). The function cannot be called from inside worker threads.

int **flint_get_num_threads**(void)

When called at the global level, this function returns one more than the number of worker threads in the FLINT thread pool, i.e. it returns the number of workers in the thread pool plus one for the master thread.

In general, this function returns one more than the number of additional worker threads that can be started by the current thread.

Use *thread_pool_wake()* to set this number for a given worker thread.

See also: *flint_get_num_available_threads()*.

int **flint_set_num_workers**(int num_workers)

Restricts the number of worker threads that can be started by the current thread to **num_workers**. This function can be called from any thread.

Assumes that the FLINT thread pool is already set up.

The function returns the old number of worker threads that can be started.

The function can only be used to reduce the number of workers that can be started from a thread. It cannot be used to increase the number. If a higher number is passed, the function has no effect.

The number of workers must be restored to the original value by a call to `flint_reset_num_workers()` before the thread is returned to the thread pool.

The main use of this function and `flint_reset_num_workers()` is to cheaply and temporarily restrict the number of workers that can be started, e.g. by a function that one wishes to call from a thread, and cheaply restore the number of workers to its original value before exiting the current thread.

void `flint_reset_num_workers`(int num_workers)

After a call to `flint_set_num_workers()` this function must be called to set the number of workers that may be started by the current thread back to its original value.

2.1.6 Input/Output

int `flint_printf`(const char *format, ...)

int `flint_fprintf`(FILE *fs, const char *format, ...)

int `flint_vprintf`(const char *format, va_list vlist)

int `flint_vfprintf`(FILE *fs, const char *format, va_list vlist)

These functions are extensions of the C standard library functions `printf`, `fprintf`, `vprintf`, and `vfprintf`.

The first extension is the addition of the length modifier `w`, used for printing the types `ulong`, `slong` and `ulong`. As these types are either defined as signed and unsigned long int or long long int, this comes in handy. Just like long int and long long int, the conversion format specifier are allowed to be `d`, `i`, `o`, `x`, `X` and `u`.

The second and final extension is printing of FLINT types. Currently supported types are the base types `ulong`, `slong`, `fmpz_t`, `fmpq_t`, `mag_t`, `arf_t`, `arb_t` and `acb_t` as well as the context structures for modulo arithmetic `nmod_t` and `fmpz_mod_ctx_t`. We also support the GMP types `mpz_t` and `mpq_t`.

We currently support printing vectors of pointers to the following base types: `slong`, `ulong`, `fmpz`, `fmpq`, `mag_struct`, `arf_struct`, `arb_struct` and `acb_struct`. In this case the nonnegative length of the vector must be passed as a second parameter following the pointer. Warning: the length parameter must be passed as a `slong`, not `int`.

We also support printing matrices of the following types: `nmod_mat_t`, `fmpz_mat_t`, `fmpq_mat_t`, `arb_mat_t` and `acb_mat_t`.

We also support printing polynomial of the following types: `nmod_poly_t`, `fmpz_poly_t`, `fmpq_poly_t`, `arb_poly_t` and `acb_poly_t`.

Triple-valued booleans of type `truth_t` can also be printed.

Finally, we support printing generic elements of type `gr_ptr` as well as `gr_poly_t` and `gr_mat_t`. For each of these types, the object to be printed must be followed by the corresponding `gr_ctx_t`. The context object itself can also be printed as a standalone object.

```
ulong bulong;
slong bslong;
fmpz_t bfmpz;
fmpq_t bfmpq;
mag_t bmag;
arf_t barf;
arb_t barb;
acb_t bacb;
nmod_t bnmod;
fmpz_mod_ctx_t bfmpz_mod_ctx;
mpz_t bmpz;
mpq_t bmpq;
```

(continues on next page)

(continued from previous page)

```

truth_t btruth;
gr_ctx_t bgr_ctx;
gr_ptr bgr;

/* Initialize and set variables */

flint_printf(
    "ulong: %ulong\n"
    "slong: %slong\n"
    "fmpz: %fmpz\n"
    "fmpq: %fmpq\n"
    "mag: %mag\n"
    "arf: %arf\n"
    "arb: %arb\n"
    "acb: %acb\n"
    "nmod: %nmod\n"
    "fmpz_mod_ctx: %fmpz_mod_ctx\n"
    "mpz: %mpz\n"
    "mpq: %mpq\n"
    "truth: %truth\n"
    "gr: %gr\n"
    "gr: %gr_ctx\n",
    bulong,
    bslong,
    bfmpz,
    bfmpq,
    bmag,
    barf,
    barb,
    bacb,
    bnmod,
    bfmpz_mod_ctx,
    bmpz,
    bmpq,
    btruth,
    bgr, bgr_ctx,
    bgr_ctx);

```

```

slong * vslong; slong vslong_len;
nn_ptr vnmod; slong vnmod_len; /* The base type for nmod is ulong */
fmpz * vfmpz; slong vfmpz_len;
/* fmpz_mod vectors are given by the type `fmpz *' */
fmpq * vfmpq; slong vfmpq_len;
mag_ptr vmag; slong vmag_len;
arf_ptr varf; slong varf_len;
arb_ptr varb; slong varb_len;
acb_ptr vacb; slong vacb_len;
gr_ptr vgr; slong vgr_len; gr_ctx_t vgr_ctx;

/* Initialize and set variables */

flint_printf(
    "slong vector: %slong*\n"
    "nmod vector: %ulong*\n"
    "fmpz vector: %fmpz*\n"
    "fmpq vector: %fmpq*\n"

```

(continues on next page)

(continued from previous page)

```

"mag vector: {%mag*}\n"
"arf vector: {%arf*}\n"
"arb vector: {%arb*}\n"
"acb vector: {%acb*}\n"
"gr vector: {%gr*}\n"
vslong, vslong_len, /* They require a vector length specifier */
vnmod, vnmod_len,
vmpz, vmmpz_len,
vmppq, vmppq_len,
vmag, vmag_len,
varf, varf_len,
varb, varb_len,
vacb, vacb_len,
vgr, vgr_len, vgr_ctx);

```

```

nmod_mat_t mnmod;
fmpz_mat_t mfmpz;
fmpz_mod_mat_t mfmpz_mod;
fmpq_mat_t mfmpq;
arb_mat_t marb;
acb_mat_t macb;
gr_mat_t mgr; gr_ctx_t mgr_ctx;

/* Initialize and set variables */

flint_printf(
    "nmod matrix: {%nmod_mat}\n"
    "fmpz matrix: {%fmpz_mat}\n"
    "fmpz_mod matrix: {%fmpz_mod_mat}\n"
    "fmpq matrix: {%fmpq_mat}\n"
    "arb matrix: {%arb_mat}\n"
    "acb matrix: {%acb_mat}\n"
    "gr matrix: {%gr_mat}\n"
    mnmod,
    mfmpz,
    mfmpz_mod,
    mfmpq,
    marb,
    macb,
    mgr, mgr_ctx);

```

```

nmod_poly_t pnmod;
fmpz_poly_t pfmpz;
fmpz_mod_poly_t pfmpz_mod;
fmpq_poly_t pfmpq;
arb_poly_t parb;
acb_poly_t pacb;
gr_poly_t pgr; gr_ctx_t pgr_ctx;

/* Initialize and set variables */

flint_printf(
    "nmod polynomial: {%nmod_poly}\n"
    "fmpz polynomial: {%fmpz_poly}\n"
    "fmpz_mod polynomial: {%fmpz_mod_poly}\n"

```

(continues on next page)

(continued from previous page)

```

"fmpq polynomial: %{fmpq_poly}\n"
"arb polynomial: %{arb_poly}\n"
"acb polynomial: %{acb_poly}\n"
pnmod,
pfmpz,
pfmpz_mod,
pfmpq,
parb,
pacb,
pgr, pgr_ctx);

```

Note: Printing of FLINT types does not currently support any flags.

Note: Any use of `%n` flags will be invalid, but will not generate any error.

Note: Invalid formats using variable minimum field width and/or precision such as `"%* p"` may be wrongly parsed, and may result in a different result compared to the C standard library functions.

int **flint_sprintf**(char *s, const char *str, ...)

This functions is an extensions of the C standard library functions `sprintf`. It is currently advised to not use this function as it is currently not coherent with `flint_printf()`.

int **flint_scanf**(const char *str, ...)

int **flint_fscanf**(FILE *f, const char *str, ...)

int **flint_sscanf**(const char *s, const char *str, ...)

These are equivalent to the standard library functions `scanf`, `fscanf`, and `sscanf` with an additional length modifier "w" for reading an *ulong* type.

2.1.7 Exceptions

enum **flint_err_t**

An error code with one of the following values

FLINT_ERROR

Describes a generic error.

FLINT_OVERFLOW

Describes an overflow.

FLINT_IMPINV

Describes an impossible inversion.

FLINT_DOMERR

Describes a domain error.

FLINT_DIVZERO

Describes a division by zero.

FLINT_EXPOF

Describes a exponent overflow.

FLINT_INEXACT

Describes an inexact operation.

FLINT_TEST_FAIL

Describes a test fail.

void **flint_abort**(void)

FLINT's function for aborting, which defaults to the C standard function `abort`. To redirect what function should be used for the abort, use `flint_set_abort()`.

void **flint_throw**(*flint_err_t* exc, const char *msg, ...)

FLINT's function for throwing, which is throwing an error of type `exc` with message `msg`. This defaults to printing `exc` followed by printing the message `msg`, to then abort via `flint_abort()`, where the back-end printing function is `flint_fprintf()` that allows printing of FLINT types.

To redirect what function should be used for throwing, use `flint_set_throw()`.

void **flint_set_abort**(void (*func)(void))

Sets the `flint_abort()` function to call `func` instead of `abort`.

void **flint_set_throw**(void (*func)(*flint_err_t*, const char*, va_list))

Sets the `flint_throw()` function use `func` instead of a private throw function. `func` is expected to call `va_end`, and to not alter *constchar** in any way.

2.1.8 Sorting and searching

void **flint_merge_sort**(void *buf, *slong* len, *slong* size, int (*cmp)(const void*, const void*, void*), void *data)

void **flint_sort**(void *buf, *slong* len, *slong* size, int (*cmp)(const void*, const void*, void*), void *data)

Sorts an array *buf* with *len* elements of size *size* according to the comparison function *cmp*. The comparison function takes as arguments pointers to the two elements being compared and a pointer to arbitrary data passed to the sorting function. It must return an *int* less than, equal to or greater than zero according to whether the first element is strictly smaller than, equal to, or strictly larger than the second.

The `merge_sort` version uses a simple implementation of the merge sort algorithm. The generic version uses the system's `qsort_r` (or equivalent) function when detected and falls back to `flint_merge_sort` otherwise.

2.2 profiler.h – performance profiling

Most of this module is only usable on certain platforms, while `timeit` functions is available on all platforms. The only platforms that can access the whole module is x86-64 and ARM64 on Unix type systems.

2.2.1 Timer based on clock

void **timeit_start**(timeit_t t)

void **timeit_stop**(timeit_t t)

void **timeit_print**(timeit_t t, *ulong* reps)

Gives wall and user time - useful for parallel programming.

Example usage:

```

timeit_t t0;
timeit_start(t0);
// do stuff, take some time
timeit_stop(t0);
timeit_print(t0, 1); // only one repetition was done

```

TIMEIT_REPEAT(timer, reps)

TIMEIT_END_REPEAT(timer, reps)

Repeatedly runs the code between the **TIMEIT_REPEAT** and the **TIMEIT_END_REPEAT** markers, automatically increasing the number of repetitions until the elapsed time exceeds the timer resolution. The macro takes as input a predefined `timeit_t` object and an integer variable to hold the number of repetitions.

TIMEIT_START

TIMEIT_STOP

Repeatedly runs the code between the **TIMEIT_START** and the **TIMEIT_STOP** markers, automatically increasing the number of repetitions until the elapsed time exceeds the timer resolution, and then prints the average elapsed cpu and wall time for a single repetition.

TIMEIT_ONCE_START

TIMEIT_ONCE_STOP

Runs the code between the **TIMEIT_ONCE_START** and the **TIMEIT_ONCE_STOP** markers exactly once and then prints the elapsed cpu and wall time. This does not give a precise measurement if the elapsed time is short compared to the timer resolution.

2.2.2 Timer based on cycle counter

void **init_clock**(int n)

void **init_all_clocks**(void)

void **start_clock**(int n)

void **stop_clock**(int n)

double **get_clock**(int n)

Gives time based on cycle counter.

First one must ensure the processor speed in cycles per second is set correctly in `profiler.h`, in the macro definition `#define FLINT_CLOCKSPEED`.

One can access the cycle counter directly by `get_cycle_counter()` which returns the current cycle counter as a `double`.

A sample usage of clocks is:

```

init_all_clocks();
start_clock(n);
// do something
stop_clock(n);
flint_printf("Time in seconds is %f.3\n", get_clock(n));

```

where `n` is a clock number (from 0-19 by default). The number of clocks can be changed by altering `FLINT_NUM_CLOCKS`. One can also initialise an individual clock with `init_clock(n)`.

void **prof_repeat**(double *min, double *max, profile_target_t target, void *arg)

Allows one to automatically time a given function. Here is a sample usage:

Suppose one has a function one wishes to profile:

```
void myfunc(ulong a, ulong b);
```

One creates a struct for passing arguments to our function:

```
typedef struct { ulong a, b; } myfunc_t;
```

a sample function:

```
void sample_myfunc(void * arg, ulong count)
{
    myfunc_t * params = (myfunc_t *) arg;

    ulong a = params->a;
    ulong b = params->b;

    for (ulong i = 0; i < count; i++)
    {
        prof_start();
        myfunc(a, b);
        prof_stop();
    }
}
```

Then we do the profile:

```
double min, max;

myfunc_t params;

params.a = 3;
params.b = 4;

prof_repeat(&min, &max, sample_myfunc, &params);

flint_printf("Min time is %lf.3s, max time is %lf.3s\n", min, max);
```

If either of the first two parameters to `prof_repeat` is `NULL`, that value is not stored.

One may set the minimum time in microseconds for a timing run by adjusting `DURATION_THRESHOLD` and one may set a target duration in microseconds by adjusting `DURATION_TARGET` in `profiler.h`.

2.2.3 Memory usage

void **fprint_memory_usage**(FILE *fs)

void **print_memory_usage**(void)

Obtains information about the memory usage of the current process.

2.3 thread_pool.h – thread pool

2.3.1 Thread pool

type **thread_pool_t**

This is a thread pool.

type **thread_pool_handle**

This is a handle to a thread in a thread pool.

void **thread_pool_init**(*thread_pool_t* T, *slong* size)

Initialise T and create **size** sleeping threads that are available to work. If *size* $\leq$ 0 no threads are created and future calls to *thread_pool_request()* will return 0 (unless *thread_pool_set_size()* has been called).

slong **thread_pool_get_size**(*thread_pool_t* T)

Return the number of threads in T.

int **thread_pool_set_size**(*thread_pool_t* T, *slong* new_size)

If all threads in T are in the available state, resize T and return 1. Otherwise, return 0.

slong **thread_pool_request**(*thread_pool_t* T, *thread_pool_handle* *out, *slong* requested)

Put at most **requested** threads in the unavailable state and return their handles. The handles are written to out and the number of handles written is returned. These threads must be released by a call to *thread_pool_give_back*.

void **thread_pool_wake**(*thread_pool_t* T, *thread_pool_handle* i, int max_workers, void (*f)(void*), void *a)

Wake up a sleeping thread i and have it work on f(a). The thread being woken will be allowed to start **max_workers** additional worker threads. Usually this value should be set to 0.

void **thread_pool_wait**(*thread_pool_t* T, *thread_pool_handle* i)

Wait for thread i to finish working and go back to sleep.

void **thread_pool_give_back**(*thread_pool_t* T, *thread_pool_handle* i)

Put thread i back in the available state. This thread should be sleeping when this function is called.

void **thread_pool_clear**(*thread_pool_t* T)

Release any resources used by T. All threads should be given back before this function is called.

2.4 mpoly.h – support functions for multivariate polynomials

An array of type `ulong *` or `fmpz **` is used to communicate exponent vectors. These exponent vectors must have length equal to the number of variables in the polynomial ring. The element of this exponent vector at index 0 corresponds to the most significant variable in the monomial ordering. For example, if the polynomial is $7 \cdot x^2 \cdot y + 8 \cdot y \cdot z + 9$ and the variables are ordered so that $x > y > z$, the degree function will return $\{2, 1, 1\}$. Similarly, the exponent vector of the 0-index term of this polynomial is $\{2, 1, 0\}$, while the 2-index term has exponent vector $\{0, 0, 0\}$ and coefficient 9.

2.4.1 Orderings

type `ordering_t`

Represents one of the following supported term orderings:

`ORD_LEX`

The lexicographic ordering.

`ORD_DEGLEX`

The degree lexicographic ordering.

`ORD_DEGREVLEX`

The degree reverse lexicographic ordering.

type `mpoly_ctx_struct`

type `mpoly_ctx_t`

An `mpoly_ctx_struct` is a structure holding information about the number of variables and the term ordering of a multivariate polynomial.

void `mpoly_ctx_init`(*mpoly_ctx_t* ctx, *slong* nvars, const *ordering_t* ord)

Initialize a context for specified number of variables and ordering.

void `mpoly_ctx_clear`(*mpoly_ctx_t* mctx)

Clean up any space used by a context object.

ordering_t `mpoly_ordering_randtest`(*flint_rand_t* state)

Return a random term ordering.

void `mpoly_ctx_init_rand`(*mpoly_ctx_t* mctx, *flint_rand_t* state, *slong* max_nvars)

Initialize a context with a random choice for the ordering.

int `mpoly_ordering_isdeg`(const *mpoly_ctx_t* ctx)

Return 1 if the ordering of the given context is a degree ordering (deglex or degrevlex).

int `mpoly_ordering_isrev`(const *mpoly_ctx_t* ctx)

Return 1 if the ordering of the given context is a reverse ordering (currently only degrevlex).

void `mpoly_ordering_print`(*ordering_t* ord)

Print a string (either “lex”, “deglex” or “degrevlex”) to standard output, corresponding to the given ordering.

2.4.2 Monomial arithmetic

These functions in this section are **only provided as inline functions** as they are somewhat trivial. This is in order to minimize the FLINT binary.

```
void mpoly_monomial_add(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)
    Set (exp_ptr, N) to the sum of the monomials (exp2, N) and (exp3, N), assuming bits <=
    FLINT_BITS.

void mpoly_monomial_add_mp(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)
    Set (exp_ptr, N) to the sum of the monomials (exp2, N) and (exp3, N).

void mpoly_monomial_sub(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)
    Set (exp_ptr, N) to the difference of the monomials (exp2, N) and (exp3, N), assuming bits
    <= FLINT_BITS

void mpoly_monomial_sub_mp(ulong *exp_ptr, const ulong *exp2, const ulong *exp3, slong N)
    Set (exp_ptr, N) to the difference of the monomials (exp2, N) and (exp3, N).

int mpoly_monomial_overflows(ulong *exp2, slong N, ulong mask)
    Return true if any of the fields of the given monomial (exp2, N) has overflowed (or is negative).
    The mask is a word with the high bit of each field set to 1. In other words, the function returns 1
    if any word of exp2 has any of the nonzero bits in mask set. Assumes that bits <= FLINT_BITS.

int mpoly_monomial_overflows_mp(ulong *exp_ptr, slong N, flint_bitcnt_t bits)
    Return true if any of the fields of the given monomial (exp_ptr, N) has overflowed. Assumes that
    bits >= FLINT_BITS.

int mpoly_monomial_overflows1(ulong exp, ulong mask)
    As per mpoly_monomial_overflows with N = 1.

void mpoly_monomial_set(ulong *exp2, const ulong *exp3, slong N)
    Set the monomial (exp2, N) to (exp3, N).

void mpoly_monomial_swap(ulong *exp2, ulong *exp3, slong N)
    Swap the words in (exp2, N) and (exp3, N).

void mpoly_monomial_mul_ui(ulong *exp2, const ulong *exp3, slong N, ulong c)
    Set the words of (exp2, N) to the words of (exp3, N) multiplied by c.
```

2.4.3 Monomial comparison

These functions in this section are **only provided as inline functions** as they are somewhat trivial. This is in order to minimize the FLINT binary.

```
int mpoly_monomial_is_zero(const ulong *exp, slong N)
    Return 1 if (exp, N) is zero.

int mpoly_monomial_equal(const ulong *exp2, const ulong *exp3, slong N)
    Return 1 if the monomials (exp2, N) and (exp3, N) are equal.

void mpoly_get_cmpmask(ulong *cmpmask, slong N, ulong bits, const mpoly_ctx_t mctx)
    Get the mask (cmpmask, N) for comparisons. bits should be set to the number of bits in the
    exponents to be compared. Any function that compares monomials should use this comparison
    mask.

int mpoly_monomial_lt(const ulong *exp2, const ulong *exp3, slong N, const ulong *cmpmask)
    Return 1 if (exp2, N) is less than (exp3, N).

int mpoly_monomial_gt(const ulong *exp2, const ulong *exp3, slong N, const ulong *cmpmask)
    Return 1 if (exp2, N) is greater than (exp3, N).
```

int **mpoly_monomial_cmp**(const *ulong* *exp2, const *ulong* *exp3, *slong* N, const *ulong* *cmpmask)
Return 1 if (exp2, N) is greater than, 0 if it is equal to and -1 if it is less than (exp3, N).

2.4.4 Monomial divisibility

These functions in this section are **only provided as inline functions** as they are somewhat trivial. This is in order to minimize the FLINT binary.

int **mpoly_monomial_divides**(*ulong* *exp_ptr, const *ulong* *exp2, const *ulong* *exp3, *slong* N, *ulong* mask)
Return 1 if the monomial (exp3, N) divides (exp2, N). If so set (exp_ptr, N) to the quotient monomial. The mask is a word with the high bit of each bit field set to 1. Assumes that bits <= FLINT_BITS.

int **mpoly_monomial_divides_mp**(*ulong* *exp_ptr, const *ulong* *exp2, const *ulong* *exp3, *slong* N, *flint_bitcnt_t* bits)
Return 1 if the monomial (exp3, N) divides (exp2, N). If so set (exp_ptr, N) to the quotient monomial. Assumes that bits >= FLINT_BITS.

int **mpoly_monomial_divides1**(*ulong* *exp_ptr, const *ulong* exp2, const *ulong* exp3, *ulong* mask)
As per **mpoly_monomial_divides** with N = 1.

int **mpoly_monomial_divides_tight**(*slong* e1, *slong* e2, *slong* *prods, *slong* num)
Return 1 if the monomial e2 divides the monomial e1, where the monomials are stored using factorial representation. The array (prods, num) should consist of 1, $b_1, b_1 \times b_2, \dots$, where the b_i are the bases of the factorial number representation.

2.4.5 Basic manipulation

flint_bitcnt_t **mpoly_exp_bits_required_ui**(const *ulong* *user_exp, const *mpoly_ctx_t* mctx)
Returns the number of bits required to store user_exp in packed format. The returned number of bits includes space for a zeroed signed bit.

flint_bitcnt_t **mpoly_exp_bits_required_ffmpz**(const *fmpz* *user_exp, const *mpoly_ctx_t* mctx)
Returns the number of bits required to store user_exp in packed format. The returned number of bits includes space for a zeroed signed bit.

flint_bitcnt_t **mpoly_exp_bits_required_pfmmpz**(*fmpz* *const *user_exp, const *mpoly_ctx_t* mctx)
Returns the number of bits required to store user_exp in packed format. The returned number of bits includes space for a zeroed signed bit.

void **mpoly_max_fields_ui_sp**(*ulong* *max_fields, const *ulong* *poly_exps, *slong* len, *ulong* bits, const *mpoly_ctx_t* mctx)
Compute the field-wise maximum of packed exponents from poly_exps of length len and unpack the result into max_fields. The maximums are assumed to fit a ulong.

void **mpoly_max_fields_fmpz**(*fmpz* *max_fields, const *ulong* *poly_exps, *slong* len, *ulong* bits, const *mpoly_ctx_t* mctx)
Compute the field-wise maximum of packed exponents from poly_exps of length len and unpack the result into max_fields.

void **mpoly_max_degrees_tight**(*slong* *max_exp, *ulong* *exps, *slong* len, *slong* *prods, *slong* num)
Return an array of num integers corresponding to the maximum degrees of the exponents in the array of exponent vectors (exps, len), assuming that the exponent are packed in a factorial representation. The array (prods, num) should consist of 1, $b_1, b_1 \times b_2, \dots$, where the b_i are the bases of the factorial number representation. The results are stored in the array max_exp, with the entry corresponding to the most significant base of the factorial representation first in the array.


```
int mpoly_monomial_exists(slong *index, const ulong *poly_exps, const ulong *exp, slong len, slong
                        N, const ulong *cmpmask)
```

Returns true if the given exponent vector **exp** exists in the array of exponent vectors (**poly_exps**, **len**), otherwise, returns false. If the exponent vector is found, its index into the array of exponent vectors is returned. Otherwise, **index** is set to the index where this exponent could be inserted to preserve the ordering. The index can be in the range $[0, \text{len}]$.

```
void mpoly_search_monomials(slong **e_ind, ulong *e, slong *e_score, slong *t1, slong *t2, slong
                          *t3, slong lower, slong upper, const ulong *a, slong a_len, const ulong
                          *b, slong b_len, slong N, const ulong *cmpmask)
```

Given packed exponent vectors **a** and **b**, compute a packed exponent **e** such that the number of monomials in the cross product **a** X **b** that are less than or equal to **e** is between **lower** and **upper**. This number is stored in **e_store**. If no such monomial exists, one is chosen so that the number of monomials is as close as possible. This function assumes that 1 is the smallest monomial and needs three arrays **t1**, **t2**, and **t3** of the size as **a** for workspace. The parameter **e_ind** is set to one of **t1**, **t2**, and **t3** and gives the locations of the monomials in **a** X **b**.

2.4.6 Setting and getting monomials

```
int mpoly_term_exp_fits_ui(ulong *exps, ulong bits, slong n, const mpoly_ctx_t mctx)
```

Return whether every entry of the exponent vector of index *n* in **exps** fits into a *ulong*.

```
int mpoly_term_exp_fits_si(ulong *exps, ulong bits, slong n, const mpoly_ctx_t mctx)
```

Return whether every entry of the exponent vector of index *n* in **exps** fits into a *slong*.

```
void mpoly_get_monomial_ui(ulong *exps, const ulong *poly_exps, ulong bits, const mpoly_ctx_t
                        mctx)
```

Convert the packed exponent **poly_exps** of bit count **bits** to a monomial from the user's perspective. The exponents are assumed to fit a *ulong*.

```
void mpoly_get_monomial_ffmpz(fmpz *exps, const ulong *poly_exps, flint_bitcnt_t bits, const
                          mpoly_ctx_t mctx)
```

Convert the packed exponent **poly_exps** of bit count **bits** to a monomial from the user's perspective.

```
void mpoly_get_monomial_pfmmpz(fmpz **exps, const ulong *poly_exps, flint_bitcnt_t bits, const
                          mpoly_ctx_t mctx)
```

Convert the packed exponent **poly_exps** of bit count **bits** to a monomial from the user's perspective.

```
void mpoly_set_monomial_ui(ulong *exp1, const ulong *exp2, ulong bits, const mpoly_ctx_t mctx)
```

Convert the user monomial **exp2** to packed format using **bits**.

```
void mpoly_set_monomial_ffmpz(ulong *exp1, const fmpz *exp2, flint_bitcnt_t bits, const
                          mpoly_ctx_t mctx)
```

Convert the user monomial **exp2** to packed format using **bits**.

```
void mpoly_set_monomial_pfmmpz(ulong *exp1, fmpz *const *exp2, flint_bitcnt_t bits, const
                          mpoly_ctx_t mctx)
```

Convert the user monomial **exp2** to packed format using **bits**.

2.4.7 Packing and unpacking monomials

void **mpoly_pack_vec_ui**(*ulong* *exp1, const *ulong* *exp2, *ulong* bits, *slong* nfields, *slong* len)

Packs a vector **exp2** into {**exp1**} using a bit count of **bits**. No checking is done to ensure that the vector actually fits into **bits** bits. The number of fields in each vector is **nfields** and the total number of vectors to unpack is **len**.

void **mpoly_pack_vec_fmpz**(*ulong* *exp1, const *fmpz* *exp2, *flint_bitcnt_t* bits, *slong* nfields, *slong* len)

Packs a vector **exp2** into {**exp1**} using a bit count of **bits**. No checking is done to ensure that the vector actually fits into **bits** bits. The number of fields in each vector is **nfields** and the total number of vectors to unpack is **len**.

void **mpoly_unpack_vec_ui**(*ulong* *exp1, const *ulong* *exp2, *ulong* bits, *slong* nfields, *slong* len)

Unpacks vector **exp2** of bit count **bits** into **exp1**. The number of fields in each vector is **nfields** and the total number of vectors to unpack is **len**.

void **mpoly_unpack_vec_fmpz**(*fmpz* *exp1, const *ulong* *exp2, *flint_bitcnt_t* bits, *slong* nfields, *slong* len)

Unpacks vector **exp2** of bit count **bits** into **exp1**. The number of fields in each vector is **nfields** and the total number of vectors to unpack is **len**.

int **mpoly_repack_monomials**(*ulong* *exps1, *ulong* bits1, const *ulong* *exps2, *ulong* bits2, *slong* len, const *mpoly_ctx_t* mctx)

Convert an array of length **len** of exponents **exps2** packed using bits **bits2** into an array **exps1** using bits **bits1**. No checking is done to ensure that the result fits into bits **bits1**.

void **mpoly_pack_monomials_tight**(*ulong* *exp1, const *ulong* *exp2, *slong* len, const *slong* *mults, *slong* num, *slong* bits)

Given an array of possibly packed exponent vectors **exp2** of length **len**, where each field of each exponent vector is packed into the given number of bits, return the corresponding array of monomial vectors packed using a factorial numbering scheme. The “bases” for the factorial numbering scheme are given as an array of integers **mults**, the first entry of which corresponds to the field of least significance in each input exponent vector. Obviously the maximum exponent to be packed must be less than the corresponding base in **mults**.

The number of multipliers is given by **num**. The code only considers least significant **num** fields of each exponent vectors and ignores the rest. The number of ignored fields should be passed in **extras**.

void **mpoly_unpack_monomials_tight**(*ulong* *e1, *ulong* *e2, *slong* len, *slong* *mults, *slong* num, *slong* bits)

Given an array of exponent vectors **e2** of length **len** packed using a factorial numbering scheme, unpack the monomials into an array **e1** of exponent vectors in standard packed format, where each field has the given number of bits. The “bases” for the factorial numbering scheme are given as an array of integers **mults**, the first entry of which corresponds to the field of least significance in each exponent vector.

The number of multipliers is given by **num**. The code only considers least significant **num** fields of each exponent vectors and ignores the rest. The number of ignored fields should be passed in **extras**.

2.4.8 Chunking

```
void mpoly_main_variable_terms1(slong *i1, slong *n1, const ulong *exp1, slong l1, slong len1, slong
                                k, slong num, slong bits)
```

Given an array of exponent vectors (**exp1**, **len1**), each exponent vector taking one word of space, with each exponent being packed into the given number of bits, compute **l1** starting offsets **i1** and lengths **n1** (which may be zero) to break the exponents into chunks. Each chunk consists of exponents have the same degree in the main variable. The index of the main variable is given by **k**. The variables are indexed from the variable of least significance, starting from 0. The value **l1** should be the degree in the main variable, plus one.

2.4.9 Chained heap functions

```
int _mpoly_heap_insert(mpoly_heap_s *heap, ulong *exp, void *x, slong *next_loc, slong
                        *heap_len, slong N, const ulong *cmpmask)
```

Given a heap, insert a new node *x* corresponding to the given exponent into the heap. Heap elements are ordered by the exponent (**exp**, **N**), with the largest element at the head of the heap. A pointer to the current heap length must be passed in via **heap_len**. This will be updated by the function. Note that the index 0 position in the heap is not used, so the length is always one greater than the number of elements.

```
void _mpoly_heap_insert1(mpoly_heap1_s *heap, ulong exp, void *x, slong *next_loc, slong
                        *heap_len, ulong maskhi)
```

As per **_mpoly_heap_insert** except that **N = 1**, and **maskhi = cmpmask[0]**.

```
void *_mpoly_heap_pop(mpoly_heap_s *heap, slong *heap_len, slong N, const ulong *cmpmask)
```

Pop the head of the heap. It is cast to a **void ***. A pointer to the current heap length must be passed in via **heap_len**. This will be updated by the function. Note that the index 0 position in the heap is not used, so the length is always one greater than the number of elements. The **maskhi** and **masklo** values are zero except for degrevlex ordering, where they are as per the monomial comparison operations above.

```
void *_mpoly_heap_pop1(mpoly_heap1_s *heap, slong *heap_len, ulong maskhi)
```

As per **_mpoly_heap_pop1** except that **N = 1**, and **maskhi = cmpmask[0]**.

2.5 machine_vectors.h – SIMD-accelerated operations on fixed-length vectors

This module currently requires building FLINT with support for AVX2 or NEON instructions.

Some functions may require that vectors are aligned in memory.

2.5.1 Types

```
type vec1n
```

```
type vec2n
```

```
type vec4n
```

```
type vec8n
```

Vector with 1, 2, 4, or 8 *ulong* entries.

```
type vec1d
```

```
type vec2d
```

```
type vec4d
```

type **vec8d**

Vector with 1, 2, 4, or 8 double entries.

2.5.2 Printing

void **vec4d_print**(*vec4d* a)

void **vec4n_print**(*vec4n* a)

2.5.3 Access and conversions

vec1d **vec1d_load**(const double *a)

vec4d **vec4d_load**(const double *a)

vec8d **vec8d_load**(const double *a)

vec1d **vec1d_load_aligned**(const double *a)

vec4d **vec4d_load_aligned**(const double *a)

vec8d **vec8d_load_aligned**(const double *a)

vec1d **vec1d_load_unaligned**(const double *a)

vec4d **vec4d_load_unaligned**(const double *a)

vec8d **vec8d_load_unaligned**(const double *a)

vec4n **vec4n_load_unaligned**(const *ulong* *a)

vec8n **vec8n_load_unaligned**(const *ulong* *a)

void **vec1d_store**(double *z, *vec1d* a)

void **vec4d_store**(double *z, *vec4d* a)

void **vec8d_store**(double *z, *vec8d* a)

void **vec1d_store_aligned**(double *z, *vec1d* a)

void **vec4d_store_aligned**(double *z, *vec4d* a)

void **vec8d_store_aligned**(double *z, *vec8d* a)

void **vec1d_store_unaligned**(double *z, *vec1d* a)

void **vec4d_store_unaligned**(double *z, *vec4d* a)

void **vec4n_store_unaligned**(*ulong* *z, *vec4n* a)

void **vec8d_store_unaligned**(double *z, *vec8d* a)

double **vec4d_get_index**(*vec4d* a, const int i)

double **vec8d_get_index**(*vec8d* a, int i)

Extract the entry at index *i*.

vec1d **vec1d_set_d**(double a)

vec4d **vec4d_set_d**(double a)

vec4n **vec4n_set_n**(*ulong* a)

vec8d **vec8d_set_d**(double a)

vec8n **vec8n_set_n**(*ulong* a)

Set all entries to the same value.

vec4d **vec4d_set_d4**(double a0, double a1, double a2, double a3)

vec4n **vec4n_set_n4**(*ulong* a0, *ulong* a1, *ulong* a2, *ulong* a3)

vec8d **vec8d_set_d8**(double a0, double a1, double a2, double a3, double a4, double a5, double a6, double a7)

Create vector from distinct entries.

vec4n **vec4d_convert_limited_vec4n**(*vec4d* a)

vec8d **vec8n_convert_limited_vec8d**(*vec8n* a)

2.5.4 Permutations

vec4d **vec4d_unpacklo**(*vec4d* a, *vec4d* b)

vec4d **vec4d_unpackhi**(*vec4d* a, *vec4d* b)

vec4d **vec4d_permute_0_2_1_3**(*vec4d* a)

vec4d **vec4d_permute_3_1_2_0**(*vec4d* a)

vec4d **vec4d_permute_3_2_1_0**(*vec4d* a)

vec4d **vec4d_permute2_0_2**(*vec4d* a, *vec4d* b)

vec4d **vec4d_permute2_1_3**(*vec4d* a, *vec4d* b)

vec4d **vec4d_unpack_lo_permute_0_2_1_3**(*vec4d* u, *vec4d* v)

vec4d **vec4d_unpack_hi_permute_0_2_1_3**(*vec4d* u, *vec4d* v)

vec4d **vec4d_unpackhi_permute_3_1_2_0**(*vec4d* u, *vec4d* v)

vec4d **vec4d_unpacklo_permute_3_1_2_0**(*vec4d* u, *vec4d* v)

VEC4D_TRANSPOSE(z0, z1, z2, z3, a0, a1, a2, a3)

Sets the rows z to the transpose of the 4x4 matrix given by rows a.

2.5.5 Comparisons

int **vec1d_same**(double a, double b)

int **vec4d_same**(*vec4d* a, *vec4d* b)

int **vec8d_same**(*vec8d* a, *vec8d* b)

Check whether the vectors are equal.

vec4d **vec4d_cmp_ge**(*vec4d* a, *vec4d* b)

vec4d **vec4d_cmp_gt**(*vec4d* a, *vec4d* b)

Entrywise comparisons.

2.5.6 Arithmetic and basic operations

vec1d **vec1d_round**(*vec1d* a)

vec4d **vec4d_round**(*vec4d* a)

vec8d **vec8d_round**(*vec8d* a)

vec1d **vec1d_zero**()

vec4d **vec4d_zero**()

vec8d **vec8d_zero**()

vec1d **vec1d_one**()

vec4d **vec4d_one**()

```
vec8d vec8d_one()

vec1d vec1d_add(vec1d a, vec1d b)
vec1d vec1d_sub(vec1d a, vec1d b)
vec4d vec4d_add(vec4d a, vec4d b)
vec4d vec4d_sub(vec4d a, vec4d b)
vec4n vec4n_add(vec4n a, vec4n b)
vec4n vec4n_sub(vec4n a, vec4n b)
vec8d vec8d_add(vec8d a, vec8d b)
vec8d vec8d_sub(vec8d a, vec8d b)

vec1d vec1d_addsub(vec1d a, vec1d b)
vec4d vec4d_addsub(vec4d a, vec4d b)

vec1d vec1d_neg(vec1d a)
vec4d vec4d_neg(vec4d a)
vec8d vec8d_neg(vec8d a)

vec1d vec1d_abs(vec1d a)
vec4d vec4d_abs(vec4d a)

vec1d vec1d_max(vec1d a, vec1d b)
vec1d vec1d_min(vec1d a, vec1d b)
vec4d vec4d_max(vec4d a, vec4d b)
vec4d vec4d_min(vec4d a, vec4d b)
vec8d vec8d_max(vec8d a, vec8d b)
vec8d vec8d_min(vec8d a, vec8d b)

vec1d vec1d_mul(vec1d a, vec1d b)
vec4d vec4d_mul(vec4d a, vec4d b)
vec8d vec8d_mul(vec8d a, vec8d b)

vec1d vec1d_half(vec1d a)
vec4d vec4d_half(vec4d a)

vec1d vec1d_div(vec1d a, vec1d b)
vec4d vec4d_div(vec4d a, vec4d b)
vec8d vec8d_div(vec8d a, vec8d b)

vec1d vec1d_fmadd(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fmadd(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fmadd(vec8d a, vec8d b, vec8d c)

vec1d vec1d_fmsub(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fmsub(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fmsub(vec8d a, vec8d b, vec8d c)

vec1d vec1d_fnmadd(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fnmadd(vec4d a, vec4d b, vec4d c)
vec8d vec8d_fnmadd(vec8d a, vec8d b, vec8d c)

vec1d vec1d_fnmsub(vec1d a, vec1d b, vec1d c)
vec4d vec4d_fnmsub(vec4d a, vec4d b, vec4d c)
```

```
vec8d vec8d_fnmsub(vec8d a, vec8d b, vec8d c)
```

```
vec1d vec1d_blendv(vec1d a, vec1d b, vec1d c)
```

```
vec4d vec4d_blendv(vec4d a, vec4d b, vec4d c)
```

```
vec8d vec8d_blendv(vec8d a, vec8d b, vec8d c)
```

```
vec4n vec4n_bit_shift_right(vec4n a, ulong b)
```

```
vec8n vec8n_bit_shift_right(vec8n a, ulong b)
```

```
vec4n vec4n_bit_and(vec4n a, vec4n b)
```

```
vec8n vec8n_bit_and(vec8n a, vec8n b)
```

2.5.7 Modular arithmetic

These functions are used internally by the small-prime FFT. Some `double` variants assume an odd modulus $n < 2^{50}$. Other assumptions are not yet documented.

```
int vec1d_same_mod(vec1d a, vec1d b, vec1d n, vec1d ninv)
```

```
int vec4d_same_mod(vec4d a, vec4d b, vec4d n, vec4d ninv)
```

Return whether a and b are the same mod n .

```
vec1d vec1d_reduce_pm1no_to_0n(vec1d a, vec1d n)
```

```
vec1d vec4d_reduce_pm1no_to_0n(vec4d a, vec4d n)
```

```
vec8d vec8d_reduce_pm1no_to_0n(vec8d a, vec8d n)
```

Return $a \bmod n$ reduced to $[0, n)$ assuming $a \in (-n, n)$.

```
vec1d vec1d_reduce_to_pm1n(vec1d a, vec1d n, vec1d ninv)
```

```
vec4d vec4d_reduce_to_pm1n(vec4d a, vec4d n, vec4d ninv)
```

```
vec8d vec8d_reduce_to_pm1n(vec8d a, vec8d n, vec8d ninv)
```

Return $a \bmod n$ reduced to $[-n, n]$.

```
vec1d vec1d_reduce_to_pm1no(vec1d a, vec1d n, vec1d ninv)
```

```
vec4d vec4d_reduce_to_pm1no(vec4d a, vec4d n, vec4d ninv)
```

```
vec8d vec8d_reduce_to_pm1no(vec8d a, vec8d n, vec8d ninv)
```

Return $a \bmod n$ reduced to $(-n, n)$.

```
vec1d vec1d_reduce_0n_to_pmhn(vec1d a, vec1d n)
```

```
vec4d vec4d_reduce_0n_to_pmhn(vec4d a, vec4d n)
```

Return $a \bmod n$ reduced to $[-n/2, n/2]$ given $a \in [0, n]$.

```
vec1d vec1d_reduce_pm1n_to_pmhn(vec1d a, vec1d n)
```

```
vec4d vec4d_reduce_pm1n_to_pmhn(vec4d a, vec4d n)
```

```
vec8d vec8d_reduce_pm1n_to_pmhn(vec8d a, vec8d n)
```

Return $a \bmod n$ reduced to $[-n/2, n/2]$ given $a \in [-n, n]$.

```
vec1d vec1d_reduce_2n_to_n(vec1d a, vec1d n)
```

```
vec4d vec4d_reduce_2n_to_n(vec4d a, vec4d n)
```

```
vec8d vec8d_reduce_2n_to_n(vec8d a, vec8d n)
```

Return $a \bmod n$ reduced to $[0, n)$ given $a \in [0, 2n)$.

```
vec1d vec1d_reduce_to_0n(vec1d a, vec1d n, vec1d ninv)
```

```
vec4d vec4d_reduce_to_0n(vec4d a, vec4d n, vec4d ninv)
```

```
vec8d vec8d_reduce_to_0n(vec8d a, vec8d n, vec8d ninv)
```

Return $a \bmod n$ reduced to $[0, n)$.

```
vec1d vec1d_mulmod(vec1d a, vec1d b, vec1d n, vec1d ninv)
```

```
vec4d vec4d_mulmod(vec4d a, vec4d b, vec4d n, vec4d ninv)
```

vec8d **vec8d_mulmod**(*vec8d* a, *vec8d* b, *vec8d* n, *vec8d* ninv)

Return $ab \bmod n$ in $[-n, n]$ with assumptions.

vec1d **vec1d_nmulmod**(*vec1d* a, *vec1d* b, *vec1d* n, *vec1d* ninv)

vec4d **vec4d_nmulmod**(*vec4d* a, *vec4d* b, *vec4d* n, *vec4d* ninv)

vec8d **vec8d_nmulmod**(*vec8d* a, *vec8d* b, *vec8d* n, *vec8d* ninv)

Return $ab \bmod n$ in $[-n, n]$ with assumptions.

vec4n **vec4n_addmod**(*vec4n* a, *vec4n* b, *vec4n* n)

vec8n **vec8n_addmod**(*vec8n* a, *vec8n* b, *vec8n* n)

Return $a + b \bmod n$ in $[0, n)$

vec4n **vec4n_addmod_limited**(*vec4n* a, *vec4n* b, *vec4n* n)

vec8n **vec8n_addmod_limited**(*vec8n* a, *vec8n* b, *vec8n* n)

Return $a + b \bmod n$ in $[0, n)$, assuming that $n < 2^{63}$.

GENERIC RINGS

3.1 gr.h – generic structures and their elements

3.1.1 Introduction

Parents and elements

To work with an element $x \in R$ of a particular mathematical structure R , we use a context object to represent R (the “parent” of x). Elements are passed around as pointers. Note:

- Parents are not stored as part of the elements; the user must track the context objects for all variables.
- Operations are strictly type-stable: elements only change parent when performing an explicit conversion.

The structure R will typically be a *ring*, but the framework supports general objects (including groups, monoids, and sets without any particular structure whatsoever). We use these terms in a strict mathematical sense: a “ring” must exactly satisfy the ring axioms. It can have inexact *representations*, but this inexactness must be handled rigorously.

To give an idea of how the interface works, this example program computes 3^{100} in the ring of integers and prints the value:

```
#include "gr.h"

int main()
{
    int status;
    gr_ctx_t ZZ;           /* a parent (context object) */
    gr_ptr x;              /* an element */

    gr_ctx_init_fmpz(ZZ);   /* ZZ = ring of integers with fmpz_t elements */
    GR_TMP_INIT(x, ZZ)     /* allocate element on the stack */

    status = gr_set_ui(x, 3, ZZ);      /* x = 3 */
    status |= gr_pow_ui(x, x, 100, ZZ); /* x = x ^ 100 */
    status |= gr_println(x, ZZ);

    GR_TMP_CLEAR(x, ZZ)
    gr_ctx_clear(ZZ);

    return status;
}
```

Parent and element types

type **gr_ptr**

Pointer to a ring element or array of contiguous ring elements. This is an alias for `void *` so that it can be used with any C type.

type **gr_srcptr**

Pointer to a read-only ring element or read-only array of contiguous ring elements. This is an alias for `const void *` so that it can be used with any C type.

type **gr_ctx_struct**

type **gr_ctx_t**

A context object representing a mathematical structure R . It contains the following data:

- The size (number of bytes) of each element.
- A pointer to a method table.
- Optionally a pointer to data defining parameters of the ring (e.g. modulus of a residue ring; element ring and dimensions of a matrix ring; precision of an inexact ring).

A `gr_ctx_t` is defined as an array of length one of type `gr_ctx_struct`, permitting a `gr_ctx_t` to be passed by reference. Context objects are not normally passed as `const` in order to allow storing mutable caches, additional debugging information, etc.

type **gr_ctx_ptr**

Pointer to a context object.

There is no type to represent a single generic element as a struct since we do not know the size of a generic element at compile time. Memory for single elements can either be allocated on the stack with the special macros provided below, or as usual with `malloc`. Methods can also be used with particular C types like `fmpz_t` when the user knows the type. Users may wish to define their own union types when only some particular types will appear in an application.

void **gr_ctx_uninitialized**(`gr_ctx_t` ctx)

Create an uninitialized context. This context object does not support any operations, but it can be printed with `gr_ctx_print()` and cleared with `gr_ctx_clear()`. A context constructor that can fail (see for example `gr_ctx_init_mpn_mod()`) should call this method so that the context object is safe to clear even if initialization failed, i.e. ensuring that `ctx` contains a valid function pointer for `gr_ctx_clear()`.

Error handling

To compute over a structure R , it is useful to conceptually extend to a larger set $R' = R \cup \{\text{undefined}, \text{unknown}\}$.

- Adding an *undefined* (error) value allows us to extend partial functions to total functions.
- An *unknown* value is useful in cases where a result may exist in principle but cannot be computed.

An alternative to having an *undefined* value is to choose some arbitrary default value in R , say `undefined = 0` in a ring. This is often done in proof assistants, but in a regular programming environment, we typically want some way to detect domain errors.

Representing R' as a type-level extension of R is tricky in C since we would either have to wrap elements in a larger structure or reserve bit patterns in each type for special values. In any case, it is useful to assume in low-level code that elements *really represent elements of the intended structure* so that there are fewer special cases to handle. We also need some form of error handling for conversions to standard C types. For these reasons, we handle special values (undefined, unknown) using return codes.

Functions can return a combination of the following status flags:

GR_SUCCESS

The operation finished as expected, i.e. the result is a correct element of the target type.

GR_DOMAIN

The result does not have a value in the domain of the target ring or type, i.e. the result is mathematically undefined. This occurs, for example, on division by zero or when attempting to compute the square root of a non-square. It also occurs when attempting to convert a too large value to a bounded type (example: `get_ui()` with input $n \geq 2^{64}$).

GR_UNABLE

The operation could not be performed because of limitations of the implementation or the data representation, i.e. the result is unknown. Typical reasons:

- The result would be too large to fit in memory
- The inputs are inexact and an exact comparison is needed
- The computation would take too long
- An algorithm is not yet implemented for this case

If this flag is set, there is also potentially a domain error (but this is unknown).

GR_TEST_FAIL

Test failure. This is only used in test code.

When the status code is any other value than **GR_SUCCESS**, any output variables may be set to meaningless values.

C functions that return a status code are marked with the **WARN_UNUSED_RESULT** attribute. This allows compilers to emit warnings when the status code is ignored.

Flags can be OR'ed and checked only at the top level of a computation to avoid complex control flow:

```
status = GR_SUCCESS;
status |= gr_add(res, a, b, ctx);
status |= gr_pow_ui(res, res, 2, ctx);
...
```

If we do not care about recovering from *undefined/unknown* results, the following macro is useful:

GR_MUST_SUCCEED(expr)

Evaluates *expr* and asserts that the return value is **GR_SUCCESS**. On failure, calls `flint_abort()`.

For uniformity, most operations return a status code, even operations that are not typically expected to fail. Exceptions include the following:

- Pure “container” operations like `init`, `clear` and `swap` do not return a status code.
- Pure predicate functions (see below) return **T_TRUE** / **T_FALSE** / **T_UNKNOWN** instead of computing a separate boolean value and error code.

Predicates

We use the following type (borrowed from Calcium) instead of a C int to represent boolean results, allowing the possibility that the value is not computable:

```
enum truth_t
```

Represents one of the following truth values:

```
T_TRUE
```

```
T_FALSE
```

T_UNKNOWN

Warning: the constants `T_TRUE` and `T_FALSE` do not correspond to 1 and 0. It is erroneous to write, for example `!t` if `t` is a *truth_t*. One should instead write `t != T_TRUE`, `t == T_FALSE`, etc. depending on whether the unknown case should be included or excluded.

```
truth_t truth_and(truth_t x, truth_t y)
```

```
truth_t truth_or(truth_t x, truth_t y)
```

```
truth_t truth_not(truth_t x)
```

Performs the corresponding operation in ternary logic.

```
void truth_println(truth_t x)
```

```
truth_t gr_in_domain(int status)
```

Returns `T_TRUE` when `status` is `GR_SUCCESS`, `T_FALSE` when `GR_DOMAIN` is set but `GR_UNABLE` is not, and `T_UNKNOWN` otherwise.

```
int gr_check(truth_t t)
```

Maps `T_TRUE` to `GR_SUCCESS`, `T_FALSE` to `GR_DOMAIN`, and `T_UNKNOWN` to `GR_UNABLE`.

3.1.2 Context operations

```
slong gr_ctx_sizeof_elem(gr_ctx_t ctx)
```

Return `sizeof(type)` where `type` is the underlying C type for elements of `ctx`.

```
void gr_ctx_clear(gr_ctx_t ctx)
```

Clears the context object `ctx`, freeing any memory allocated by this object.

Some context objects may require that no elements are cleared after calling this method, and may leak memory if not all elements have been cleared when calling this method.

If `ctx` is derived from a base ring, the base ring context may also be required to stay alive until after this method is called.

```
int gr_ctx_write(gr_stream_t out, gr_ctx_t ctx)
```

```
int gr_ctx_print(gr_ctx_t ctx)
```

```
int gr_ctx_println(gr_ctx_t ctx)
```

```
int gr_ctx_get_str(char **s, gr_ctx_t ctx)
```

Writes a description of the structure `ctx` to the stream `out`, prints it to `stdout`, or sets `s` to a pointer to a heap-allocated string of the description (the user must free the string with `flint_free`). The *println* version prints a trailing newline.

```
int gr_ctx_set_gen_name(gr_ctx_t ctx, const char *s)
```

```
int gr_ctx_set_gen_names(gr_ctx_t ctx, const char **s)
```

Set the name of the generator (univariate polynomial ring, finite field, etc.) or generators (multivariate). The name is used when printing and may be used to choose coercions.

```
int gr_ctx_ngens(slong *ngens, gr_ctx_t ctx)
```

Get the number of generators.

```
int gr_ctx_gen_name(char **name, slong i, gr_ctx_t ctx)
```

Get the name of the generator of index `i`. The returned buffer must be freed with `flint_free`.

3.1.3 Element operations

Memory management

void **gr_init**(*gr_ptr* res, *gr_ctx_t* ctx)

Initializes *res* to a valid variable and sets it to the zero element of the ring *ctx*.

void **gr_clear**(*gr_ptr* res, *gr_ctx_t* ctx)

Clears *res*, freeing any memory allocated by this object.

void **gr_swap**(*gr_ptr* x, *gr_ptr* y, *gr_ctx_t* ctx)

Swaps *x* and *y* efficiently.

void **gr_set_shallow**(*gr_ptr* res, *gr_srcptr* x, *gr_ctx_t* ctx)

Sets *res* to a shallow copy of *x*, copying the struct data.

gr_ptr **gr_heap_init**(*gr_ctx_t* ctx)

Return a pointer to a single new heap-allocated element of *ctx* set to 0.

void **gr_heap_clear**(*gr_ptr* x, *gr_ctx_t* ctx)

Free the single heap-allocated element *x* of *ctx* which should have been created with **gr_heap_init**().

gr_ptr **gr_heap_init_vec**(*slong* len, *gr_ctx_t* ctx)

Return a pointer to a new heap-allocated vector of *len* initialized elements.

void **gr_heap_clear_vec**(*gr_ptr* x, *slong* len, *gr_ctx_t* ctx)

Clear the *len* elements in the heap-allocated vector *len* and free the vector itself.

The following macros support allocating temporary variables efficiently. Data will be allocated on the stack using **alloca** unless the size is excessive (risking stack overflow), in which case the implementation transparently switches to **malloc/free** instead. The usage pattern is as follows:

```
{
    gr_ptr x, y;
    GR_TMP_INIT2(x1, x2, ctx);

    /* do computations with x1, x2 */

    GR_TMP_CLEAR2(x1, x2, ctx);
}
```

Init and clear macros must match exactly, as variables may be allocated contiguously in a block.

Warning: never use these macros directly inside a loop. This is likely to overflow the stack, as memory will not be reclaimed until the function exits. Instead, allocate the needed space before entering any loops, move the loop body to a separate function, or allocate the memory on the heap if needed.

GR_TMP_INIT_VEC(vec, len, ctx)

GR_TMP_CLEAR_VEC(vec, len, ctx)

Allocates and frees a vector of *len* contiguous elements, all initialized to the value 0, assigning the first element to the pointer *vec*.

GR_TMP_INIT(x1, ctx)

GR_TMP_INIT2(x1, x2, ctx)

GR_TMP_INIT3(x1, x2, x3, ctx)

GR_TMP_INIT4(x1, x2, x3, x4, ctx)

GR_TMP_INIT5(x1, x2, x3, x4, x5, ctx)

Allocates one or several temporary elements, all initialized to the value 0, assigning the elements to the pointers *x1*, *x2*, etc.

```
GR_TMP_CLEAR(x1, ctx)
GR_TMP_CLEAR2(x1, x2, ctx)
GR_TMP_CLEAR3(x1, x2, x3, ctx)
GR_TMP_CLEAR4(x1, x2, x3, x4, ctx)
GR_TMP_CLEAR5(x1, x2, x3, x4, x5, ctx)
```

Corresponding macros to clear temporary variables.

Random elements

```
int gr_randtest(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)
    Sets res to a random element of the domain ctx. The distribution is determined by the implemen-
    tation. Typically the distribution is non-uniform in order to find corner cases more easily in test
    code.

int gr_randtest_invertible(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)
    Sets res to a random invertible element of the domain ctx.

int gr_randtest_not_zero(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)
    Sets res to a random nonzero element of the domain ctx. This operation will fail and return
    GR_DOMAIN in the zero ring.

int gr_randtest_small(gr_ptr res, flint_rand_t state, gr_ctx_t ctx)
    Sets res to a “small” element of the domain ctx. This is suitable for randomized testing where a
    “large” argument could result in excessive computation time.
```

Input, output and string conversion

```
int gr_write(gr_stream_t out, gr_srcptr x, gr_ctx_t ctx)
int gr_print(gr_srcptr x, gr_ctx_t ctx)
int gr_println(gr_srcptr x, gr_ctx_t ctx)
int gr_get_str(char **s, gr_srcptr x, gr_ctx_t ctx)
    Writes a description of the element x to the stream out, or prints it to stdout, or sets s to a pointer
    to a heap-allocated string of the description (the user must free the string with flint_free). The
    println version prints a trailing newline.

int gr_set_str(gr_ptr res, const char *x, gr_ctx_t ctx)
    Sets res to the string description in x.

int gr_write_n(gr_stream_t out, gr_srcptr x, slong n, gr_ctx_t ctx)
int gr_get_str_n(char **s, gr_srcptr x, slong n, gr_ctx_t ctx)
    String conversion where real and complex numbers may be rounded to n digits.
```

Assignment and conversions

```
int gr_set(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
    Sets res to a copy of the element x.

int gr_set_other(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t ctx)
    Sets res to the element x of the structure x_ctx which may be different from ctx. This returns the
    GR_DOMAIN flag if x is not an element of ctx or cannot be converted unambiguously to ctx. The
    GR_UNABLE flag is returned if the conversion is not implemented.

int gr_set_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_set_si(gr_ptr res, slong x, gr_ctx_t ctx)
int gr_set_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
```

int **gr_set_fmpq**(*gr_ptr* res, const *fmpq_t* x, *gr_ctx_t* ctx)

int **gr_set_d**(*gr_ptr* res, double x, *gr_ctx_t* ctx)

Sets *res* to the value *x*. If no reasonable conversion to the domain *ctx* is possible, returns GR_DOMAIN.

int **gr_get_si**(*slong* *res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_get_ui**(*ulong* *res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_get_fmpz**(*fmpz_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_get_fmpq**(*fmpq_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_get_d**(double *res, *gr_srcptr* x, *gr_ctx_t* ctx)

Sets *res* to the value *x*. This returns the GR_DOMAIN flag if *x* cannot be converted to the target type. For floating-point output types, the output may be rounded.

int **gr_set_fmpz_2exp_fmpz**(*gr_ptr* res, const *fmpz_t* a, const *fmpz_t* b, *gr_ctx_t* ctx)

int **gr_get_fmpz_2exp_fmpz**(*fmpz_t* res1, *fmpz_t* res2, *gr_srcptr* x, *gr_ctx_t* ctx)

Set or retrieve a dyadic number $a \cdot 2^b$.

int **gr_get_d_2exp_si**(double *res, *slong* *exp, *gr_srcptr* x, *gr_ctx_t* ctx)

Returns $m2^e \approx x$ with $0.5 \leq |m| < 1$ if *x* is nonzero.

int **gr_set_fmpz_10exp_fmpz**(*gr_ptr* res, const *fmpz_t* a, const *fmpz_t* b, *gr_ctx_t* ctx)

Set to a decimal number $a \cdot 10^b$.

int **gr_get_fexpr**(*fexpr_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_get_fexpr_serialize**(*fexpr_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

Sets *res* to a symbolic expression representing *x*. The *serialize* version may generate a representation of the internal representation which is not intended to be human-readable.

int **gr_set_fexpr**(*gr_ptr* res, *fexpr_vec_t* inputs, *gr_vec_t* outputs, const *fexpr_t* x, *gr_ctx_t* ctx)

Sets *res* to the evaluation of the expression *x* in the given ring or structure. The user must provide vectors *inputs* and *outputs* which may be empty initially and which may be used as scratch space during evaluation. Non-empty vectors may be given to map symbols to predefined values.

Special values

int **gr_zero**(*gr_ptr* res, *gr_ctx_t* ctx)

int **gr_one**(*gr_ptr* res, *gr_ctx_t* ctx)

int **gr_neg_one**(*gr_ptr* res, *gr_ctx_t* ctx)

Sets *res* to the ring element 0, 1 or -1.

int **gr_gen**(*gr_ptr* res, *gr_ctx_t* ctx)

Sets *res* to a generator of this domain. The meaning of “generator” depends on the domain.

int **gr_gens**(*gr_vec_t* res, *gr_ctx_t* ctx)

int **gr_gens_recursive**(*gr_vec_t* res, *gr_ctx_t* ctx)

Sets *res* to a vector containing the generators of this domain where this makes sense, for example in a multivariate polynomial ring. The *recursive* version also includes any generators of the base ring, and of any recursive base rings.

Basic properties

`truth_t gr_is_zero(gr_srcptr x, gr_ctx_t ctx)`

`truth_t gr_is_one(gr_srcptr x, gr_ctx_t ctx)`

`truth_t gr_is_neg_one(gr_srcptr x, gr_ctx_t ctx)`

Returns whether x is equal to the ring element 0, 1 or -1, respectively.

`truth_t gr_equal(gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`

Returns whether the elements x and y are equal.

`truth_t gr_is_integer(gr_srcptr x, gr_ctx_t ctx)`

Returns whether x represents an integer.

`truth_t gr_is_rational(gr_srcptr x, gr_ctx_t ctx)`

Returns whether x represents a rational number.

Arithmetic

User-defined rings should supply `neg`, `add`, `sub` and `mul` methods; the variants with other operand types have generic fallbacks that may be overridden for performance. The `fmpq` versions may return `GR_DOMAIN` if the denominator is not invertible. The *other* versions accept operands belonging to a different domain, attempting to perform a coercion into the target domain.

`int gr_neg(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`

Sets res to $-x$.

`int gr_add(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`

`int gr_add_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)`

`int gr_add_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)`

`int gr_add_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)`

`int gr_add_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)`

`int gr_add_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)`

`int gr_other_add(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)`

Sets res to $x + y$.

`int gr_sub(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`

`int gr_sub_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)`

`int gr_sub_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)`

`int gr_sub_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)`

`int gr_sub_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)`

`int gr_sub_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)`

`int gr_other_sub(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)`

Sets res to $x - y$.

`int gr_mul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`

`int gr_mul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)`

`int gr_mul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)`

`int gr_mul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)`

`int gr_mul_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)`

`int gr_mul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)`

`int gr_other_mul(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)`

Sets res to $x \cdot y$.

`int gr_addmul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`


```

int gr_addmul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_addmul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_addmul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_addmul_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)
int gr_addmul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
    Sets res to  $res + x \cdot y$ . Rings may override the default implementation to perform this operation in
    one step without allocating a temporary variable, without intermediate rounding, etc.

int gr_submul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_submul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_submul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_submul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_submul_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)
int gr_submul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
    Sets res to  $res - x \cdot y$ . Rings may override the default implementation to perform this operation in
    one step without allocating a temporary variable, without intermediate rounding, etc.

int gr_mul_two(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
    Sets res to  $2x$ . The default implementation adds  $x$  to itself.

int gr_sqr(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
    Sets res to  $x^2$ . The default implementation multiplies  $x$  with itself.

int gr_mul_2exp_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_mul_2exp_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
    Sets res to  $x \cdot 2^y$ . This may perform  $x \cdot 2^{-y}$  when  $y$  is negative, allowing exact division by powers
    of two even if  $2^y$  is not representable.

Iterated arithmetic operations are best performed using vector functions. See in particular
_gr_vec_dot() and _gr_vec_dot_rev().

```

Division

The default implementations of the following methods check for divisors 0, 1, -1 and otherwise return GR_UNABLE. Particular rings should override the methods when an inversion or division algorithm is available.

```

truth_t gr_is_invertible(gr_srcptr x, gr_ctx_t ctx)
    Returns whether  $x$  has a multiplicative inverse in the present ring, i.e. whether  $x$  is a unit.

int gr_inv(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
    Sets res to the multiplicative inverse of  $x$  in the present ring, if such an element exists. Returns
    the flag GR_DOMAIN if  $x$  is not invertible, or GR_UNABLE if the implementation is unable to perform
    the computation.

int gr_div(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_div_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_div_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_div_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_div_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)
int gr_div_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_other_div(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t y_ctx)
    Sets res to the quotient  $x/y$ . In a field, this returns GR_DOMAIN if  $y$  is zero; in an integral domain, it
    returns GR_DOMAIN if  $y$  is zero or if the quotient does not exist. In a non-integral domain, we consider
    a quotient to exist only if it is unique, and otherwise return GR_DOMAIN; see gr_div_nonunique()
    for a different behavior.

    Returns the flag GR_UNABLE if the implementation is unable to perform the computation.

```

int **gr_div_nonunique**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

Sets *res* to an arbitrary solution q of the equation $x = qy$. Returns the flag `GR_DOMAIN` if no such solution exists. Returns the flag `GR_UNABLE` if the implementation is unable to perform the computation. This method allows dividing x/y in some cases where *gr_div()* fails:

- $0/0$ has solutions (for example, 0) in any ring.
- It allows solving division problems in nonintegral domains. For example, it allows assigning a value to $6/2$ in $R = \mathbb{Z}/10\mathbb{Z}$ even though 2^{-1} does not exist in R . In this case, both 3 and 8 are possible solutions, and which one is chosen is unpredictable.

truth_t **gr_divides**(*gr_srcptr* d, *gr_srcptr* x, *gr_ctx_t* ctx)

Returns whether $d \mid x$; that is, whether there is an element q such that $x = dq$. Note that this corresponds to divisibility in the sense of *gr_div_nonunique()*, which is weaker than that of *gr_div()*. For example, $0 \mid 0$ is true even in rings where $0/0$ is undefined.

int **gr_divexact**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

int **gr_divexact_ui**(*gr_ptr* res, *gr_srcptr* x, *ulong* y, *gr_ctx_t* ctx)

int **gr_divexact_si**(*gr_ptr* res, *gr_srcptr* x, *slong* y, *gr_ctx_t* ctx)

int **gr_divexact_fmpz**(*gr_ptr* res, *gr_srcptr* x, const *fmpz_t* y, *gr_ctx_t* ctx)

int **gr_divexact_other**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* y_ctx, *gr_ctx_t* ctx)

int **gr_other_divexact**(*gr_ptr* res, *gr_srcptr* x, *gr_ctx_t* x_ctx, *gr_srcptr* y, *gr_ctx_t* ctx)

Sets *res* to the quotient x/y , assuming that this quotient is exact in the present ring. Rings may optimize this operation by not verifying that the division is possible. If the division is not actually exact, the implementation may set *res* to a nonsense value and still return the `GR_SUCCESS` flag.

int **gr_euclidean_div**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

int **gr_euclidean_rem**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

int **gr_euclidean_divrem**(*gr_ptr* res1, *gr_ptr* res2, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

In a Euclidean ring, these functions perform some version of Euclidean division with remainder, where the choice of quotient is implementation-defined. For example, it is standard to use the round-to-floor quotient in $\mathbb{Z}$ and a round-to-nearest quotient in $\mathbb{Z}[i]$. In non-Euclidean rings, these functions may implement some generalization of Euclidean division with remainder.

Powering

int **gr_pow**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

int **gr_pow_ui**(*gr_ptr* res, *gr_srcptr* x, *ulong* y, *gr_ctx_t* ctx)

int **gr_pow_si**(*gr_ptr* res, *gr_srcptr* x, *slong* y, *gr_ctx_t* ctx)

int **gr_pow_fmpz**(*gr_ptr* res, *gr_srcptr* x, const *fmpz_t* y, *gr_ctx_t* ctx)

int **gr_pow_fmpq**(*gr_ptr* res, *gr_srcptr* x, const *fmpq_t* y, *gr_ctx_t* ctx)

int **gr_pow_other**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* y_ctx, *gr_ctx_t* ctx)

int **gr_other_pow**(*gr_ptr* res, *gr_srcptr* x, *gr_ctx_t* x_ctx, *gr_srcptr* y, *gr_ctx_t* ctx)

Sets *res* to the power x^y , the interpretation of which depends on the ring when $y \notin \mathbb{Z}$. Returns the flag `GR_DOMAIN` if this power cannot be assigned a meaningful value in the present ring, or `GR_UNABLE` if the implementation is unable to perform the computation.

For subrings of $\mathbb{C}$, it is implied that the principal power $x^y = \exp(y \log(x))$ is computed for $x \neq 0$.

Default implementations of the powering methods support raising elements to integer powers using a generic implementation of exponentiation by squaring. Particular rings should override these methods with faster versions or to support more general notions of exponentiation when possible.

Derivative

int **gr_derivative_gen**(*gr_ptr* res, *gr_srcptr* x, *slong* var, *gr_ctx_t* ctx)

Sets *res* to the derivative of *x* with respect to the variable of index *var*. The variables here are the generators of *ctx*.

Square roots

The default implementations of the following methods check for the elements 0 and 1 and otherwise return `GR_UNABLE`. Particular rings should override the methods when a square root algorithm is available.

In subrings of $\mathbb{C}$, it is implied that the principal square root is computed; in other cases (e.g. in finite fields), the choice of root is implementation-dependent.

truth_t **gr_is_square**(*gr_srcptr* x, *gr_ctx_t* ctx)

Returns whether *x* is a perfect square in the present ring.

int **gr_sqrt**(*gr_ptr* res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_rsqrt**(*gr_ptr* res, *gr_srcptr* x, *gr_ctx_t* ctx)

Sets *res* to a square root of *x* (respectively reciprocal square root) in the present ring, if such an element exists. Returns the flag `GR_DOMAIN` if *x* is not a perfect square (also for zero, when computing the reciprocal square root), or `GR_UNABLE` if the implementation is unable to perform the computation.

Greatest common divisors

int **gr_gcd**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

Sets *res* to a greatest common divisor (GCD) of *x* and *y*. Since the GCD is unique only up to multiplication by a unit, an implementation-defined representative is chosen.

int **gr_lcm**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

Sets *res* to a least common multiple (LCM) of *x* and *y*. Since the LCM is unique only up to multiplication by a unit, an implementation-defined representative is chosen.

Factorization

int **gr_canonical_associate**(*gr_ptr* res, *gr_ptr* u, *gr_srcptr* x, *gr_ctx_t* ctx)

Given a commutative ring element *x*, sets *res* to a canonical form *ux* with respect to multiplication by units, also writing the normalizing unit *u* as a second output. If *x* = 0, sets *res* to 0 and *u* to 1.

int **gr_factor**(*gr_ptr* c, *gr_vec_t* factors, *gr_vec_t* exponents, *gr_srcptr* x, int flags, *gr_ctx_t* ctx)

Given $x \in R$, computes a factorization

$$x = cf_1^{e_1} \dots f_n^{e_n}$$

where f_k will be irreducible or prime (depending on *R*).

The prefactor *c* stores a unit, sign, or coefficient, e.g., the sign -1 , 0 or $+1$ in $\mathbb{Z}$, or a sign multiplied by the coefficient content in $\mathbb{Z}[x]$. Note that this function outputs *c* as an element of the same ring as the input: for example, in $\mathbb{Z}[x]$, *c* will be a constant polynomial rather than an element of the coefficient ring. The exponents e_k are output as a vector of `fmpz` elements.

The factors f_k are guaranteed to be distinct, but they are not guaranteed to be sorted in any particular order.

Fractions

int **gr_numerator**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_denominator**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

Return a numerator p and denominator q such that $x = p/q$. For typical fraction fields, the denominator will be minimal and canonical. However, some rings may return an arbitrary denominator as long as the numerator matches. The default implementations simply return $p = x$ and $q = 1$.

Integer and complex parts

int **gr_floor**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_ceil**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_trunc**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_nint**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

In the real and complex numbers, sets *res* to the integer closest to x , respectively rounding towards minus infinity, plus infinity, zero, or the nearest integer (with tie-to-even).

int **gr_abs**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

Sets *res* to the absolute value of x , which maybe defined both in complex rings and in any ordered ring.

int **gr_i**(gr_ptr res, gr_ctx_t ctx)

Sets *res* to the imaginary unit.

int **gr_conj**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_re**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_im**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_sgn**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_csgn**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int **gr_arg**(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

These methods may return the flag `GR_DOMAIN` (or `GR_UNABLE`) when the ring is not a subring of the real or complex numbers.

Infinities and extended values

int **gr_pos_inf**(gr_ptr res, gr_ctx_t ctx)

int **gr_neg_inf**(gr_ptr res, gr_ctx_t ctx)

int **gr_uinf**(gr_ptr res, gr_ctx_t ctx)

int **gr_undefined**(gr_ptr res, gr_ctx_t ctx)

int **gr_unknown**(gr_ptr res, gr_ctx_t ctx)

Sets *res* to the signed positive infinity $+\infty$, signed negative infinity $-\infty$, unsigned infinity $\tilde{\infty}$, *Undefined*, or *Unknown*, respectively.

Ordering methods

int **gr_cmp**(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)

int **gr_cmp_other**(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)

Sets *res* to -1, 0 or 1 according to whether x is less than, equal or greater than y . This may return `GR_DOMAIN` if the ring is not an ordered ring.

int **gr_cmpabs**(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)

int **gr_cmpabs_other**(int *res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* y_ctx, *gr_ctx_t* ctx)

Sets *res* to -1, 0 or 1 according to whether the absolute value of *x* is less than, equal or greater than the absolute value of *y*. This may return GR_DOMAIN if the ring is not an ordered ring.

truth_t **gr_le**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_lt**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_ge**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_gt**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_abs_le**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_abs_lt**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_abs_ge**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

truth_t **gr_abs_gt**(*gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

Wrappers of **gr_cmp** and **gr_cmpabs** returning truth values for the comparison operations $\leq$, $<$, $\geq$, $>$.

int **gr_min**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

int **gr_max**(*gr_ptr* res, *gr_srcptr* x, *gr_srcptr* y, *gr_ctx_t* ctx)

Minimum and maximum value.

Enclosure and interval methods

int **gr_set_interval_mid_rad**(*gr_ptr* res, *gr_srcptr* m, *gr_srcptr* r, *gr_ctx_t* ctx)

In ball representations of the real numbers, sets *res* to the interval $m \pm r$.

In vector spaces over the real numbers represented using balls, intervals are handled independently for the generators; for example, in the complex numbers, $a + bi \pm (0.1 + 0.2i)$ is equivalent to $(a \pm 0.1) + (b \pm 0.2)i$.

Finite field methods

int **gr_ctx_fq_prime**(*fmpz_t* p, *gr_ctx_t* ctx)

int **gr_ctx_fq_degree**(*slong* *deg, *gr_ctx_t* ctx)

int **gr_ctx_fq_order**(*fmpz_t* q, *gr_ctx_t* ctx)

int **gr_fq_frobenius**(*gr_ptr* res, *gr_srcptr* x, *slong* e, *gr_ctx_t* ctx)

int **gr_fq_multiplicative_order**(*fmpz_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_fq_norm**(*fmpz_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_fq_trace**(*fmpz_t* res, *gr_srcptr* x, *gr_ctx_t* ctx)

truth_t **gr_fq_is_primitive**(*gr_srcptr* x, *gr_ctx_t* ctx)

int **gr_fq_pth_root**(*gr_ptr* res, *gr_srcptr* x, *gr_ctx_t* ctx)

3.2 gr.h (continued) – implementing rings

Defining a ring requires putting appropriate data into a `gr_ctx_t` parent object, most importantly the method table and the size of elements.

3.2.1 Example

This is an extract from the `fmpr` wrapper in `gr/fmpr.c`:

```
/* Some methods */
...
int
_gr_fmpz_add(fmpz_t res, const fmpz_t x, const fmpz_t y, const gr_ctx_t ctx)
{
    fmpz_add(res, x, y);
    return GR_SUCCESS;
}
...

/* The method table */

int _fmpr_methods_initialized = 0;

gr_static_method_table _fmpr_methods;

gr_method_tab_input _fmpr_methods_input[] =
{
    {GR_METHOD_CTX_IS_RING,      (gr_funcptr) gr_generic_ctx_predicate_true},
    ...
    {GR_METHOD_INIT,            (gr_funcptr) _gr_fmpz_init},
    {GR_METHOD_CLEAR,           (gr_funcptr) _gr_fmpz_clear},
    ...
    {GR_METHOD_ADD_FMPZ,        (gr_funcptr) _gr_fmpz_add},
    ...
    {0,                          (gr_funcptr) NULL},
};

/* Context object initializer */

void
gr_ctx_init_fmpz(gr_ctx_t ctx)
{
    ctx->which_ring = GR_CTX_FMPZ;
    ctx->sizeof_elem = sizeof(fmpz);
    ctx->size_limit = WORD_MAX;

    ctx->methods = _fmpr_methods;

    if (!_fmpr_methods_initialized)
    {
        gr_method_tab_init(_fmpr_methods, _fmpr_methods_input);
        _fmpr_methods_initialized = 1;
    }
}
```

Note that the method table only has to be constructed once, allowing new context objects for the same domain to be initialized cheaply.

3.2.2 Method table

type **gr_funcptr**

Typedef for a pointer to a function with signature `int func(void)`, used to represent method table entries.

type **gr_method**

Enumeration type for indexing method tables. Enum values named `GR_METHOD_INIT`, `GR_METHOD_ADD_UI`, etc. correspond to methods `gr_init`, `gr_add_ui`, etc. The number of methods is given by `GR_METHOD_TAB_SIZE`, which can be used to declare static method tables.

type **gr_static_method_table**

Typedef for an array of length `GR_METHOD_TAB_SIZE` with *gr_funcptr* entries.

type **gr_method_tab_input**

Typedef representing a (index, function pointer) pair.

void **gr_method_tab_init**(*gr_funcptr* *methods, *gr_method_tab_input* *tab)

Initializes the method table *methods*. This first inserts default and generic methods in all slots, and then overwrites with the specialized methods listed in *tab*.

3.2.3 Placeholder and trivial methods

int **gr_not_implemented**(void)

This function does nothing and returns `GR_UNABLE`. It is used as a generic fallback method when no implementation is available.

int **gr_not_in_domain**(void)

This function does nothing and returns `GR_DOMAIN`. It can be used for an operation that never makes sense in the present domain, e.g., for the constant π in the rational numbers.

truth_t **gr_generic_ctx_predicate**(*gr_ctx_t* ctx)

Does nothing and returns `T_UNKNOWN`, used as a generic fallback for predicate methods.

truth_t **gr_generic_ctx_predicate_true**(*gr_ctx_t* ctx)

A predicate that does nothing and returns `T_TRUE`.

truth_t **gr_generic_ctx_predicate_false**(*gr_ctx_t* ctx)

A predicate that does nothing and returns `T_FALSE`.

3.2.4 Required methods

A context object must at minimum define the following methods for a ring:

- `init`
- `clear`
- `swap`
- `randtest`
- `write`
- `zero`
- `one`
- `equal`
- `set`
- `set_si`

- `set_ui`
- `set_fmpz`
- `neg`
- `add`
- `sub`
- `mul`

Other methods have generic defaults which may be overridden for performance or completeness.

Implementing context predicates (`ctx_is_integral_domain`, `ctx_is_field`, etc.) is strongly recommended so that the most appropriate algorithms can be used in generic implementations.

Rings with cheap operations on single elements should also provide non-generic versions of performance-critical vector operations to minimize overhead. The most important vector operations include:

- `vec_init`
- `vec_clear`
- `vec_swap`
- `vec_zero`
- `vec_neg`
- `vec_add`
- `vec_sub`
- `vec_mul_scalar_ui/si`
- `vec_addmul_scalar_ui/si`
- `vec_dot`
- `vec_dot_rev`

Dot products, for example, are the main building block for classical polynomial multiplication and matrix multiplication. The methods

- `poly_mullov`
- `matrix_mul`

should be overridden for rings where faster-than-classical polynomial and matrix multiplication is possible. Other higher-complexity generic algorithms will try to reduce to polynomial and matrix multiplication automatically, but may in turn need to be overridden to select accurate cutoffs between different algorithms.

3.2.5 Testing rings

void **gr_test_ring**(*gr_ctx_t* R, *slong* iters, int test_flags)

Test correctness of the ring *R*. This calls test functions for various methods, each being repeated up to *iters* times.

3.3 gr.h (continued) – builtin domains and types

3.3.1 Coercions

int `gr_ctx_cmp_coercion`(*gr_ctx_t* ctx1, *gr_ctx_t* ctx2)

Returns 1 if coercing elements into *ctx1* is more meaningful, and returns -1 otherwise.

3.3.2 Domain properties

truth_t `gr_ctx_is_finite`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_multiplicative_group`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_ring`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_commutative_ring`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_integral_domain`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_unique_factorization_domain`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_field`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_algebraically_closed`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_rational_vector_space`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_real_vector_space`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_complex_vector_space`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_finite_characteristic`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_ordered_ring`(*gr_ctx_t* ctx)

truth_t `gr_ctx_is_zero_ring`(*gr_ctx_t* ctx)

Returns whether the structure satisfies the respective mathematical property. The result can be T_UNKNOWN.

truth_t `gr_ctx_is_approx_commutative_ring`(*gr_ctx_t* ctx)

Returns whether the structure $\tilde{R}$ implemented by *ctx* is assumed to represent an approximate version of a structure R with the respective mathematical property. The usual axioms need not hold exactly as long as they hold within a small perturbation, i.e. an axiom of the type $a = b$ in R is replaced by an axiom of the type $|\tilde{a} - \tilde{b}| < \varepsilon$ in $\tilde{R}$. For example, polynomials with floating-point coefficients are considered an approximate commutative ring, but two by two matrices with floating-point coefficients are not.

The exact structure is trivially considered an approximate version of itself; for example, every commutative ring is considered to be an approximate commutative ring.

truth_t `gr_ctx_is_exact`(*gr_ctx_t* ctx)

Returns whether the representation of elements is always exact.

truth_t `gr_ctx_is_canonical`(*gr_ctx_t* ctx)

Returns whether the representation of elements is always canonical.

truth_t `gr_ctx_has_real_prec`(*gr_ctx_t* ctx)

Returns whether *ctx* or a base field thereof represents real or complex numbers using finite-precision approximations. This returns T_TRUE both for floating-point approximate fields and for rigorous fields based on ball or interval arithmetic.

int `gr_ctx_set_real_prec`(*gr_ctx_t* ctx, *slong* prec)

int `gr_ctx_get_real_prec`(*slong* *prec, *gr_ctx_t* ctx)

Sets or retrieves the floating-point precision in bits.

3.3.3 Debugging

void **gr_ctx_init_debug**(*gr_ctx_t* ctx, *gr_ctx_t* elem_ctx, int flags, double unable_probability)

Initialize *ctx* to a wrapper around *elem_ctx* for debugging. This will execute supported methods (currently only a handful of methods are supported, e.g. **gr_add**) as if one is computing directly over *elem_ctx*, but with added debugging features.

If *unable_probability* is positive, some methods will randomly return **GR_UNABLE** and predicates will randomly return **T_UNKNOWN**.

The following flags are supported:

- **GR_DEBUG_WRAP** - wrap elements in a box with a pointer. This allows performing extra checks such as whether a variable is being cleared twice or whether the wrong context object is passed as input. It can also help catch some bugs that are not revealed when entries are stored shallowly.
- **GR_DEBUG_VERBOSE** - print debugging information for each operation.
- **GR_DEBUG_TIMING** - print elapsed time for each operation.

3.3.4 Groups

void **gr_ctx_init_perm**(*gr_ctx_t* ctx, *ulong* n)

Initializes *ctx* to the symmetric group S_n representing permutations of $[0, 1, \dots, n - 1]$. Elements are currently represented as pointers (the representation may change in the future).

void **gr_ctx_init_psl2z**(*gr_ctx_t* ctx)

Initializes *ctx* to the modular group $\mathrm{PSL}(2, \mathbb{Z})$ with elements of type *psl2z_t*.

int **gr_ctx_init_dirichlet_group**(*gr_ctx_t* ctx, *ulong* q)

Initializes *ctx* to the Dirichlet group G_q with elements of type *dirichlet_char_t*. Fails and returns **GR_DOMAIN** if *q* is zero. Fails and returns **GR_UNABLE** if *q* has a prime factor larger than 10^{16} , which is currently unsupported by the implementation.

3.3.5 Basic rings and fields

void **gr_ctx_init_random**(*gr_ctx_t* ctx, *flint_rand_t* state)

void **gr_ctx_init_random_commutative_ring**(*gr_ctx_t* ctx, *flint_rand_t* state)

void **gr_ctx_init_random_field**(*gr_ctx_t* ctx, *flint_rand_t* state)

Initializes *ctx* to a random ring. This will currently only generate base rings and composite rings over certain simple base rings.

void **gr_ctx_init_fmpz**(*gr_ctx_t* ctx)

Initializes *ctx* to the ring of integers $\mathbb{Z}$ with elements of type *fmpz*.

void **gr_ctx_init_fmpq**(*gr_ctx_t* ctx)

Initializes *ctx* to the field of rational numbers $\mathbb{Q}$ with elements of type *fmpq*.

void **gr_ctx_init_fmpz_i**(*gr_ctx_t* ctx)

Initializes *ctx* to the ring of Gaussian integers $\mathbb{Z}[i]$ with elements of type *fmpz_i_t*.

3.3.6 Residue rings and finite fields

int `gr_ctx_set_is_field`(*gr_ctx_t* ctx, *truth_t* is_field)

Set whether the given ring is actually a field. For example, in the case of $\mathbb{Z}/n\mathbb{Z}$, this sets whether the modulus is prime. This can speed up some computations and enable some functions to complete that otherwise would return `GR_UNABLE`.

See:

- `gr_ctx_init_nmod()` `gr_ctx_init_nmod8()` `gr_ctx_init_nmod32()`
`gr_ctx_init_nmod_redc()` `gr_ctx_init_nmod_redc_fast()`

void `gr_ctx_init_fmpz_mod`(*gr_ctx_t* ctx, const *fmpz_t* n)

Initializes *ctx* to the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo *n* where elements have type *fmpz*. The modulus must be positive.

- `gr_ctx_init_mpn_mod()`

Initializes *ctx* to the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo *n* where elements are flat limb arrays with the same number of limbs as *n*.

void `gr_ctx_init_fq`(*gr_ctx_t* ctx, const *fmpz_t* p, *slong* d, const char *var)

void `gr_ctx_init_fq_nmod`(*gr_ctx_t* ctx, *ulong* p, *slong* d, const char *var)

void `gr_ctx_init_fq_zech`(*gr_ctx_t* ctx, *ulong* p, *slong* d, const char *var)

Initializes *ctx* to the finite field $\mathbb{F}_q$ where $q = p^d$. It is assumed (not checked) that *p* is prime. The variable name *var* can be NULL to use a default.

The corresponding element types are *fq_t*, *fq_nmod_t*, *fq_zech_t*. The *fq_zech* context requires $q < 2^{64}$ (and in practice a much smaller value than this).

3.3.7 Number fields and algebraic numbers

void `gr_ctx_init_nf`(*gr_ctx_t* ctx, const *fmpz_poly_t* poly)

void `gr_ctx_init_nf_fmpz_poly`(*gr_ctx_t* ctx, const *fmpz_poly_t* poly)

Initializes *ctx* to the number field with defining polynomial *poly* which *must* be irreducible (this is not checked). The elements have type *nf_elem_t*.

void `gr_ctx_init_real_qqbar`(*gr_ctx_t* ctx)

void `gr_ctx_init_complex_qqbar`(*gr_ctx_t* ctx)

Initializes *ctx* to the field of real or complex algebraic numbers with elements of type *qqbar_t*.

void `_gr_ctx_qqbar_set_limits`(*gr_ctx_t* ctx, *slong* deg_limit, *slong* bits_limit)

Limit degrees of intermediate operands of a *qqbar* context to *deg_limit* and their bit sizes to *bits_limit* (approximately). The limits refer to the sizes of resultants prior to factorization (see `qqbar_binop_within_limits()`), so for example adding two degree-100 algebraic numbers requires a degree limit of at least 10000. Warning: currently not all methods respect these limits.

3.3.8 Real and complex numbers

void `gr_ctx_init_real_arb`(*gr_ctx_t* ctx, *slong* prec)

void `gr_ctx_init_complex_acb`(*gr_ctx_t* ctx, *slong* prec)

Initializes *ctx* to the field of real or complex numbers represented by elements of type *arb_t* and *acb_t*.

void `gr_ctx_arb_set_prec`(*gr_ctx_t* ctx, *slong* prec)

slong **gr_ctx_arb_get_prec**(*gr_ctx_t* ctx)

Sets or retrieves the bit precision of *ctx* which must be an Arb context (this is currently not checked).

void **gr_ctx_init_real_ca**(*gr_ctx_t* ctx)

void **gr_ctx_init_complex_ca**(*gr_ctx_t* ctx)

void **gr_ctx_init_real_algebraic_ca**(*gr_ctx_t* ctx)

void **gr_ctx_init_complex_algebraic_ca**(*gr_ctx_t* ctx)

Initializes *ctx* to the field of real, complex, real algebraic or complex algebraic numbers represented by elements of type *ca_t*.

void **gr_ctx_ca_set_option**(*gr_ctx_t* ctx, *slong* option, *slong* value)

slong **gr_ctx_ca_get_option**(*gr_ctx_t* ctx, *slong* option)

Sets or retrieves options of a Calcium context object.

void **gr_ctx_init_gr_complex**(*gr_ctx_t* ctx, *gr_ctx_t* real_ctx)

Initializes *ctx* to a generic implementation of the complex algebra $R[i]$ where R is represented by *real_ctx*. Elements $a + bi$ are represented as pairs (a, b) of elements of R stored contiguously in memory.

Typically R will be an implementation of the real numbers or a subring of the real numbers in which case this creates the complex numbers or a subring of the complex numbers. This construction also makes sense e.g. over real vector spaces and even general rings (where i will just be a formal element satisfying $i^2 = -1$), but in that case operations beyond basic arithmetic (e.g. absolute value) may not make sense.

3.3.9 Extended number sets

void **gr_ctx_init_complex_extended_ca**(*gr_ctx_t* ctx)

Like *gr_ctx_init_complex_ca()* but allows special values (infinities, undefined).

3.3.10 Floating-point arithmetic

Although domains of floating-point numbers approximate real and complex fields, they are not rings or fields. Floating-point arithmetic can be used in many places where a ring or field is normally assumed, but predicates like “is field” return false.

- Equality compares equality of floating-point numbers, with the special rule that NaN is not equal to itself.
- In general, the following implementations do not currently guarantee correct rounding except for atomic arithmetic operations (add, sub, mul, div, sqrt) on real floating-point numbers.

void **gr_ctx_init_real_float_arf**(*gr_ctx_t* ctx, *slong* prec)

Initializes *ctx* to the floating-point arithmetic with elements of type *arf_t* and a default precision of *prec* bits.

void **gr_ctx_init_complex_float_acf**(*gr_ctx_t* ctx, *slong* prec)

Initializes *ctx* to the complex floating-point arithmetic with elements of type *acf_t* and a default precision of *prec* bits.

3.3.11 Vectors

void **gr_ctx_init_vector_gr_vec**(*gr_ctx_t* ctx, *gr_ctx_t* base_type)

Initializes *ctx* to the domain of all vectors (of any length) over the given *base_type*. Elements have type *gr_vec_struct*.

void **gr_ctx_init_vector_space_gr_vec**(*gr_ctx_t* ctx, *gr_ctx_t* base_type, *slong* n)

Initializes *ctx* to the space of all vectors of length *n* over the given *base_type*. Elements have type *gr_vec_struct*.

3.3.12 Matrices

void **gr_ctx_init_matrix_domain**(*gr_ctx_t* ctx, *gr_ctx_t* base_ring)

Initializes *ctx* to the domain of all matrices (of any shape) over the given *base_ring*. Elements have type *gr_mat_struct*.

void **gr_ctx_init_matrix_space**(*gr_ctx_t* ctx, *gr_ctx_t* base_ring, *slong* n, *slong* m)

Initializes *ctx* to the space of matrices over *base_ring* with *n* rows and *m* columns. Elements have type *gr_mat_struct*.

void **gr_ctx_init_matrix_ring**(*gr_ctx_t* ctx, *gr_ctx_t* base_ring, *slong* n)

Initializes *ctx* to the ring of matrices over *base_ring* with *n* rows columns. Elements have type *gr_mat_struct*.

3.3.13 Polynomial rings

void **gr_ctx_init_fmpz_poly**(*gr_ctx_t* ctx)

Initializes *ctx* to a ring of integer polynomials of type *fmpz_poly_struct*.

void **gr_ctx_init_fmpq_poly**(*gr_ctx_t* ctx)

Initializes *ctx* to a ring of rational polynomials of type *fmpq_poly_struct*.

void **gr_ctx_init_gr_poly**(*gr_ctx_t* ctx, *gr_ctx_t* base_ring)

Initializes *ctx* to a ring of densely represented univariate polynomials over the given *base_ring*. Elements have type *gr_poly_struct*.

void **gr_ctx_init_fmpz_mpoly**(*gr_ctx_t* ctx, *slong* nvars, const *ordering_t* ord)

Initializes *ctx* to a ring of sparsely represented multivariate polynomials in *nvars* variables over the integers, with monomial ordering *ord*. Elements have type *fmpz_mpoly_struct*.

void **gr_ctx_init_gr_mpoly**(*gr_ctx_t* ctx, *gr_ctx_t* base_ring, *slong* nvars, const *ordering_t* ord)

Initializes *ctx* to a ring of sparsely represented multivariate polynomials in *nvars* variables over the given *base_ring*, with monomial ordering *ord*. Elements have type *gr_mpoly_struct*.

3.3.14 Power series

See *gr_series_ctx_init()* and *gr_series_mod_ctx_init()* in *gr_series.h* – formal power series over generic rings.

3.3.15 Fraction fields

void **gr_ctx_init_gr_fraction**(*gr_ctx_t* ctx, *gr_ctx_t* domain, int flags)

Initializes *ctx* to a generic implementation of the fraction field over the given integral domain *domain*. Fractions are represented as pairs of elements of *domain*, stored consecutively in memory as (numerator, denominator). By default, fractions are simplified by removing common content between the numerator and denominator (using *gr_gcd()*) and normalising the denominator by a canonical unit (using *gr_canonical_associate()*). The following optional *flags* can be set:

GR_FRACTION_NO_REDUCTION

Setting this flag disables canonicalisation, allowing one to perform fraction field arithmetic over domains which do not implement GCD or unit canonicalisation. This flag can also improve performance in certain cases when GCDs are more expensive than just allowing coefficients to blow up.

GR_FRACTION_STRONGLY_CANONICAL

Assert that fractions are in strongly canonical form, meaning that $a/b = c/d$ if and only if $a = c$ and $b = d$. Setting this flag allows faster equality testing. This should be valid in UFDs that implement a correct GCD and correct unit canonicalisation, but need not be true over integral domains which are not UFDs. In this future this may be automatic.

This constructor does not verify that *domain* is really an integral domain. The behavior over non-integral domains is undefined, as no attempt is made to detect the product of two nonzero denominators becoming zero. In the future, such checks may be implemented as an optional feature.

void **gr_ctx_init_fmpz_mpoly_q**(*gr_ctx_t* ctx, *slong* nvars, const *ordering_t* ord)

Initializes *ctx* to a ring of sparsely represented multivariate fractions in *nvars* variables over the integers (equivalently, rationals), with monomial ordering *ord*. Elements have type *fmpz_mpoly_q_struct*.

void **gr_ctx_init_fmpz_mod_mpoly_q**(*gr_ctx_t* ctx, *slong* nvars, const *ordering_t* ord, const *fmpz_t* mod)

Initializes *ctx* to a ring of sparsely represented multivariate fractions in *nvars* variables over the $\mathbb{F}_{mod}$ field, with monomial ordering *ord*, and *mod* being a prime number. The user is responsible for verifying that *mod* is a prime number; if *mod* is composite, undefined behaviour may occur. Elements have type *fmpz_mod_mpoly_q_struct*.

3.3.16 Symbolic expressions

void **gr_ctx_init_fexpr**(*gr_ctx_t* ctx)

Initializes *ctx* to handle symbolic expressions. Elements have type *fexpr_struct*.

3.4 `gr_generic.h` – basic algorithms and fallback implementations for generic elements

```

void gr_generic_init(void)
void gr_generic_clear(void)
void gr_generic_swap(void)
void gr_generic_randtest(void)
void gr_generic_write(void)
void gr_generic_zero(void)
void gr_generic_one(void)
void gr_generic_equal(void)
void gr_generic_set(void)
void gr_generic_set_si(void)
void gr_generic_set_ui(void)
void gr_generic_set_fmpz(void)
void gr_generic_neg(void)
void gr_generic_add(void)
void gr_generic_sub(void)
void gr_generic_mul(void)

void gr_generic_ctx_clear(gr_ctx_t ctx)

void gr_generic_set_shallow(gr_ptr res, gr_srcptr x, const gr_ctx_t ctx)

int gr_generic_write_n(gr_stream_t out, gr_srcptr x, slong n, gr_ctx_t ctx)

int gr_generic_randtest_not_zero(gr_ptr x, flint_rand_t state, gr_ctx_t ctx)

int gr_generic_randtest_small(gr_ptr x, flint_rand_t state, gr_ctx_t ctx)

truth_t gr_generic_is_zero(gr_srcptr x, gr_ctx_t ctx)
truth_t gr_generic_is_one(gr_srcptr x, gr_ctx_t ctx)
truth_t gr_generic_is_neg_one(gr_srcptr x, gr_ctx_t ctx)

int gr_generic_neg_one(gr_ptr res, gr_ctx_t ctx)

int gr_generic_set_other(gr_ptr res, gr_srcptr x, gr_ctx_t xctx, gr_ctx_t ctx)
int gr_generic_set_fmpz(gr_ptr res, const fmpz_t y, gr_ctx_t ctx)

```

3.4.1 Generic string parsing

`GR_PARSE_BALANCE_ADDITIONS`

`GR_PARSE_RING_EXPONENTS`

```

int gr_generic_set_str_expr(gr_ptr res, const char *s, int flags, gr_ctx_t ctx)
int gr_generic_set_str(gr_ptr res, const char *s, gr_ctx_t ctx)
int gr_generic_set_str_balance_additions(gr_ptr res, const char *s, gr_ctx_t ctx)
int gr_generic_set_str_ring_exponents(gr_ptr res, const char *s, gr_ctx_t ctx)

```

Parses expression string. Generators returned by `gr_gens_recursive()` are handled automatically. We have the following flags:

- `GR_PARSE_RING_EXPONENTS` - by default, only (nonnegative) integer literals are allowed for exponents. If this flag is set, exponents are parsed as arbitrary subexpressions within the same ring.
- `GR_PARSE_BALANCE_ADDITIONS` - attempt to improve performance for huge sums by reordering additions (useful for polynomials)

3.4.2 Generic arithmetic

```
int gr_generic_add_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_add_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_add_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_add_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_add_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_add(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)

int gr_generic_sub_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_sub_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_sub_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_sub_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_sub_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_sub(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)

int gr_generic_mul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_mul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_mul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_mul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_mul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_mul(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)

int gr_generic_addmul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_addmul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_addmul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_addmul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_addmul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_addmul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)

int gr_generic_submul(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_submul_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_submul_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_submul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_submul_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_submul_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)

int gr_generic_mul_two(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_generic_sqr(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_generic_mul_2exp_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_mul_2exp_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)

int gr_generic_set_fmpz_2exp_fmpz(gr_ptr res, const fmpz_t x, const fmpz_t y, gr_ctx_t ctx)
```



```

int gr_generic_get_fmpz_2exp_fmpz(fmpz_t res1, fmpz_t res2, gr_ptr x, gr_ctx_t ctx)

int gr_generic_inv(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

truth_t gr_generic_is_invertible(gr_srcptr x, gr_ctx_t ctx)

int gr_generic_div_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int gr_generic_div_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_generic_div_si(gr_ptr res, gr_srcptr x, slong y, gr_ctx_t ctx)
int gr_generic_div_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)
int gr_generic_div_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_div(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)

int gr_generic_divexact(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)

int gr_generic_pow_fmpz_sliding(gr_ptr f, gr_srcptr g, const fmpz_t pow, gr_ctx_t ctx)
int gr_generic_pow_ui_sliding(gr_ptr f, gr_srcptr g, ulong pow, gr_ctx_t ctx)
int gr_generic_pow_fmpz_binexp(gr_ptr res, gr_srcptr x, const fmpz_t exp, gr_ctx_t ctx)
int gr_generic_pow_ui_binexp(gr_ptr res, gr_srcptr x, ulong e, gr_ctx_t ctx)

int gr_generic_pow_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t e, gr_ctx_t ctx)
int gr_generic_pow_si(gr_ptr res, gr_srcptr x, slong e, gr_ctx_t ctx)
int gr_generic_pow_ui(gr_ptr res, gr_srcptr x, ulong e, gr_ctx_t ctx)
int gr_generic_pow_fmpq(gr_ptr res, gr_srcptr x, const fmpq_t y, gr_ctx_t ctx)
int gr_generic_pow_other(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_other_pow(gr_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_srcptr y, gr_ctx_t ctx)

int _gr_fmpz_poly_evaluate_horner(gr_ptr res, const fmpz_t *f, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_fmpz_poly_evaluate_horner(gr_ptr res, const fmpz_poly_t f, gr_srcptr x, gr_ctx_t ctx)
int _gr_fmpz_poly_evaluate_rectangular(gr_ptr res, const fmpz_t *f, slong len, gr_srcptr x,
                                       gr_ctx_t ctx)
int gr_fmpz_poly_evaluate_rectangular(gr_ptr res, const fmpz_poly_t f, gr_srcptr x, gr_ctx_t
                                       ctx)

int _gr_fmpz_poly_evaluate(gr_ptr res, const fmpz_t *f, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_fmpz_poly_evaluate(gr_ptr res, const fmpz_poly_t f, gr_srcptr x, gr_ctx_t ctx)
    Sets res to the value of the integer polynomial f evaluated at the argument x.

int gr_fmpz_mpoly_evaluate_iter(gr_ptr res, const fmpz_mpoly_t f, gr_srcptr x, const
                               fmpz_mpoly_ctx_t mctx, gr_ctx_t ctx)
int gr_fmpz_mpoly_evaluate_horner(gr_ptr res, const fmpz_mpoly_t f, gr_srcptr x, const
                                  fmpz_mpoly_ctx_t mctx, gr_ctx_t ctx)
int gr_fmpz_mpoly_evaluate(gr_ptr res, const fmpz_mpoly_t f, gr_srcptr x, const
                           fmpz_mpoly_ctx_t mctx, gr_ctx_t ctx)
    Sets res to value of the multivariate polynomial f (with corresponding context object mctx) evaluated at the vector of arguments in x.

truth_t gr_generic_is_square(gr_srcptr x, gr_ctx_t ctx)
int gr_generic_sqrt(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_generic_rsqr(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
    Currently these methods check for the special values 0 and 1.

int gr_generic_numerator(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

```

```
int gr_generic_denominator(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_generic_cmp(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_cmpabs(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_generic_cmp_other(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
int gr_generic_cmpabs_other(int *res, gr_srcptr x, gr_srcptr y, gr_ctx_t y_ctx, gr_ctx_t ctx)
```

3.4.3 Generic special functions

To do: move to `gr_special`

```
int gr_generic_bernoulli_ui(gr_ptr res, ulong n, gr_ctx_t ctx)
int gr_generic_bernoulli_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_generic_bernoulli_vec(gr_ptr res, slong len, gr_ctx_t ctx)
int gr_generic_eulernum_ui(gr_ptr res, ulong n, gr_ctx_t ctx)
int gr_generic_eulernum_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_generic_eulernum_vec(gr_ptr res, slong len, gr_ctx_t ctx)
int gr_generic_stirling_s1u_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_generic_stirling_s1_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_generic_stirling_s2_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_generic_stirling_s1u_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
int gr_generic_stirling_s1_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
int gr_generic_stirling_s2_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
```

3.4.4 Generic vector methods

To do: move to `gr_vec`

```
void gr_generic_vec_init(gr_ptr vec, slong len, gr_ctx_t ctx)
void gr_generic_vec_clear(gr_ptr vec, slong len, gr_ctx_t ctx)
void gr_generic_vec_swap(gr_ptr vec1, gr_ptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_zero(gr_ptr vec, slong len, gr_ctx_t ctx)
int gr_generic_vec_set(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)
int gr_generic_vec_neg(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)
int gr_generic_vec_normalise(slong *res, gr_srcptr vec, slong len, gr_ctx_t ctx)
slong gr_generic_vec_normalise_weak(gr_srcptr vec, slong len, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_2exp_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t
                                     ctx)
int gr_generic_vec_scalar_addmul(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                ctx)
int gr_generic_vec_scalar_submul(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                ctx)
int gr_generic_vec_scalar_addmul_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_scalar_submul_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
```

```

truth_t gr_generic_vec_equal(gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_is_zero(gr_srcptr vec, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2,
    slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_rev(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr
    vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_ui(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const ulong
    *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_si(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const slong
    *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_dot_fmpz(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const fmpz
    *vec2, slong len, gr_ctx_t ctx)

int gr_generic_vec_set_powers(gr_ptr res, gr_srcptr x, slong len, gr_ctx_t ctx)

int gr_generic_vec_reciprocals(gr_ptr res, slong len, gr_ctx_t ctx)

int gr_generic_vec_add(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int gr_generic_vec_sub(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int gr_generic_vec_mul(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int gr_generic_vec_div(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int gr_generic_vec_divexact(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int gr_generic_vec_pow(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int gr_generic_vec_add_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_generic_vec_sub_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_generic_vec_div_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
    ctx)

int gr_generic_vec_pow_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_generic_vec_add_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_sub_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_div_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t
    ctx)

int gr_generic_vec_pow_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int gr_generic_vec_add_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int gr_generic_vec_sub_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int gr_generic_vec_div_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t
    ctx)

int gr_generic_vec_pow_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int gr_generic_vec_add_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
    gr_ctx_t ctx)
int gr_generic_vec_sub_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
    gr_ctx_t ctx)

```

```
int gr_generic_vec_mul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_div_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                        gr_ctx_t ctx)
int gr_generic_vec_pow_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_add_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_sub_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_div_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                        gr_ctx_t ctx)
int gr_generic_vec_pow_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c,
                                   gr_ctx_t ctx)
int gr_generic_scalar_add_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_sub_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_mul_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_div_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_divexact_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t
                                   ctx)
int gr_generic_scalar_pow_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int gr_generic_vec_add_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong
                             len, gr_ctx_t ctx)
int gr_generic_vec_sub_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong
                             len, gr_ctx_t ctx)
int gr_generic_vec_mul_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong
                             len, gr_ctx_t ctx)
int gr_generic_vec_div_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong
                             len, gr_ctx_t ctx)
int gr_generic_vec_divexact_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3,
                                  slong len, gr_ctx_t ctx)
int gr_generic_vec_pow_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong
                             len, gr_ctx_t ctx)
int gr_generic_other_add_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong
                             len, gr_ctx_t ctx)
int gr_generic_other_sub_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong
                             len, gr_ctx_t ctx)
int gr_generic_other_mul_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong
                             len, gr_ctx_t ctx)
int gr_generic_other_div_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong
                             len, gr_ctx_t ctx)
int gr_generic_other_divexact_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3,
                                  slong len, gr_ctx_t ctx)
int gr_generic_other_pow_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong
                             len, gr_ctx_t ctx)
```

```
int gr_generic_vec_add_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                     cctx, gr_ctx_t ctx)
int gr_generic_vec_sub_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                     cctx, gr_ctx_t ctx)
int gr_generic_vec_mul_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                     cctx, gr_ctx_t ctx)
int gr_generic_vec_div_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                     cctx, gr_ctx_t ctx)
int gr_generic_vec_divexact_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c,
                                         gr_ctx_t cctx, gr_ctx_t ctx)
int gr_generic_vec_pow_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                     cctx, gr_ctx_t ctx)
int gr_generic_scalar_other_add_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2,
                                     slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_sub_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2,
                                     slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_mul_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2,
                                     slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_div_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2,
                                     slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_divexact_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr
                                         vec2, slong len, gr_ctx_t ctx)
int gr_generic_scalar_other_pow_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2,
                                     slong len, gr_ctx_t ctx)
```

3.4.5 Generic polynomial methods

```
int gr_generic_poly_factor_roots(gr_ptr c, gr_vec_t fac, gr_vec_t mult, gr_srcptr elt, int flags,
                                 gr_ctx_t ctx)
```

Calls *gr_poly_roots()* to factor a polynomial that splits into linear factors.

3.5 gr_special.h – special arithmetic and transcendental functions

3.5.1 Mathematical constants

```
int gr_pi(gr_ptr res, gr_ctx_t ctx)
int gr_euler(gr_ptr res, gr_ctx_t ctx)
int gr_catalan(gr_ptr res, gr_ctx_t ctx)
int gr_khinchin(gr_ptr res, gr_ctx_t ctx)
int gr_glaisher(gr_ptr res, gr_ctx_t ctx)
```

Standard real constants: π , Euler's constant γ , Catalan's constant, Khinchin's constant, Glaisher's constant.

3.5.2 Elementary functions

```
int gr_exp(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_expml(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_exp2(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_exp10(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_exp_pi_i(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log1p(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log2(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log10(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_log_pi_i(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_sin(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cos(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_cos(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_ctx_t ctx)
int gr_tan(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cot(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sec(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_csc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_sin_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cos_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sin_cos_pi(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_ctx_t ctx)
int gr_tan_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cot_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sec_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_csc_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_sinc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sinc_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_sinh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_cosh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sinh_cosh(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_ctx_t ctx)
int gr_tanh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_coth(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_sech(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```



```

int gr_csch(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_asin(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acos(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atan(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atan2(gr_ptr res, gr_srcptr y, gr_srcptr x, gr_ctx_t ctx)
int gr_acot(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asec(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acsc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_asin_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acos_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atan_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acot_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asec_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acsc_pi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_asinh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acosh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_atanh(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acoth(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_asech(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_acsch(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_lambertw(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_lambertw_fmpz(gr_ptr res, gr_srcptr x, const fmpz_t k, gr_ctx_t ctx)

```

3.5.3 Factorials and gamma functions

```

int gr_fac(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_fac_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_fac_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_fac_vec(gr_ptr res, slong len, gr_ctx_t ctx)
    Factorial  $x!$ . The vec version writes the first len consecutive values  $1, 1, 2, 6, \dots, (len - 1)!$  to the
    preallocated vector res.

int gr_rfac(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_rfac_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_rfac_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
int gr_rfac_vec(gr_ptr res, slong len, gr_ctx_t ctx)
    Reciprocal factorial. The vec version writes the first len consecutive values
     $1, 1, 1/2, 1/6, \dots, 1/(len - 1)!$  to the preallocated vector res.

int gr_bin(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bin_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)
int gr_bin_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)
int gr_bin_vec(gr_ptr res, gr_srcptr x, slong len, gr_ctx_t ctx)
int gr_bin_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
    Binomial coefficient  $\binom{x}{y}$ . The vec versions write the first len consecutive values  $\binom{x}{0}, \binom{x}{1}, \dots, \binom{x}{len-1}$ 
    to the preallocated vector res. For constructing a two-dimensional array of binomial coefficients
    (Pascal's triangle), it is more efficient to call gr_mat_pascal() than to call these functions repeat-
    edly.

```

`int gr_rising(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`
`int gr_rising_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)`
`int gr_falling(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`
`int gr_falling_ui(gr_ptr res, gr_srcptr x, ulong y, gr_ctx_t ctx)`
Rising and falling factorials $x(x+1)\cdots(x+y-1)$ and $x(x-1)\cdots(x-y+1)$, or their generalizations to non-integer y via the gamma function.

`int gr_gamma(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
`int gr_gamma_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)`
`int gr_gamma_fmpq(gr_ptr res, const fmpq_t x, gr_ctx_t ctx)`
`int gr_rgamma(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
`int gr_lgamma(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
`int gr_digamma(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
Gamma function $\Gamma(x)$, its reciprocal $1/\Gamma(x)$, the log-gamma function $\log\Gamma(x)$, and the digamma function $\psi(x)$.

`int gr_barnes_g(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
`int gr_log_barnes_g(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
Barnes G-function.

`int gr_beta(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)`
Beta function $B(x, y)$.

`int gr_doublefac(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
`int gr_doublefac_ui(gr_ptr res, ulong x, gr_ctx_t ctx)`
Double factorial $x!!$.

`int gr_harmonic(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)`
`int gr_harmonic_ui(gr_ptr res, ulong x, gr_ctx_t ctx)`
Harmonic number H_x .

3.5.4 Combinatorial numbers

The *vec* version of functions for number sequences c_n write the *len* consecutive values $c_0, c_1, \dots, c_{len-1}$ to the preallocated vector *res*.

`int gr_bernoulli_ui(gr_ptr res, ulong n, gr_ctx_t ctx)`
`int gr_bernoulli_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)`
`int gr_bernoulli_vec(gr_ptr res, slong len, gr_ctx_t ctx)`
Bernoulli numbers B_n .

`int gr_eulernum_ui(gr_ptr res, ulong x, gr_ctx_t ctx)`
`int gr_eulernum_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)`
`int gr_eulernum_vec(gr_ptr res, slong len, gr_ctx_t ctx)`
Euler numbers E_n .

`int gr_fib_ui(gr_ptr res, ulong n, gr_ctx_t ctx)`
`int gr_fib_fmpz(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)`
`int gr_fib_vec(gr_ptr res, slong len, gr_ctx_t ctx)`
Fibonacci numbers F_n .

`int gr_stirling_s1u_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)`
`int gr_stirling_s1_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)`
`int gr_stirling_s2_uiui(gr_ptr res, ulong x, ulong y, gr_ctx_t ctx)`
`int gr_stirling_s1u_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)`
`int gr_stirling_s1_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)`


```
int gr_stirling_s2_ui_vec(gr_ptr res, ulong x, slong len, gr_ctx_t ctx)
```

Stirling numbers $S(x, y)$: unsigned of the first kind, signed of the first kind, and second kind. The *vec* versions write the *len* consecutive values $S(x, 0), S(x, 1), \dots, S(x, len - 1)$ to the preallocated vector *res*. For constructing a two-dimensional array of Stirling numbers, it is more efficient to call *gr_mat_stirling()* than to call these functions repeatedly.

```
int gr_bellnum_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
```

```
int gr_bellnum_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
```

```
int gr_bellnum_vec(gr_ptr res, slong len, gr_ctx_t ctx)
```

Bell numbers B_n .

```
int gr_partitions_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
```

```
int gr_partitions_fmpz(gr_ptr res, const fmpz_t x, gr_ctx_t ctx)
```

```
int gr_partitions_vec(gr_ptr res, slong len, gr_ctx_t ctx)
```

Partition numbers $p(n)$.

3.5.5 Error function and exponential integrals

```
int gr_erf(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_erfc(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_erfcx(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_erfi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_erfinv(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_erfcinv(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_fresnel_s(gr_ptr res, gr_srcptr x, int normalized, gr_ctx_t ctx)
```

```
int gr_fresnel_c(gr_ptr res, gr_srcptr x, int normalized, gr_ctx_t ctx)
```

```
int gr_fresnel(gr_ptr res1, gr_ptr res2, gr_srcptr x, int normalized, gr_ctx_t ctx)
```

```
int gr_gamma_upper(gr_ptr res, gr_srcptr x, gr_srcptr y, int regularized, gr_ctx_t ctx)
```

```
int gr_gamma_lower(gr_ptr res, gr_srcptr x, gr_srcptr y, int regularized, gr_ctx_t ctx)
```

```
int gr_beta_lower(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int regularized, gr_ctx_t ctx)
```

```
int gr_exp_integral(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
```

```
int gr_exp_integral_ei(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_sin_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_cos_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_sinh_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_cosh_integral(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_log_integral(gr_ptr res, gr_srcptr x, int offset, gr_ctx_t ctx)
```

```
int gr_dilog(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
```

3.5.6 Orthogonal polynomials

```
int gr_chebyshev_t_fmpz(gr_ptr res, const fmpz_t n, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_chebyshev_t(gr_ptr res, gr_srcptr n, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_chebyshev_u_fmpz(gr_ptr res, const fmpz_t n, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_chebyshev_u(gr_ptr res, gr_srcptr n, gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_jacobi_p(gr_ptr res, gr_srcptr n, gr_srcptr a, gr_srcptr b, gr_srcptr z, gr_ctx_t ctx)
```

```
int gr_gegenbauer_c(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_srcptr z, gr_ctx_t ctx)
```

```
int gr_laguerre_l(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_srcptr z, gr_ctx_t ctx)
int gr_hermite_h(gr_ptr res, gr_srcptr n, gr_srcptr z, gr_ctx_t ctx)
int gr_legendre_p(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_srcptr z, int type, gr_ctx_t ctx)
int gr_legendre_q(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_srcptr z, int type, gr_ctx_t ctx)
int gr_spherical_y_si(gr_ptr res, slong n, slong m, gr_srcptr theta, gr_srcptr phi, gr_ctx_t ctx)
int gr_legendre_p_root_ui(gr_ptr root, gr_ptr weight, ulong n, ulong k, gr_ctx_t ctx)
```

3.5.7 Bessel, Airy and Coulomb functions

```
int gr_bessel_j(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_y(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_i(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_k(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_j_y(gr_ptr res1, gr_ptr res2, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_i_scaled(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_bessel_k_scaled(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)

int gr_airy(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_ptr res4, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_ai(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_bi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_ai_prime(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_airy_bi_prime(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)

int gr_airy_ai_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_airy_bi_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_airy_ai_prime_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_airy_bi_prime_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)

int gr_coulomb(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_ptr res4, gr_srcptr x, gr_srcptr y, gr_srcptr
    z, gr_ctx_t ctx)
int gr_coulomb_f(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_g(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_hpos(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_coulomb_hneg(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
```

3.5.8 Hypergeometric functions

```
int gr_hypgeom_0f1(gr_ptr res, gr_srcptr a, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_hypgeom_1f1(gr_ptr res, gr_srcptr a, gr_srcptr b, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_hypgeom_u(gr_ptr res, gr_srcptr a, gr_srcptr b, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_hypgeom_2f1(gr_ptr res, gr_srcptr a, gr_srcptr b, gr_srcptr c, gr_srcptr z, int flags, gr_ctx_t
    ctx)
int gr_hypgeom_pfq(gr_ptr res, const gr_vec_t a, const gr_vec_t b, gr_srcptr z, int flags, gr_ctx_t
    ctx)
```

3.5.9 Riemann zeta, polylogarithms and Dirichlet L-functions

```

int gr_zeta(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_zeta_ui(gr_ptr res, ulong x, gr_ctx_t ctx)
int gr_hurwitz_zeta(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_polygamma(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_polylog(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)
int gr_lerch_phi(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_ctx_t ctx)
int gr_stieltjes(gr_ptr res, const fmpz_t x, gr_srcptr y, gr_ctx_t ctx)

int gr_dirichlet_eta(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_riemann_xi(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_zeta_zero(gr_ptr res, const fmpz_t n, gr_ctx_t ctx)
int gr_zeta_zero_vec(gr_ptr res, const fmpz_t n, slong len, gr_ctx_t ctx)
int gr_zeta_nzeros(gr_ptr res, gr_srcptr t, gr_ctx_t ctx)

int gr_dirichlet_chi_fmpz(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, const
    fmpz_t n, gr_ctx_t ctx)
int gr_dirichlet_chi_vec(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, slong
    len, gr_ctx_t ctx)
int gr_dirichlet_l(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi, gr_srcptr s,
    gr_ctx_t ctx)
int gr_dirichlet_l_all(gr_vec_t res, const dirichlet_group_t G, gr_srcptr s, gr_ctx_t ctx)
int gr_dirichlet_hardy_theta(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi,
    gr_srcptr t, gr_ctx_t ctx)
int gr_dirichlet_hardy_z(gr_ptr res, const dirichlet_group_t G, const dirichlet_char_t chi,
    gr_srcptr t, gr_ctx_t ctx)

```

3.5.10 Elliptic integrals

```

int gr_agm1(gr_ptr res, gr_srcptr x, gr_ctx_t ctx)
int gr_agm(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_ctx_t ctx)

int gr_elliptic_k(gr_ptr res, gr_srcptr m, gr_ctx_t ctx)
int gr_elliptic_e(gr_ptr res, gr_srcptr m, gr_ctx_t ctx)
int gr_elliptic_pi(gr_ptr res, gr_srcptr n, gr_srcptr m, gr_ctx_t ctx)
int gr_elliptic_f(gr_ptr res, gr_srcptr phi, gr_srcptr m, int pi, gr_ctx_t ctx)
int gr_elliptic_e_inc(gr_ptr res, gr_srcptr phi, gr_srcptr m, int pi, gr_ctx_t ctx)
int gr_elliptic_pi_inc(gr_ptr res, gr_srcptr n, gr_srcptr phi, gr_srcptr m, int pi, gr_ctx_t ctx)

int gr_carlson_rc(gr_ptr res, gr_srcptr x, gr_srcptr y, int flags, gr_ctx_t ctx)
int gr_carlson_rf(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_carlson_rd(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_carlson_rg(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, int flags, gr_ctx_t ctx)
int gr_carlson_rj(gr_ptr res, gr_srcptr x, gr_srcptr y, gr_srcptr z, gr_srcptr w, int flags, gr_ctx_t
    ctx)

```

3.5.11 Elliptic, modular and theta functions

```
int gr_jacobi_theta(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_ptr res4, gr_srcptr z, gr_srcptr tau,
                   gr_ctx_t ctx)
int gr_jacobi_theta_1(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_2(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_3(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_jacobi_theta_4(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)

int gr_dedekind_eta(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_dedekind_eta_q(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)

int gr_modular_j(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_modular_lambda(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)
int gr_modular_delta(gr_ptr res, gr_srcptr tau, gr_ctx_t ctx)

int gr_hilbert_class_poly(gr_ptr res, slong D, gr_srcptr x, gr_ctx_t ctx)

int gr_eisenstein_e(gr_ptr res, ulong n, gr_srcptr tau, gr_ctx_t ctx)
int gr_eisenstein_g(gr_ptr res, ulong n, gr_srcptr tau, gr_ctx_t ctx)
int gr_eisenstein_g_vec(gr_ptr res, gr_srcptr tau, slong len, gr_ctx_t ctx)

int gr_elliptic_invariants(gr_ptr res1, gr_ptr res2, gr_srcptr tau, gr_ctx_t ctx)
int gr_elliptic_roots(gr_ptr res1, gr_ptr res2, gr_ptr res3, gr_srcptr tau, gr_ctx_t ctx)

int gr_weierstrass_p(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_p_prime(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_p_inv(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_zeta(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
int gr_weierstrass_sigma(gr_ptr res, gr_srcptr z, gr_srcptr tau, gr_ctx_t ctx)
```

3.6 gr_vec.h – vectors over generic rings

3.6.1 Types and basic operations

type **gr_vec_struct**

type **gr_vec_t**

void **gr_vec_init**(*gr_vec_t* vec, *slong* len, *gr_ctx_t* ctx)

Initializes *vec* to a vector of length *len* with elements in the ring *ctx*. The length must be nonnegative. All entries are set to zero.

void **gr_vec_clear**(*gr_vec_t* vec, *gr_ctx_t* ctx)

Clears the vector *vec*.

GR_VEC_ENTRY(vec, i, sz)

Macro to access the *i*-th element in the vector *vec*, indexed from zero, assuming that entries have size *sz*. The index must be in bounds.

gr_ptr **gr_vec_entry_ptr**(*gr_vec_t* vec, *slong* i, *gr_ctx_t* ctx)

gr_srcptr **gr_vec_entry_srcptr**(*gr_vec_t* vec, *slong* i, *gr_ctx_t* ctx)

Returns a pointer to the *i*-th element in the vector *vec*, indexed from zero. The index must be in bounds.

slong **gr_vec_length**(const *gr_vec_t* vec, *gr_ctx_t* ctx)

Returns the length of the vector *vec*.

void **gr_vec_fit_length**(*gr_vec_t* vec, *slong* len, *gr_ctx_t* ctx)

Allocates space for at least *len* elements in the vector *vec*. This does not change the size of the vector.

void **gr_vec_set_length**(*gr_vec_t* vec, *slong* len, *gr_ctx_t* ctx)

Resizes the vector to length *len*, which must be nonnegative. The vector will be extended with zeros.

int **gr_vec_set**(*gr_vec_t* res, const *gr_vec_t* src, *gr_ctx_t* ctx)

Sets *res* to a copy of the vector *src*.

int **gr_vec_append**(*gr_vec_t* vec, *gr_srcptr* x, *gr_ctx_t* ctx)

Appends the element *x* to the end of vector *vec*.

int **_gr_vec_write**(*gr_stream_t* out, *gr_srcptr* vec, *slong* len, *gr_ctx_t* ctx)

int **gr_vec_write**(*gr_stream_t* out, const *gr_vec_t* vec, *gr_ctx_t* ctx)

int **gr_vec_print**(const *gr_vec_t* vec, *gr_ctx_t* ctx)

GR_ENTRY(vec, i, size)

Macro to access the *i*-th entry of a *gr_ptr* or *gr_srcptr* vector *vec*, where each element is *size* bytes.

void **_gr_vec_init**(*gr_ptr* vec, *slong* len, *gr_ctx_t* ctx)

Initialize *len* elements of *vec* to the value 0. The pointer *vec* must already refer to allocated memory.

void **_gr_vec_clear**(*gr_ptr* vec, *slong* len, *gr_ctx_t* ctx)

Clears *len* elements of *vec*. This frees memory allocated by individual elements, but does not free the memory allocated by *vec* itself.

void **_gr_vec_swap**(*gr_ptr* vec1, *gr_ptr* vec2, *slong* len, *gr_ctx_t* ctx)

Swap the entries of *vec1* and *vec2*.

int **_gr_vec_randtest**(*gr_ptr* res, *flint_rand_t* state, *slong* len, *gr_ctx_t* ctx)

```
int _gr_vec_set(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)

truth_t _gr_vec_equal(gr_srcptr vec1, gr_srcptr vec2, slong len, gr_ctx_t ctx)

int _gr_vec_zero(gr_ptr vec, slong len, gr_ctx_t ctx)

truth_t _gr_vec_is_zero(gr_srcptr vec, slong len, gr_ctx_t ctx)

int _gr_vec_normalise(slong *res, gr_srcptr vec, slong len, gr_ctx_t ctx)

slong _gr_vec_normalise_weak(gr_srcptr vec, slong len, gr_ctx_t ctx)
```

3.6.2 Arithmetic

```
int _gr_vec_neg(gr_ptr res, gr_srcptr src, slong len, gr_ctx_t ctx)

int _gr_vec_add(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int _gr_vec_sub(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int _gr_vec_mul(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int _gr_vec_div(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int _gr_vec_divexact(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
int _gr_vec_pow(gr_ptr res, gr_srcptr src1, gr_srcptr src2, slong len, gr_ctx_t ctx)
```

Binary operations applied elementwise.

```
int _gr_vec_add_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_sub_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_mul_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_div_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_pow_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_scalar_add_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int _gr_scalar_sub_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int _gr_scalar_mul_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int _gr_scalar_div_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int _gr_scalar_divexact_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
int _gr_scalar_pow_vec(gr_ptr vec1, gr_srcptr c, gr_srcptr vec2, slong len, gr_ctx_t ctx)
```

Binary operations applied elementwise with a fixed scalar operand.

```
int _gr_vec_add_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong len,
                     gr_ctx_t ctx)
int _gr_vec_sub_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong len,
                     gr_ctx_t ctx)
int _gr_vec_mul_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong len,
                     gr_ctx_t ctx)
int _gr_vec_div_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong len,
                     gr_ctx_t ctx)
int _gr_vec_divexact_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong len,
                           gr_ctx_t ctx)
int _gr_vec_pow_other(gr_ptr vec1, gr_srcptr vec2, gr_srcptr vec3, gr_ctx_t ctx3, slong len,
                     gr_ctx_t ctx)
int _gr_other_add_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len,
                     gr_ctx_t ctx)
```

```

int _gr_other_sub_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len,
                     gr_ctx_t ctx)
int _gr_other_mul_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len,
                     gr_ctx_t ctx)
int _gr_other_div_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len,
                     gr_ctx_t ctx)
int _gr_other_divexact_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len,
                           gr_ctx_t ctx)
int _gr_other_pow_vec(gr_ptr vec1, gr_srcptr vec2, gr_ctx_t ctx2, gr_srcptr vec3, slong len,
                     gr_ctx_t ctx)

```

Binary operations applied elementwise, allowing a different type for one of the vectors.

```

int _gr_vec_add_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx,
                             gr_ctx_t ctx)
int _gr_vec_sub_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx,
                             gr_ctx_t ctx)
int _gr_vec_mul_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx,
                             gr_ctx_t ctx)
int _gr_vec_div_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx,
                             gr_ctx_t ctx)
int _gr_vec_divexact_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t
                                cctx, gr_ctx_t ctx)
int _gr_vec_pow_scalar_other(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t cctx,
                             gr_ctx_t ctx)
int _gr_scalar_other_add_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len,
                             gr_ctx_t ctx)
int _gr_scalar_other_sub_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len,
                             gr_ctx_t ctx)
int _gr_scalar_other_mul_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len,
                             gr_ctx_t ctx)
int _gr_scalar_other_div_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len,
                             gr_ctx_t ctx)
int _gr_scalar_other_divexact_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong
                                len, gr_ctx_t ctx)
int _gr_scalar_other_pow_vec(gr_ptr vec1, gr_srcptr c, gr_ctx_t cctx, gr_srcptr vec2, slong len,
                             gr_ctx_t ctx)

int _gr_vec_add_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_sub_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_mul_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_div_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_pow_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_add_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int _gr_vec_sub_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int _gr_vec_mul_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int _gr_vec_div_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int _gr_vec_pow_scalar_ui(gr_ptr vec1, gr_srcptr vec2, slong len, ulong c, gr_ctx_t ctx)
int _gr_vec_add_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_sub_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)

```



```
int _gr_vec_mul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_div_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t
                                ctx)

int _gr_vec_pow_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_add_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_sub_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_mul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_div_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
int _gr_vec_divexact_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t
                                ctx)

int _gr_vec_pow_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t ctx)
    Binary operations applied elementwise with a fixed scalar operand, allowing a different type for
    the scalar.

int _gr_vec_addmul_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_submul_scalar(gr_ptr vec1, gr_srcptr vec2, slong len, gr_srcptr c, gr_ctx_t ctx)
int _gr_vec_addmul_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_submul_scalar_si(gr_ptr vec1, gr_srcptr vec2, slong len, slong c, gr_ctx_t ctx)
int _gr_vec_addmul_scalar_fmpz(gr_ptr vec1, gr_srcptr vec2, slong len, const fmpz_t c, gr_ctx_t
                                ctx)

int _gr_vec_mul_scalar_2exp_si(gr_ptr res, gr_srcptr vec, slong len, slong c, gr_ctx_t ctx)
```

3.6.3 Sums and products

```
int _gr_vec_sum(gr_ptr res, gr_srcptr vec, slong len, gr_ctx_t ctx)
int _gr_vec_product(gr_ptr res, gr_srcptr vec, slong len, gr_ctx_t ctx)
```

3.6.4 Dot products

```
int _gr_vec_dot(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2, slong len,
               gr_ctx_t ctx)
int _gr_vec_dot_si(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const slong *vec2,
                  slong len, gr_ctx_t ctx)
int _gr_vec_dot_ui(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const ulong *vec2,
                  slong len, gr_ctx_t ctx)
int _gr_vec_dot_fmpz(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, const fmpz *vec2,
                    slong len, gr_ctx_t ctx)

    Sets res to  $c \pm \sum_{i=0}^{n-1} a_i b_i$ .

int _gr_vec_dot_rev(gr_ptr res, gr_srcptr initial, int subtract, gr_srcptr vec1, gr_srcptr vec2, slong
                  len, gr_ctx_t ctx)

    Sets res to  $c \pm \sum_{i=0}^{n-1} a_i b_{n-1-i}$ .
```


3.6.5 Sorting and searching

truth_t **_gr_vec_contains**(*gr_srcptr* vec, *slong* len, *gr_srcptr* x, *gr_ctx_t* ctx)

truth_t **gr_vec_contains**(const *gr_vec_t* vec, *gr_srcptr* x, *gr_ctx_t* ctx);

int **_gr_vec_sort**(*gr_ptr* vec, *slong* len, *gr_ctx_t* ctx)

int **gr_vec_sort**(*gr_vec_t* dest, const *gr_vec_t* src, *gr_ctx_t* ctx)

Sorts the entries in increasing order. The underscore version works in-place; when it returns a status code other than *GR_SUCCESS*, the vector may not be sorted but still contains some permutation of the original entries.

void **_gr_vec_permute**(*gr_ptr* vec, *slong* *perm, *slong* len, *gr_ctx_t* ctx)

int **gr_vec_permute**(*gr_vec_t* dest, *gr_vec_t* src, *slong* *perm, *gr_ctx_t* ctx)

Applies a permutation to a vector. The underscore version works in-place and overwrites *perm*.

void **_gr_vec_shuffle**(*gr_ptr* vec, *flint_rand_t* state, *slong* len, *gr_ctx_t* ctx)

Applies a uniform random permutation.

3.6.6 Other functions

int **_gr_vec_step**(*gr_ptr* vec, *gr_srcptr* start, *gr_srcptr* step, *slong* len, *gr_ctx_t* ctx)

int **_gr_vec_reciprocals**(*gr_ptr* res, *slong* len, *gr_ctx_t* ctx)

Sets *res* to the vector of reciprocals of the positive integers 1, 2, ... up to *len* inclusive.

int **_gr_vec_set_powers**(*gr_ptr* res, *gr_srcptr* x, *slong* len, *gr_ctx_t* ctx)

3.7 `gr_mat.h` – dense matrices over generic rings

A `gr_mat_t` represents a matrix implemented as a dense array of entries in a generic ring R .

- In this module, the context object `ctx` always represents the coefficient ring R unless otherwise stated. Creating a context object representing a matrix space only becomes necessary when one wants to manipulate matrices using generic ring methods like `gr_add` instead of the designated matrix methods like `gr_mat_add`.
- Matrix functions generally assume that input as well as output operands have compatible shapes. Some functions return `GR_DOMAIN` for matrices with the wrong shape, but this is not always consistent.
- Some operations (like rank, LU factorization) generally only make sense when the base ring is an integral domain. Typically the algorithms designed for integral domains also work over non-integral domains as long as all inversions of nonzero elements succeed. If an inversion fails, the algorithm will return the `GR_DOMAIN` or `GR_UNABLE` flag. This might not yet be entirely consistent.

3.7.1 Type compatibility

The `gr_mat` type has the same data layout as most FLINT, Arb and Calcium matrix types. Methods in this module can therefore be mixed freely with methods in the corresponding FLINT, Arb and Calcium modules when the underlying coefficient type is the same.

It is not directly compatible with the `nmod_mat` type, which stores modulus data as part of the matrix object.

3.7.2 Types, macros and constants

type `gr_mat_struct`

type `gr_mat_t`

Contains a pointer to an array of coefficients (`entries`), the number of rows (`r`), the number of columns (`c`), and an array to pointers marking the start of each row (`rows`).

A `gr_mat_t` is defined as an array of length one of type `gr_mat_struct`, permitting a `gr_mat_t` to be passed by reference.

`GR_MAT_ENTRY(mat, i, j, sz)`

Macro to access the entry at row i and column j of the matrix mat whose entries have size sz bytes.

`gr_ptr gr_mat_entry_ptr(gr_mat_t mat, slong i, slong j, gr_ctx_t ctx)`

Function returning a pointer to the entry at row i and column j of the matrix mat . The indices must be in bounds.

`gr_mat_nrows(mat, ctx)`

Macro accessing the number of rows of mat .

`gr_mat_ncols(mat, ctx)`

Macro accessing the number of columns of mat .

3.7.3 Memory management

void **gr_mat_init**(*gr_mat_t* mat, *slong* rows, *slong* cols, *gr_ctx_t* ctx)

Initializes *mat* to a matrix with the given number of rows and columns.

int **gr_mat_init_set**(*gr_mat_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Initializes *res* to a copy of the matrix *mat*.

void **gr_mat_clear**(*gr_mat_t* mat, *gr_ctx_t* ctx)

Clears the matrix.

void **gr_mat_swap**(*gr_mat_t* mat1, *gr_mat_t* mat2, *gr_ctx_t* ctx)

Swaps *mat1* and *mat2* efficiently.

int **gr_mat_swap_entrywise**(*gr_mat_t* mat1, const *gr_mat_t* mat2, *gr_ctx_t* ctx)

Performs a deep swap of *mat1* and *mat2*, swapping the individual entries rather than the top-level structures.

3.7.4 Window matrices

void **gr_mat_window_init**(*gr_mat_t* window, const *gr_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2, *gr_ctx_t* ctx)

Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive). The indices must be within bounds.

void **gr_mat_window_clear**(*gr_mat_t* window, *gr_ctx_t* ctx)

Frees the window matrix.

GR_MAT_TMP_INIT_SHALLOW_TRANSPOSE(*AT*, *A*, ctx)

Initialize *AT* to a shallow transpose of *A* for temporary use. This macro uses stack allocation if *A* is sufficiently small. The matrix *AT* can be used similarly to a window matrix for reading. It can also be used for writing, provided that one finishes the operation by performing **GR_MAT_SHALLOW_TRANSPOSE**(*A*, *AT*, ctx) to write back any changes to the shallow data.

GR_MAT_SHALLOW_TRANSPOSE(*AT*, *A*, ctx)

Sets *AT* to the transpose of *A*, copying entries shallowly. Assumes that *AT* and *A* are not aliased (if aliased, the normal *gr_mat_transpose()* already uses shallow operations).

GR_MAT_TMP_CLEAR_SHALLOW_TRANSPOSE(*AT*, ctx)

Free the shallow transpose allocated by **GR_MAT_TMP_INIT_SHALLOW_TRANSPOSE**.

3.7.5 Input and output

int **gr_mat_write**(*gr_stream_t* out, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Write *mat* to the stream *out*.

int **gr_mat_print**(const *gr_mat_t* mat, *gr_ctx_t* ctx)

Prints *mat* to standard output.

3.7.6 Comparisons

`truth_t gr_mat_equal(const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)`

Returns whether *mat1* and *mat2* are equal.

3.7.7 Assignment and special values

`truth_t gr_mat_is_zero(const gr_mat_t mat, gr_ctx_t ctx)`

`truth_t gr_mat_is_one(const gr_mat_t mat, gr_ctx_t ctx)`

`truth_t gr_mat_is_neg_one(const gr_mat_t mat, gr_ctx_t ctx)`

Returns whether *mat* respectively is the zero matrix or the scalar matrix with 1 or -1 on the main diagonal.

`truth_t gr_mat_is_scalar(const gr_mat_t mat, gr_ctx_t ctx)`

Returns whether *mat* is a scalar matrix, being a diagonal matrix with identical elements on the main diagonal.

`int gr_mat_zero(gr_mat_t res, gr_ctx_t ctx)`

Sets *res* to the zero matrix.

`int gr_mat_one(gr_mat_t res, gr_ctx_t ctx)`

Sets *res* to the scalar matrix with 1 on the main diagonal and zero elsewhere.

`int gr_mat_set(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)`

`int gr_mat_set_fmpz_mat(gr_mat_t res, const fmpz_mat_t mat, gr_ctx_t ctx)`

`int gr_mat_set_fmpq_mat(gr_mat_t res, const fmpq_mat_t mat, gr_ctx_t ctx)`

`int gr_mat_set_gr_mat_other(gr_mat_t res, const gr_mat_t mat, gr_ctx_t mat_ctx, gr_ctx_t ctx)`

Sets *res* to the value of *mat*.

`int gr_mat_set_scalar(gr_mat_t res, gr_srcptr c, gr_ctx_t ctx)`

`int gr_mat_set_ui(gr_mat_t res, ulong c, gr_ctx_t ctx)`

`int gr_mat_set_si(gr_mat_t res, slong c, gr_ctx_t ctx)`

`int gr_mat_set_fmpz(gr_mat_t res, const fmpz_t c, gr_ctx_t ctx)`

`int gr_mat_set_fmpq(gr_mat_t res, const fmpq_t c, gr_ctx_t ctx)`

Set *res* to the scalar matrix with *c* on the main diagonal and zero elsewhere.

3.7.8 Basic row, column and entry operations

`int gr_mat_concat_horizontal(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)`

`int gr_mat_concat_vertical(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)`

`int gr_mat_transpose(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)`

Sets *res* to the transpose of *mat*. Dimensions must be compatible. Aliasing is allowed for square matrices.

`int gr_mat_swap_rows(gr_mat_t mat, slong *perm, slong r, slong s, gr_ctx_t ctx)`

Swaps rows *r* and *s* of *mat*. If *perm* is non-NULL, the permutation of the rows will also be applied to *perm*.

`int gr_mat_swap_cols(gr_mat_t mat, slong *perm, slong r, slong s, gr_ctx_t ctx)`

Swaps columns *r* and *s* of *mat*. If *perm* is non-NULL, the permutation of the columns will also be applied to *perm*.

int **gr_mat_invert_rows**(*gr_mat_t* mat, *slong* *perm, *gr_ctx_t* ctx)

Swaps rows *i* and *r - i* of *mat* for $0 \leq i < r/2$, where *r* is the number of rows of *mat*. If *perm* is non-NULL, the permutation of the rows will also be applied to *perm*.

int **gr_mat_invert_cols**(*gr_mat_t* mat, *slong* *perm, *gr_ctx_t* ctx)

Swaps columns *i* and *c - i* of *mat* for $0 \leq i < c/2$, where *c* is the number of columns of *mat*. If *perm* is non-NULL, the permutation of the columns will also be applied to *perm*.

int **gr_mat_move_row**(*gr_mat_t* A, *slong* i, *slong* new_i, *gr_ctx_t* ctx)

Moves row *i* to the new position *new_i*, displacing all intervening rows. For example, with *i* = 4 and *new_i* = 7 this replaces the block of rows A4, A5, A6, A7 by A5, A6, A7, A4. With *i* = 7 and *new_i* = 4 replaces A4, A5, A6, A7 by A7, A4, A5, A6. Returns GR_DOMAIN if either index is not in bounds, otherwise is guaranteed to succeed.

truth_t **gr_mat_is_empty**(const *gr_mat_t* mat, *gr_ctx_t* ctx)

Returns whether *mat* is an empty matrix, having either zero rows or zero column. This predicate is always decidable (even if the underlying ring is not computable), returning T_TRUE or T_FALSE.

truth_t **gr_mat_is_square**(const *gr_mat_t* mat, *gr_ctx_t* ctx)

Returns whether *mat* is a square matrix, having the same number of rows as columns (not the same thing as being a perfect square!). This predicate is always decidable (even if the underlying ring is not computable), returning T_TRUE or T_FALSE.

3.7.9 Entrywise operations

int **gr_mat_entrywise_unary_op**(*gr_mat_t* res, *gr_method_unary_op* f, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Sets *res* to the application of the function *f* to the entries of matrix *mat*. Returns GR_DOMAIN if the matrix dimensions do not match.

int **gr_mat_entrywise_binary_op**(*gr_mat_t* res, *gr_method_binary_op* f, const *gr_mat_t* mat1, const *gr_mat_t* mat2, *gr_ctx_t* ctx)

Sets *res* to the application of the function *f* to the entries of *mat1* as first argument and the entries of *mat2* as second argument. Returns GR_DOMAIN if the matrix dimensions do not match.

int **gr_mat_entrywise_binary_op_scalar**(*gr_mat_t* res, *gr_method_binary_op* f, const *gr_mat_t* mat, *gr_sreptr* c, *gr_ctx_t* ctx)

Sets *res* to the application of the function *f* to the entries of *mat* as first argument and the scalar *c* as second argument. Returns GR_DOMAIN if the matrix dimensions do not match.

truth_t **gr_mat_entrywise_unary_predicate_all**(*gr_method_unary_predicate* f, const *gr_mat_t* mat, *gr_ctx_t* ctx)

truth_t **gr_mat_entrywise_unary_predicate_any**(*gr_method_unary_predicate* f, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Returns whether the predicate *f* is true for all entries, respectively for any entry, in the matrix *mat*.

truth_t **gr_mat_entrywise_binary_predicate_all**(*gr_method_binary_predicate* f, const *gr_mat_t* mat1, const *gr_mat_t* mat2, *gr_ctx_t* ctx)

Returns whether the binary predicate *f* is true for all entries in *mat1* paired with the corresponding entries in *mat2*. Returns T_FALSE if the matrix dimensions are not compatible.

3.7.10 Norms

int `gr_mat_norm_max`(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Max norm: $\max_{i,j} |a_{i,j}|$.

int `gr_mat_norm_1`(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

1-norm (largest absolute column sum): $\max_{1 \leq j \leq n} \sum_{i=1}^m |a_{i,j}|$.

int `gr_mat_norm_inf`(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Infinity-norm (largest absolute row sum): $\max_{1 \leq i \leq m} \sum_{j=1}^n |a_{i,j}|$.

int `gr_mat_norm_frobenius`(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Frobenius norm: $\sqrt{\sum_{i,j} |a_{i,j}|^2}$.

3.7.11 Addition and scalar arithmetic

int `gr_mat_neg`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int `gr_mat_add`(*gr_mat_t* res, const *gr_mat_t* mat1, const *gr_mat_t* mat2, *gr_ctx_t* ctx)

int `gr_mat_sub`(*gr_mat_t* res, const *gr_mat_t* mat1, const *gr_mat_t* mat2, *gr_ctx_t* ctx)

int `gr_mat_add_scalar`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_srcptr* x, *gr_ctx_t* ctx)

int `gr_mat_scalar_add`(*gr_mat_t* res, *gr_srcptr* x, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int `gr_mat_add_ui`(*gr_mat_t* res, const *gr_mat_t* mat, *ulong* x, *gr_ctx_t* ctx)

int `gr_mat_add_si`(*gr_mat_t* res, const *gr_mat_t* mat, *slong* x, *gr_ctx_t* ctx)

int `gr_mat_add_fmpz`(*gr_mat_t* res, const *gr_mat_t* mat, const *fmpz_t* x, *gr_ctx_t* ctx)

int `gr_mat_add_fmpq`(*gr_mat_t* res, const *gr_mat_t* mat, const *fmpq_t* x, *gr_ctx_t* ctx)

int `gr_mat_add_scalar_other`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_srcptr* x, *gr_ctx_t* x_ctx, *gr_ctx_t* ctx)

int `gr_mat_scalar_other_add`(*gr_mat_t* res, *gr_srcptr* x, *gr_ctx_t* x_ctx, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Perform the matrix-scalar or scalar-matrix operation $A + Ix$ or $Ix + A$.

int `gr_mat_sub_scalar`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_srcptr* x, *gr_ctx_t* ctx)

int `gr_mat_scalar_sub`(*gr_mat_t* res, *gr_srcptr* x, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int `gr_mat_sub_ui`(*gr_mat_t* res, const *gr_mat_t* mat, *ulong* x, *gr_ctx_t* ctx)

int `gr_mat_sub_si`(*gr_mat_t* res, const *gr_mat_t* mat, *slong* x, *gr_ctx_t* ctx)

int `gr_mat_sub_fmpz`(*gr_mat_t* res, const *gr_mat_t* mat, const *fmpz_t* x, *gr_ctx_t* ctx)

int `gr_mat_sub_fmpq`(*gr_mat_t* res, const *gr_mat_t* mat, const *fmpq_t* x, *gr_ctx_t* ctx)

int `gr_mat_sub_scalar_other`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_srcptr* x, *gr_ctx_t* x_ctx, *gr_ctx_t* ctx)

int `gr_mat_scalar_other_sub`(*gr_mat_t* res, *gr_srcptr* x, *gr_ctx_t* x_ctx, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Perform the matrix-scalar or scalar-matrix operation $A - Ix$ or $Ix - A$.

int `gr_mat_mul_scalar`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_srcptr* x, *gr_ctx_t* ctx)

int `gr_mat_scalar_mul`(*gr_mat_t* res, *gr_srcptr* x, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int `gr_mat_mul_ui`(*gr_mat_t* res, const *gr_mat_t* mat, *ulong* x, *gr_ctx_t* ctx)

int `gr_mat_mul_si`(*gr_mat_t* res, const *gr_mat_t* mat, *slong* x, *gr_ctx_t* ctx)

int `gr_mat_mul_fmpz`(*gr_mat_t* res, const *gr_mat_t* mat, const *fmpz_t* x, *gr_ctx_t* ctx)

int `gr_mat_mul_fmpq`(*gr_mat_t* res, const *gr_mat_t* mat, const *fmpq_t* x, *gr_ctx_t* ctx)

int `gr_mat_mul_scalar_other`(*gr_mat_t* res, const *gr_mat_t* mat, *gr_srcptr* x, *gr_ctx_t* x_ctx, *gr_ctx_t* ctx)

```
int gr_mat_scalar_other_mul(gr_mat_t res, gr_srcptr x, gr_ctx_t x_ctx, const gr_mat_t mat,
                           gr_ctx_t ctx)
```

Perform the matrix-scalar or scalar-matrix operation Ax or xA .

```
int gr_mat_div_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr x, gr_ctx_t ctx)
int gr_mat_div_scalar_other(gr_mat_t res, const gr_mat_t mat, gr_srcptr x, gr_ctx_t x_ctx,
                           gr_ctx_t ctx)
```

```
int gr_mat_div_ui(gr_mat_t res, const gr_mat_t mat, ulong x, gr_ctx_t ctx)
int gr_mat_div_si(gr_mat_t res, const gr_mat_t mat, slong x, gr_ctx_t ctx)
int gr_mat_div_fmpz(gr_mat_t res, const gr_mat_t mat, const fmpz_t x, gr_ctx_t ctx)
int gr_mat_div_fmpq(gr_mat_t res, const gr_mat_t mat, const fmpq_t x, gr_ctx_t ctx)
```

Perform the matrix-scalar operation A/x .

```
int gr_mat_addmul_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_submul_scalar(gr_mat_t res, const gr_mat_t mat, gr_srcptr c, gr_ctx_t ctx)
```

3.7.12 Matrix multiplication

```
int gr_mat_mul(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t ctx)
```

Compute AB using the default algorithm chosen by the element ring. If the element ring does not overload matrix multiplication, this will fall back to `gr_mat_mul_generic()` by default.

```
int gr_mat_mul_generic(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
```

Multiply matrices using a generic algorithm choice. Currently this always chooses classical multiplication, but may implement other strategies in the future.

```
int gr_mat_mul_classical(gr_mat_t res, const gr_mat_t mat1, const gr_mat_t mat2, gr_ctx_t
                        ctx)
```

Computes the $m \times n \times p$ matrix product using the classical algorithm, performing mp dot products of length n .

```
int gr_mat_mul_strassen(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
```

Uses Strassen's algorithm to evaluate AB using 7 recursive matrix multiplications of roughly half the size. This function calls `gr_mat_mul()` for the recursive multiplications; to use Strassen recursively, the base ring must overload `gr_mat_mul()` to choose Strassen multiplication above some cutoff. This results in $O(n^{2.81})$ asymptotic complexity in the case of an $n \times n \times n$ product.

This function does not implement Strassen's original evaluation sequence but that of Bodrato [Bodrato2010] which uses fewer additions and offers some further time savings when squaring.

```
int gr_mat_mul_waksman(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
```

Compute the product AB using roughly half the number of multiplications of the classical algorithm, using Waksman's algorithm [Waksman1970]. The base ring must be commutative and must support exact division by two.

```
int gr_mat_mul_rosowski(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
```

Compute the product AB using roughly half the number of multiplications of the classical algorithm, using Rosowski's algorithm [Rosowski2023]. The base ring must be commutative. This uses the same number of multiplications as Waksman's algorithm when the inner dimension n is even, but uses fewer multiplications when n is odd, and does not require division by two.

3.7.13 Powering

See the section on matrix functions for non-integer exponents.

```

int gr_mat_sqr(gr_mat_t res, const gr_mat_t mat, gr_ctx_t ctx)

int gr_mat_pow_ui(gr_mat_t res, const gr_mat_t mat, ulong e, gr_ctx_t ctx)
int gr_mat_pow_si(gr_mat_t res, const gr_mat_t mat, slong e, gr_ctx_t ctx)
int gr_mat_pow_fmpz(gr_mat_t res, const gr_mat_t mat, const fmpz_t e, gr_ctx_t ctx)

```

3.7.14 Polynomial evaluation

```

int _gr_mat_gr_poly_evaluate(gr_mat_t res, gr_srcptr poly, slong len, const gr_mat_t mat,
                             gr_ctx_t ctx)
int gr_mat_gr_poly_evaluate(gr_mat_t res, const gr_poly_t poly, const gr_mat_t mat, gr_ctx_t
                             ctx)

```

Sets *res* to the matrix obtained by evaluating the scalar polynomial *poly* with matrix argument *mat*.

3.7.15 Diagonal and triangular matrices

```

truth_t gr_mat_is_upper_triangular(const gr_mat_t mat, gr_ctx_t ctx)
truth_t gr_mat_is_lower_triangular(const gr_mat_t mat, gr_ctx_t ctx)

```

Returns whether *mat* is upper (respectively lower) triangular, having zeros everywhere below (respectively above) the main diagonal. The matrix need not be square.

```

truth_t gr_mat_is_diagonal(const gr_mat_t mat, gr_ctx_t ctx)

```

Returns whether *mat* is a diagonal matrix, having zeros everywhere except on the main diagonal. The matrix need not be square.

```

int gr_mat_mul_diag(gr_mat_t res, const gr_mat_t A, const gr_vec_t D, gr_ctx_t ctx)
int gr_mat_diag_mul(gr_mat_t res, const gr_vec_t D, const gr_mat_t A, gr_ctx_t ctx)

```

Set *res* to the product AD or DA respectively, where D is a diagonal matrix represented as a vector of entries.

3.7.16 Gaussian elimination

```

int gr_mat_find_nonzero_pivot_large_abs(slong *pivot_row, gr_mat_t mat, slong start_row,
                                         slong end_row, slong column, gr_ctx_t ctx)
int gr_mat_find_nonzero_pivot_generic(slong *pivot_row, gr_mat_t mat, slong start_row, slong
                                     end_row, slong column, gr_ctx_t ctx)
int gr_mat_find_nonzero_pivot(slong *pivot_row, gr_mat_t mat, slong start_row, slong end_row,
                              slong column, gr_ctx_t ctx)

```

Attempts to find a nonzero element in column number *column* of the matrix *mat* in a row between *start_row* (inclusive) and *end_row* (exclusive). On success, sets *pivot_row* to the row index and returns GR_SUCCESS. If no nonzero pivot element exists, returns GR_DOMAIN. If no nonzero pivot element exists and zero-testing fails for some element, returns the flag GR_UNABLE.

This function may be destructive: any elements that are nontrivially zero but can be certified zero may be overwritten by exact zeros.

```

int gr_mat_lu_classical(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check,
                       gr_ctx_t ctx)

```



```
int gr_mat_lu_recursive(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check,
                       gr_ctx_t ctx)
int gr_mat_lu_generic(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check,
                     gr_ctx_t ctx)
```

```
int gr_mat_lu(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check, gr_ctx_t ctx)
    Computes a generalized LU decomposition  $A = PLU$  of a given matrix  $A$ , writing the rank of  $A$ 
    to rank.
```

If A is a nonsingular square matrix, LU will be set to a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and set the rank to 0 if A is detected to be rank-deficient. This currently only works as expected for square matrices.

The algorithm can fail if it fails to certify that a pivot element is zero or nonzero, in which case the correct rank cannot be determined. It can also fail if a pivot element is not invertible. In these cases the `GR_UNABLE` and/or `GR_DOMAIN` flags will be returned. On failure, the data in the output variables `rank`, `P` and `LU` will be meaningless.

The *classical* version uses iterative Gaussian elimination. The *recursive* version uses a block recursive algorithm to take advantage of fast matrix multiplication. The *generic* version calls the recursive algorithm with a default cutoff.

```
int gr_mat_fflu(slong *rank, slong *P, gr_mat_t LU, gr_ptr den, const gr_mat_t A, int
               rank_check, gr_ctx_t ctx)
```

Similar to `gr_mat_lu()`, but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to *den*.

3.7.17 Solving

```
int gr_mat_nonsingular_solve_tril_classical(gr_mat_t X, const gr_mat_t L, const gr_mat_t
                                           B, int unit, gr_ctx_t ctx)
int gr_mat_nonsingular_solve_tril_recursive(gr_mat_t X, const gr_mat_t L, const gr_mat_t
                                           B, int unit, gr_ctx_t ctx)
int gr_mat_nonsingular_solve_tril_generic(gr_mat_t X, const gr_mat_t L, const gr_mat_t B,
                                          int unit, gr_ctx_t ctx)
int gr_mat_nonsingular_solve_tril(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int unit,
                                  gr_ctx_t ctx)
int gr_mat_nonsingular_solve_triu_classical(gr_mat_t X, const gr_mat_t U, const gr_mat_t
                                           B, int unit, gr_ctx_t ctx)
int gr_mat_nonsingular_solve_triu_recursive(gr_mat_t X, const gr_mat_t U, const gr_mat_t
                                           B, int unit, gr_ctx_t ctx)
int gr_mat_nonsingular_solve_triu_generic(gr_mat_t X, const gr_mat_t U, const gr_mat_t B,
                                          int unit, gr_ctx_t ctx)
int gr_mat_nonsingular_solve_triu(gr_mat_t X, const gr_mat_t U, const gr_mat_t B, int unit,
                                  gr_ctx_t ctx)
```

Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. Division by the diagonal entries must be possible; if not a division fails, `GR_DOMAIN` is returned even if the system is solvable. If *unit* is set, the main diagonal of L or U is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

```
int gr_mat_nonsingular_solve_fflu(gr_mat_t X, const gr_mat_t A, const gr_mat_t B, gr_ctx_t
                                ctx)
```

```
int gr_mat_nonsingular_solve_lu(gr_mat_t X, const gr_mat_t A, const gr_mat_t B, gr_ctx_t
                               ctx)
```

```
int gr_mat_nonsingular_solve(gr_mat_t X, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
    Solves  $AX = B$ . If  $A$  is not invertible, returns GR_DOMAIN even if the system has a solution.
```

```
int gr_mat_nonsingular_solve_fflu_precomp(gr_mat_t X, const slong *perm, const gr_mat_t LU,
                                          const gr_mat_t B, gr_ctx_t ctx)
```

```
int gr_mat_nonsingular_solve_lu_precomp(gr_mat_t X, const slong *perm, const gr_mat_t LU,
                                         const gr_mat_t B, gr_ctx_t ctx)
```

Solves $AX = B$ given a precomputed FFLU or LU factorization of A .

```
int gr_mat_nonsingular_solve_den_fflu(gr_mat_t X, gr_ptr den, const gr_mat_t A, const
                                      gr_mat_t B, gr_ctx_t ctx)
```

```
int gr_mat_nonsingular_solve_den(gr_mat_t X, gr_ptr den, const gr_mat_t A, const gr_mat_t
                                 B, gr_ctx_t ctx)
```

Solves $AX = B$ over the fraction field of the present ring (assumed to be an integral domain), returning X with an implied denominator den . If A is not invertible over the fraction field, returns `GR_DOMAIN` even if the system has a solution.

```
int gr_mat_solve_field(gr_mat_t X, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
```

Solves $AX = B$ where A is not necessarily square and not necessarily invertible. Assuming that the ring is a field, a return value of `GR_DOMAIN` indicates that the system has no solution. If there are multiple solutions, an arbitrary solution is returned.

3.7.18 Determinant and trace

```
int gr_mat_det_fflu(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_det_berkowitz(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_det_lu(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_det_cofactor(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_det_generic_field(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_det_generic_integral_domain(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_det_generic(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_det(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the determinant of the square matrix *mat*. Various algorithms are available:

- The *berkowitz* version uses the division-free Berkowitz algorithm performing $O(n^4)$ operations. Since no zero tests are required, it is guaranteed to succeed if the ring arithmetic succeeds.
- The *cofactor* version performs cofactor expansion. This is currently only supported for matrices up to size 4, and for larger matrices returns the `GR_UNABLE` flag.
- The *lu* and *fflu* versions use rational LU decomposition and fraction-free LU decomposition (Bareiss algorithm) respectively, requiring $O(n^3)$ operations. These algorithms can fail if zero certification or inversion fails, in which case the `GR_UNABLE` flag is returned.
- The *generic*, *generic_field* and *generic_integral_domain* versions choose an appropriate algorithm for a generic ring depending on the availability of division.
- The *default* method can be overloaded.

If the matrix is not square, `GR_DOMAIN` is returned.

```
int gr_mat_trace(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the trace (sum of entries on the main diagonal) of the square matrix *mat*. If the matrix is not square, GR_DOMAIN is returned.

3.7.19 Permanent

```
int gr_mat_permanent_cofactor(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_permanent_ryser(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_permanent_glynn(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_permanent_glynn_threaded(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_generic(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_permanent(gr_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

Sets *res* to the permanent of the square matrix *A*. The permanent $\text{perm}(A)$ is a polynomial expression in the entries of *A* equivalent to that for the determinant but with all subtractions replaced by additions. Several algorithms are implemented:

- The *cofactor* version uses recursive cofactor expansion, requiring $O(n!)$ additions and multiplications.
- The *ryser* version uses Ryser's formula [Rys1963] with Gray code traversal, requiring $O(2^n n)$ additions, subtractions and multiplications.
- The *glynn* version uses Glynn's formula [Gly2010] (actually equivalent to a method described earlier by Nijenhuis and Wilf [NW1978]). This requires about half as many operations as *ryser* but requires (exact) division by 2.
- The *glynn_threaded* version is a multithreaded implementation of Glynn's formula.

The *generic* method chooses cofactor expansion for $n \leq 4$ and otherwise chooses *ryser*, *glynn* or *glynn_threaded* depending on whether the ring supports division by 2. The default method can be overloaded.

3.7.20 Rank

```
int gr_mat_rank_fflu(slong *rank, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_rank_lu(slong *rank, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_rank(slong *rank, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the rank of *mat*. The default method returns GR_DOMAIN if the element ring is not an integral domain, in which case the usual rank is not well-defined. The *fflu* and *lu* variants currently do not check the element domain, and simply return this flag if they encounter an impossible inverse in the execution of the respective algorithms.

3.7.21 Row echelon form

```
int gr_mat_rref_lu(slong *rank, gr_mat_t R, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_rref_fflu(slong *rank, gr_mat_t R, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_rref(slong *rank, gr_mat_t R, const gr_mat_t A, gr_ctx_t ctx)
```

Sets *R* to the reduced row echelon form of *A*, also setting *rank* to its rank.

```
int gr_mat_rref_den_fflu(slong *rank, gr_mat_t R, gr_ptr den, const gr_mat_t A, gr_ctx_t ctx)
```

```
int gr_mat_rref_den(slong *rank, gr_mat_t R, gr_ptr den, const gr_mat_t A, gr_ctx_t ctx)
```

Like *rref*, but computes the reduced row echelon multiplied by a common (not necessarily minimal) denominator which is written to *den*. This can be used to compute the rref over an integral domain which is not a field.

3.7.22 Nullspace

int **gr_mat_nullspace**(*gr_mat_t* X, const *gr_mat_t* A, *gr_ctx_t* ctx)

Sets *X* to a basis for the (right) nullspace of *A*. On success, the output matrix will be resized to the correct number of columns.

The basis is not guaranteed to be presented in a canonical or minimal form.

If the ring is not a field, this is implied to compute a nullspace basis over the fraction field. The result may be meaningless if the ring is not an integral domain.

int **gr_mat_nullspace_from_rref**(*gr_mat_t* X, const *gr_mat_t* A, *gr_srcptr* Aden, *slong* rank, *gr_ctx_t* ctx)

Computes nullspace given the precomputed reduced row echelon form matrix *A* with rank *rank*. If *Aden* is not *NULL*, assume tha *A* has been multiplied by this common denominator as in the output of *gr_mat_rref_den()*.

int **gr_mat_nullspace_no_resize**(*slong* *nullity, *gr_mat_t* X, const *gr_mat_t* A, *gr_ctx_t* ctx)

Similar to *gr_mat_nullspace()*, but does not resize the matrix *X*, instead zero-padding if needed and returning the nullity (number of basis columns) in a separate variable. The user must supply a output matrix *X* with at least as many columns as the nullity.

3.7.23 Inverse and adjugate

int **gr_mat_inv**(*gr_mat_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Sets *res* to the inverse of *mat*, computed by solving $AA^{-1} = I$.

Returns GR_DOMAIN if it can be determined that *mat* is not invertible over the present ring (warning: this may not work over non-integral domains). If invertibility cannot be proved, returns GR_UNABLE.

To compute the inverse over the fraction field, one may use *gr_mat_nonsingular_solve_den()* or *gr_mat_adjugate()*.

int **gr_mat_adjugate_charpoly**(*gr_mat_t* adj, *gr_ptr* det, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int **gr_mat_adjugate_cofactor**(*gr_mat_t* adj, *gr_ptr* det, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int **gr_mat_adjugate**(*gr_mat_t* adj, *gr_ptr* det, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Sets *adj* to the adjugate matrix of *mat*, simultaneously setting *det* to the determinant of *mat*. We have $\text{adj}(A)A = A\text{adj}(A) = \det(A)I$, and $A^{-1} = \text{adj}(A)/\det(A)$ when *A* is invertible.

The *cofactor* version uses cofactor expansion, requiring the evaluation of n^2 determinants. The *charpoly* version computes and then evaluates the characteristic polynomial, requiring $O(n^{1/2})$ matrix multiplications plus $O(n^3)$ or $O(n^4)$ operations for the characteristic polynomial itself depending on the algorithm used.

3.7.24 Characteristic polynomial

int **_gr_mat_charpoly**(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int **gr_mat_charpoly**(*gr_poly_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Computes the characteristic polynomial using an algorithm choice which defaults to *_gr_mat_charpoly_generic()* but may be overridden by specific rings for performance. The underscore method assumes that *res* is a preallocated array of $n + 1$ coefficients.

int **_gr_mat_charpoly_generic**(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int **gr_mat_charpoly_generic**(*gr_poly_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Computes the characteristic polynomial using a generic algorithm choice.

int **_gr_mat_charpoly_berkowitz**(*gr_ptr* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

```
int gr_mat_charpoly_berkowitz(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the characteristic polynomial of the square matrix *mat*, computed using the division-free Berkowitz algorithm. The number of operations is $O(n^4)$ where n is the size of the matrix.

```
int _gr_mat_charpoly_danilevsky_inplace(gr_ptr res, gr_mat_t mat, gr_ctx_t ctx)
```

```
int _gr_mat_charpoly_danilevsky(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_charpoly_danilevsky(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int _gr_mat_charpoly_gauss(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_charpoly_gauss(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int _gr_mat_charpoly_householder(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_charpoly_householder(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the characteristic polynomial of the square matrix *mat*, computed using the Danilevsky algorithm, Hessenberg reduction using Gaussian elimination, and Hessenberg reduction using Householder reflections. The number of operations of each method is $O(n^3)$ where n is the size of the matrix. The *inplace* version overwrites the input matrix.

These methods require divisions and can therefore fail when the ring is not a field. They also require zero tests. The *householder* version also requires square roots. The flags `GR_UNABLE` or `GR_DOMAIN` are returned when an impossible division or square root is encountered or when a comparison cannot be performed.

```
int _gr_mat_charpoly_faddeev(gr_ptr res, gr_mat_t adj, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_charpoly_faddeev(gr_poly_t res, gr_mat_t adj, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int _gr_mat_charpoly_faddeev_bsgs(gr_ptr res, gr_mat_t adj, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_charpoly_faddeev_bsgs(gr_poly_t res, gr_mat_t adj, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the characteristic polynomial of the square matrix *mat*, computed using the Faddeev-LeVerrier algorithm. If the optional output argument *adj* is not `NULL`, it is set to the adjugate matrix, which is computed free of charge.

The *bsgs* version uses a baby-step giant-step strategy, also known as the Preparata-Sarwate algorithm. This reduces the complexity from $O(n^4)$ to $O(n^{3.5})$ operations at the cost of requiring $n^{0.5}$ temporary matrices to be stored.

This method requires divisions by small integers and can therefore fail (returning the `GR_UNABLE` or `GR_DOMAIN` flags) in finite characteristic or when the underlying ring does not implement a division algorithm.

```
int _gr_mat_charpoly_from_hessenberg(gr_ptr res, const gr_mat_t mat, gr_ctx_t ctx)
```

```
int gr_mat_charpoly_from_hessenberg(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
```

Sets *res* to the characteristic polynomial of the square matrix *mat*, which is assumed to be in Hessenberg form (this is currently not checked).

3.7.25 Minimal polynomial

```
int gr_mat_minpoly_field(gr_poly_t res, const gr_mat_t mat, gr_ctx_t ctx)
```

Compute the minimal polynomial of the matrix *mat*. The algorithm assumes that the coefficient ring is a field.

3.7.26 Companion matrix

```
int _gr_mat_companion(gr_mat_t res, gr_srcptr poly, gr_ctx_t ctx)
```

```
int gr_mat_companion(gr_mat_t res, const gr_poly_t poly, gr_ctx_t ctx)
```

Sets the n by n matrix *res* to the companion matrix of the polynomial *poly* which must have degree n . The underscore method reads $n + 1$ input coefficients. The algorithm assumes that the leading coefficient of *poly* is invertible.

```
int _gr_mat_companion_fraction(gr_mat_t res_num, gr_ptr res_den, gr_srcptr poly, gr_ctx_t ctx)
```

```
int gr_mat_companion_fraction(gr_mat_t res_num, gr_ptr res_den, const gr_poly_t poly, gr_ctx_t ctx)
```

Sets the n by n matrix *res_num* and the polynomial *res_den* so that the fraction is the companion matrix of the polynomial *poly* which must have degree n . The underscore method reads $n + 1$ input coefficients.

3.7.27 Similarity transformations

```
int gr_mat_apply_row_similarity(gr_mat_t M, slong r, gr_ptr d, gr_ctx_t ctx)
```

Applies an elementary similarity transform to the $n \times n$ matrix *M* in-place.

If *P* is the $n \times n$ identity matrix the zero entries of whose row *r* (0-indexed) have been replaced by *d*, this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

3.7.28 Eigenvalues

```
int gr_mat_eigenvalues(gr_vec_t lambda, gr_vec_t mult, const gr_mat_t mat, int flags, gr_ctx_t ctx)
```

```
int gr_mat_eigenvalues_other(gr_vec_t lambda, gr_vec_t mult, const gr_mat_t mat, gr_ctx_t mat_ctx, int flags, gr_ctx_t ctx)
```

Finds all eigenvalues of the given matrix in the ring defined by *ctx*, storing the eigenvalues without duplication in *lambda* (a vector with elements of type *ctx*) and the corresponding multiplicities in *mult* (a vector with elements of type *fmpz*).

The interface is essentially the same as that of *gr_poly_roots()*; see its documentation for details.

```
int gr_mat_diagonalization_precomp(gr_vec_t D, gr_mat_t L, gr_mat_t R, const gr_mat_t A, const gr_vec_t eigenvalues, const gr_vec_t mult, gr_ctx_t ctx)
```

```
int gr_mat_diagonalization_generic(gr_vec_t D, gr_mat_t L, gr_mat_t R, const gr_mat_t A, int flags, gr_ctx_t ctx)
```

```
int gr_mat_diagonalization(gr_vec_t D, gr_mat_t L, gr_mat_t R, const gr_mat_t A, int flags, gr_ctx_t ctx)
```

Computes a diagonalization $LAR = D$ given a square matrix *A*, where *D* is a diagonal matrix (returned as a vector) of the eigenvalues repeated according to their multiplicities, *L* is a matrix of left eigenvectors, and *R* is a matrix of right eigenvectors, normalized such that $L = R^{-1}$. This implies that $A = RDL = RDR^{-1}$. Either *L* or *R* (or both) can be set to NULL to omit computing the respective matrix.

If the matrix has entries in a field then a return flag of GR_DOMAIN indicates that the matrix is non-diagonalizable over this field.

The *precomp* version requires as input a precomputed set of eigenvalues with corresponding multiplicities, which can be computed with *gr_mat_eigenvalues()*.

3.7.29 Jordan decomposition

```
int gr_mat_set_jordan_blocks(gr_mat_t mat, const gr_vec_t lambda, slong num_blocks, slong
                           *block_lambda, slong *block_size, gr_ctx_t ctx)
int gr_mat_jordan_blocks(gr_vec_t lambda, slong *num_blocks, slong *block_lambda, slong
                       *block_size, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_jordan_transformation(gr_mat_t mat, const gr_vec_t lambda, slong num_blocks,
                                slong *block_lambda, slong *block_size, const gr_mat_t A,
                                gr_ctx_t ctx)
int gr_mat_jordan_form(gr_mat_t J, gr_mat_t P, const gr_mat_t A, gr_ctx_t ctx)
```

3.7.30 Matrix functions

```
int gr_mat_func_jordan(gr_mat_t res, const gr_mat_t A, gr_method_vec_op jet_func, gr_ctx_t
                      ctx)
int gr_mat_func_param_jordan(gr_mat_t res, const gr_mat_t A, gr_method_vec_scalar_op
                             jet_func, gr_srcptr c, gr_ctx_t ctx)
```

Computes the matrix function $f(A)$ using Jordan decomposition. The user supplies `int jet_func(gr_ptr r, gr_srcptr x, slong n, gr_ctx_t ctx)` which given a scalar x writes the jet $f(x), f'(x), \dots, f^{(n-1)}(x)/(n-1)!$ to the array r .

The *param* version takes as input a function with signature `int jet_func(gr_ptr r, gr_srcptr x, slong n, gr_srcptr c, gr_ctx_t ctx)` for evaluating a function $f(x, c)$ depending on an extra parameter c . Although c is nominally passed as a `gr_srcptr`, it can be a void pointer to arbitrary data that `jet_func` knows how to handle.

```
int gr_mat_exp_jordan(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_exp(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_log_jordan(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_log(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_pow_scalar_jordan(gr_mat_t res, const gr_mat_t A, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_pow_scalar(gr_mat_t res, const gr_mat_t A, gr_srcptr c, gr_ctx_t ctx)
int gr_mat_pow_fmpq_jordan(gr_mat_t res, const gr_mat_t mat, const fmpq_t exp, gr_ctx_t ctx)
int gr_mat_pow_fmpq(gr_mat_t res, const gr_mat_t mat, const fmpq_t exp, gr_ctx_t ctx)
```

Compute A^c using Jordan decomposition. The non-Jordan methods also check for small integer exponents and delegate those to the standard powering method.

```
int gr_mat_sqrt(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
int gr_mat_rsqrt(gr_mat_t res, const gr_mat_t A, gr_ctx_t ctx)
```

Compute a square root $A^{1/2}$ or a reciprocal square root $A^{-1/2}$. Currently the only implemented algorithm is the Jordan decomposition. Warning: this will often fail and return `GR_UNABLE` when the scalar type is not an algebraically closed field, even if the matrix is a perfect square.

3.7.31 Hessenberg form

truth_t **gr_mat_is_hessenberg**(const *gr_mat_t* mat, *gr_ctx_t* ctx)

Returns whether *mat* is in upper Hessenberg form.

int **gr_mat_hessenberg_gauss**(*gr_mat_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int **gr_mat_hessenberg_householder**(*gr_mat_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

int **gr_mat_hessenberg**(*gr_mat_t* res, const *gr_mat_t* mat, *gr_ctx_t* ctx)

Sets *res* to an upper Hessenberg form of *mat*. The *gauss* version uses Gaussian elimination. The *householder* version uses Householder reflections.

These methods require divisions and zero testing and can therefore fail (returning **GR_UNABLE** or **GR_DOMAIN**) when the ring is not a field. The *householder* version additionally requires complex conjugation and the ability to compute square roots.

3.7.32 Random matrices

int **gr_mat_randtest**(*gr_mat_t* res, *flint_rand_t* state, *gr_ctx_t* ctx)

Sets *res* to a random matrix. The distribution is nonuniform.

int **gr_mat_randops**(*gr_mat_t* mat, *flint_rand_t* state, *slong* count, *gr_ctx_t* ctx)

Randomises *mat* in-place by performing elementary row or column operations. More precisely, at most *count* random additions or subtractions of distinct rows and columns will be performed.

int **gr_mat_randpermdiag**(int *parity, *gr_mat_t* mat, *flint_rand_t* state, *gr_ptr* diag, *slong* n, *gr_ctx_t* ctx)

Sets *mat* to a random permutation of the diagonal matrix with *n* leading entries given by the vector *diag*. Returns **GR_DOMAIN** if the main diagonal of *mat* does not have room for at least *n* entries. The parity (0 or 1) of the permutation is written to *parity*.

int **gr_mat_randrank**(*gr_mat_t* mat, *flint_rand_t* state, *slong* rank, *gr_ctx_t* ctx)

Sets *mat* to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank. The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling *gr_mat_randops()*.

This operation only makes sense over integral domains (currently not checked).

int **gr_mat_randsimilar**(*gr_mat_t* mat, *flint_rand_t* state, *slong* opcount, *gr_ctx_t* ctx)

Randomises *mat* in-place by conjugating by elementary row/column operations. More precisely, at most *opcount* conjugations by random elementary row/column operations will be performed.

3.7.33 Orthogonal matrices

truth_t **gr_mat_is_orthogonal**(const *gr_mat_t* A, *gr_ctx_t* ctx)

Returns whether *A* is an orthogonal matrix (orthonormal matrix), i.e. a square matrix satisfying $AA^T = A^T A = I$. It is assumed (not checked) that the scalar ring is commutative.

truth_t **gr_mat_is_row_orthogonal**(const *gr_mat_t* A, *gr_ctx_t* ctx)

truth_t **gr_mat_is_col_orthogonal**(const *gr_mat_t* A, *gr_ctx_t* ctx)

truth_t **gr_mat_is_row_orthonormal**(const *gr_mat_t* A, *gr_ctx_t* ctx)

truth_t **gr_mat_is_col_orthonormal**(const *gr_mat_t* A, *gr_ctx_t* ctx)

Returns whether *A* is of the following type:

- Row-orthogonal: $AA^T = D$ for some diagonal matrix *D*.
- Column-orthogonal: $A^T A = D$ for some diagonal matrix *D*.
- Row-orthonormal: $AA^T = I$.

- Column-orthonormal: $A^T A = I$.

int **gr_mat_randtest_orthogonal**(*gr_mat_t* A, *flint_rand_t* state, *gr_ctx_t* ctx)

Generates a random orthogonal matrix. Uses Cayley’s construction, with a permutation matrix as a fallback. It is assumed (not checked) that the scalar ring is commutative. Fails with **GR_DOMAIN** if A is not square.

3.7.34 QR decomposition

int **gr_mat_lq_gso**(*gr_mat_t* L, *gr_mat_t* Q, const *gr_mat_t* A, *gr_ctx_t* ctx)

int **gr_mat_lq_recursive**(*gr_mat_t* L, *gr_mat_t* Q, const *gr_mat_t* A, *gr_ctx_t* ctx)

int **gr_mat_lq_generic**(*gr_mat_t* L, *gr_mat_t* Q, const *gr_mat_t* A, *gr_ctx_t* ctx)

int **gr_mat_lq**(*gr_mat_t* L, *gr_mat_t* Q, const *gr_mat_t* A, *gr_ctx_t* ctx)

int **gr_mat_qr**(*gr_mat_t* Q, *gr_mat_t* R, const *gr_mat_t* A, *gr_ctx_t* ctx)

Computes a QR or LQ decomposition. The $A = QR$ decomposition orthogonalizes the columns of A :

- A is $m \times n$ with $m \geq n$ and full rank (A has n linearly independent columns)
- Q is $m \times n$ and column-orthonormal
- R is $n \times n$ and upper triangular

The $A = LQ$ decomposition is the transposed operation, which orthogonalizes the rows of A :

- A is $m \times n$ with $m \leq n$ and full rank (A has m linearly independent rows)
- L is $m \times m$ and lower triangular
- Q is $m \times n$ and row-orthonormal

For rectangular matrices, these definitions correspond to the “reduced”, “compact” or “economy-sized” QR or LQ decomposition. To construct a “full” QR decomposition where Q is a square, orthogonal matrix, one can compute the reduced QR decomposition, extend Q with its orthogonal complement, and pad R or L with zeros.

The input matrix A should have elements in a subfield of $\mathbb{R}$ or real floating-point entries. These functions may succeed for other types, but the Euclidean norm normalizations will not necessarily be meaningful.

The *gso* algorithm uses Gram-Schmidt orthogonalization. The *recursive* algorithm uses block recursion. The QR decomposition is a simple wrapper around the LQ decomposition. We use the LQ decomposition internally as row operations are more efficient than column operations in the row-major format of *gr_mat_t*.

Aliasing of A and Q is handled efficiently in-place. Aliasing of A and R or L is allowed, but requires a temporary internal copy.

These methods return **GR_DOMAIN** if the matrix dimensions are not compatible, if A does not have full rank, or if the base ring does not support the necessary divisions or square roots to normalize vectors.

3.7.35 Special matrices

For the following functions, the user supplies an output matrix with the intended number of rows and columns.

int **gr_mat_ones**(*gr_mat_t* res, *gr_ctx_t* ctx)

Sets all entries in *res* to one.

int **gr_mat_pascal**(*gr_mat_t* res, int triangular, *gr_ctx_t* ctx)

Sets *res* to a Pascal matrix, whose entries are binomial coefficients. If *triangular* is 0, constructs a full symmetric matrix with the rows of Pascal's triangle as successive antidiagonals. If *triangular* is 1, constructs the upper triangular matrix with the rows of Pascal's triangle as columns, and if *triangular* is -1, constructs the lower triangular matrix with the rows of Pascal's triangle as rows.

int **gr_mat_stirling**(*gr_mat_t* res, int kind, *gr_ctx_t* ctx)

Sets *res* to a Stirling matrix, whose entries are Stirling numbers. If *kind* is 0, the entries are set to the unsigned Stirling numbers of the first kind. If *kind* is 1, the entries are set to the signed Stirling numbers of the first kind. If *kind* is 2, the entries are set to the Stirling numbers of the second kind.

int **gr_mat_hilbert**(*gr_mat_t* res, *gr_ctx_t* ctx)

Sets *res* to the Hilbert matrix, which has entries $1/(i + j + 1)$ for $i, j \geq 0$.

int **gr_mat_hadamard**(*gr_mat_t* res, *gr_ctx_t* ctx)

If possible, sets *res* to a Hadamard matrix of the provided size and returns **GR_SUCCESS**. Returns **GR_DOMAIN** if no Hadamard matrix of the given size exists, and **GR_UNABLE** if the implementation does not know how to construct a Hadamard matrix of the given size.

A Hadamard matrix of size n can only exist if n is 0, 1, 2, or a multiple of 4. It is not known whether a Hadamard matrix exists for every size that is a multiple of 4. This function uses the Paley construction, which succeeds for all n of the form $n = 2^e$ or $n = 2^e(q + 1)$ where q is an odd prime power. Orders n for which Hadamard matrices are known to exist but for which this construction fails are 92, 116, 156, ... (OEIS A046116).

3.7.36 Helper functions for reduction

int **gr_mat_reduce_row_generic**(*slong* *column, *gr_mat_t* A, *slong* *P, *slong* *L, *slong* m, *gr_ctx_t* ctx)

int **gr_mat_reduce_row**(*slong* *column, *gr_mat_t* A, *slong* *P, *slong* *L, *slong* m, *gr_ctx_t* ctx)

Reduce row n of the matrix A , assuming the prior rows are in Gauss form. However those rows may not be in order. The entry i of the array P is the row of A which has a pivot in the i -th column. If no such row exists, the entry of P will be -1 . The function sets *column* to the column in which the n -th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in P . Entry i of the array L contains the number of possibly nonzero columns of A row i . This speeds up reduction in the case that A is chambered on the right. Otherwise the entries of L can all be set to the number of columns of A . We require the entries of L to be monotonic increasing.

By default the *generic* version is called; specific rings can overload this (typically to implement delayed canonicalisation).

3.7.37 LLL

Let $A = (a_0, \dots, a_{n-1})$ be a set of linearly independent vectors over $\mathbb{R}$ with Gram-Schmidt orthogonalization $(b_0, \dots, b_{n-1})$ and Gram-Schmidt coefficients $\mu_{i,j} = \langle a_i, b_j \rangle / \|b_j\|^2$. The basis A is said to be LLL-reduced with parameter (δ, η) where $0.25 < \delta \leq 1$ and $0.5 \leq \eta < \sqrt{\delta}$ if they satisfy the size reduction condition

$$|\mu_{i,j}| \leq \eta, \quad 0 \leq j < i < n$$

and the Lovász condition

$$(\delta - \mu_{i,i-1}^2) \|b_{i-1}\|_2^2 \leq \|b_i\|_2^2, \quad 1 \leq i \leq n-1.$$

```
truth_t gr_mat_is_row_lll_reduced_naive(const gr_mat_t A, gr_srcptr delta, gr_srcptr eta,
                                         gr_ctx_t ctx)

truth_t gr_mat_is_row_lll_reduced_with_removal_naive(const gr_mat_t A, gr_srcptr delta,
                                                      gr_srcptr eta, gr_srcptr gs_B, slong
                                                      newd, gr_ctx_t ctx)
```

Check if the rows of A are LLL-reduced by naively performing the Gram-Schmidt orthogonalization and checking the conditions one row at a time.

In interval arithmetic, these functions terminate eagerly: `T_UNKNOWN` is returned if the tests are inconclusive for one row, even if there is a possibility that a later row could prove that the result should be `T_FALSE`.

3.7.38 Linear ODEs

```
int _gr_mat_gr_poly_solve_lode_newton_start(gr_mat_t Y, gr_mat_t Z, gr_poly_t
                                             A_denominator_inv, const gr_mat_t
                                             A_numerator, const gr_poly_t A_denominator,
                                             const gr_mat_t Y0, gr_ctx_t sol_poly_ctx)

int _gr_mat_gr_poly_solve_lode_newton_step(gr_mat_t Y, gr_mat_t Z, gr_poly_t
                                             A_denominator_inv, slong *len, const gr_mat_t
                                             A_numerator, const gr_poly_t A_denominator, int
                                             A_is_companion, gr_ctx_t sol_poly_ctx)

int _gr_mat_gr_poly_solve_lode_newton(gr_mat_t Y, gr_mat_t Z, const gr_mat_t A_numerator,
                                       const gr_poly_t A_denominator, const gr_mat_t Y0,
                                       slong len, gr_ctx_t A_poly_ctx, gr_ctx_t sol_poly_ctx)

int gr_mat_gr_poly_solve_lode_newton(gr_mat_t Y, const gr_mat_t A_numerator, const
                                       gr_poly_t A_denominator, const gr_mat_t Y0, slong len,
                                       gr_ctx_t A_poly_ctx, gr_ctx_t sol_poly_ctx)
```

Solves the system of linear ordinary differential equations $Y'(t) = A(t)Y(t)$ with rational function coefficients $A(t)$ and initial condition $Y(0) = Y_0$. Sets Y to the power series solution truncated to length len . The algorithm is a Newton iteration as described in [BCOSSS2007], but using an iteration for the power series inverse of $A\_denominator$, and using a specialized implementation of multiplication by $A(t) \bmod t^m$ in the case where $A(t)$ is a companion matrix. An optimal sequence of precisions is used to reach the target len , even when it is not a power of two.

3.7.39 Test functions

The following functions run *iters* test iterations, generating matrices up to size *maxn*. If *ctx* is set to NULL, a random ring is generated on each test iteration, otherwise the given ring is tested.

```
void gr_mat_test_mul(gr_method_mat_binary_op mul_impl, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `mul_impl` for correctness as an implementation of `gr_mat_mul()`.

```
void gr_mat_test_lu(gr_method_mat_lu_op lu_impl, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `mul_impl` for correctness as an implementation of `gr_mat_lu()`.

```
void gr_mat_test_det(gr_method_mat_unary_op_get_scalar det_impl, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `det_impl` for correctness as an implementation of `gr_mat_det()`.

```
void gr_mat_test_charpoly(gr_method_mat_unary_op_get_scalar charpoly_impl, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `charpoly_impl` for correctness as an implementation of `_gr_mat_charpoly()`.

```
void gr_mat_test_nonsingular_solve_tril(gr_method_mat_binary_op_with_flag solve_impl, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

```
void gr_mat_test_nonsingular_solve_triu(gr_method_mat_binary_op_with_flag solve_impl, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `solve_impl` for correctness as an implementation of `gr_mat_nonsingular_solve_tril()` / `gr_mat_nonsingular_solve_triu()`.

```
void gr_mat_test_approx_mul_max_norm(gr_method_mat_binary_op mul_impl, gr_srcptr rel_tol, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given implementation of matrix multiplication for accuracy over an approximate numerical ring by checking that $|C - AB| \leq |A||B|rel\_tol$ holds in the max norm, using classical multiplication for reference.

```
void gr_mat_test_approx_mul_pos_entrywise_accurate(gr_method_mat_binary_op mul_impl, gr_srcptr rel_tol, flint_rand_t state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given implementation of matrix multiplication for accuracy over an approximate numerical ring by generating nonnegative matrices and checking that the entrywise relative error compared to classical multiplication does not exceed *rel_tol*.

3.8 `gr_poly.h` – dense univariate polynomials over generic rings

A `gr_poly_t` represents a univariate polynomial $f \in R[X]$ implemented as a dense array of coefficients in a generic ring R .

In this module, the context object `ctx` always represents the coefficient ring R unless otherwise stated. Creating a context object representing the polynomial ring $R[X]$ only becomes necessary when one wants to manipulate polynomials using generic ring methods like `gr_add` instead of the designated polynomial methods like `gr_poly_add`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (often requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

3.8.1 Supported coefficient domains

Some methods in this module implicitly assume that R is a commutative ring or an approximate (e.g. floating-point) commutative ring. When used with a more general R , they may output nonsense without returning the appropriate `GR_DOMAIN` or `GR_UNABLE` flags. Better support for noncommutative coefficients is planned for the future.

Some methods make stronger implicit assumptions, for example that R is an integral domain or a field. Such assumptions are documented on a case by case basis.

3.8.2 Type compatibility

The `gr_poly` type has the same data layout as the following polynomial types: `fmpz_poly`, `fq_poly`, `fq_nmod_poly`, `fq_zech_poly`, `arb_poly`, `acb_poly`, `ca_poly`. Methods in this module can therefore be mixed freely with methods in the corresponding FLINT modules when the underlying coefficient type is the same. It is not directly compatible with the following types: `fmpq_poly` (coefficients are stored with a common denominator), `nmod_poly` (modulus data is stored as part of the polynomial object).

3.8.3 Weak normalization

A `gr_poly_t` is always normalised by removing leading zeros. For rings without decidable equality (e.g. rings with inexact representation), only coefficients that are provably zero will be removed, and there can thus be spurious leading zeros in the internal representation. Methods that depend on knowing the exact degree of a polynomial will act appropriately, typically by returning `GR_UNABLE` when it is unknown whether the leading stored coefficient is nonzero.

3.8.4 Types, macros and constants

type `gr_poly_struct`

type `gr_poly_t`

Contains a pointer to an array of coefficients (`coeffs`), the used length (`length`), and the allocated size of the array (`alloc`).

A `gr_poly_t` is defined as an array of length one of type `gr_poly_struct`, permitting a `gr_poly_t` to be passed by reference.

3.8.5 Memory management

```
void gr_poly_init(gr_poly_t poly, gr_ctx_t ctx)
void gr_poly_init2(gr_poly_t poly, slong len, gr_ctx_t ctx)
void gr_poly_clear(gr_poly_t poly, gr_ctx_t ctx)

gr_ptr gr_poly_coeff_ptr(gr_poly_t poly, slong i, gr_ctx_t ctx)
gr_srcptr gr_poly_coeff_srcptr(const gr_poly_t poly, slong i, gr_ctx_t ctx)

gr_ptr gr_poly_entry_ptr(gr_poly_t poly, slong i, gr_ctx_t ctx)
gr_srcptr gr_poly_entry_srcptr(const gr_poly_t poly, slong i, gr_ctx_t ctx)
    These functions are deprecated aliases of gr_poly_coeff_ptr() and gr_poly_coeff_srcptr();
    use those functions instead.

slong gr_poly_length(const gr_poly_t poly, gr_ctx_t ctx)
void gr_poly_swap(gr_poly_t poly1, gr_poly_t poly2, gr_ctx_t ctx)
void gr_poly_fit_length(gr_poly_t poly, slong len, gr_ctx_t ctx)
void _gr_poly_set_length(gr_poly_t poly, slong len, gr_ctx_t ctx)
```

3.8.6 Basic manipulation

```
void _gr_poly_normalise(gr_poly_t poly, gr_ctx_t ctx)

int gr_poly_set(gr_poly_t res, const gr_poly_t src, gr_ctx_t ctx)
int gr_poly_get_fmpz_poly(gr_poly_t res, const fmpz_poly_t src, gr_ctx_t ctx)
int gr_poly_set_fmpz_poly(gr_poly_t res, const fmpz_poly_t src, gr_ctx_t ctx)
int gr_poly_set_gr_poly_other(gr_poly_t res, const gr_poly_t x, gr_ctx_t x_ctx, gr_ctx_t ctx)

int _gr_poly_reverse(gr_ptr res, gr_srcptr poly, slong len, slong n, gr_ctx_t ctx)
int gr_poly_reverse(gr_poly_t res, const gr_poly_t poly, slong n, gr_ctx_t ctx)

int gr_poly_truncate(gr_poly_t res, const gr_poly_t poly, slong newlen, gr_ctx_t ctx)

int gr_poly_zero(gr_poly_t poly, gr_ctx_t ctx)
int gr_poly_one(gr_poly_t poly, gr_ctx_t ctx)
int gr_poly_neg_one(gr_poly_t poly, gr_ctx_t ctx)
int gr_poly_gen(gr_poly_t poly, gr_ctx_t ctx)

int gr_poly_write(gr_stream_t out, const gr_poly_t poly, const char *x, gr_ctx_t ctx)
int _gr_poly_write(gr_stream_t out, gr_srcptr poly, slong n, const char *x, gr_ctx_t ctx)
int _gr_poly_get_str(char **res, gr_srcptr f, slong len, const char *x, gr_ctx_t ctx)
int gr_poly_get_str(char **res, const gr_poly_t f, const char *x, gr_ctx_t ctx)
int gr_poly_print(const gr_poly_t poly, gr_ctx_t ctx)

int _gr_poly_set_str(gr_ptr res, const char *s, const char *x, slong len, gr_ctx_t ctx)
int gr_poly_set_str(gr_poly_t res, const char *s, const char *x, gr_ctx_t ctx)
    Parse polynomial from an expression string, assuming that the string in x gives the name of the
    generator. The underscore method zero-pads the result if the length of the parsed polynomial
    is shorter than len, and returns GR_UNABLE if the length of the parsed polynomial exceeds len.
    Intermediate terms are allowed to be longer than len.

    Warning: these methods are not currently optimized for polynomials of high degree and may run
    with quadratic complexity.
```

```

int gr_poly_randtest(gr_poly_t poly, flint_rand_t state, slong len, gr_ctx_t ctx)

truth_t _gr_poly_equal(gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
truth_t gr_poly_equal(const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)

truth_t gr_poly_is_zero(const gr_poly_t poly, gr_ctx_t ctx)
truth_t gr_poly_is_one(const gr_poly_t poly, gr_ctx_t ctx)
truth_t gr_poly_is_gen(const gr_poly_t poly, gr_ctx_t ctx)
truth_t gr_poly_is_scalar(const gr_poly_t poly, gr_ctx_t ctx)

int gr_poly_set_scalar(gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
int gr_poly_set_si(gr_poly_t poly, slong c, gr_ctx_t ctx)
int gr_poly_set_ui(gr_poly_t poly, ulong c, gr_ctx_t ctx)
int gr_poly_set_fmpz(gr_poly_t poly, const fmpz_t c, gr_ctx_t ctx)
int gr_poly_set_fmpq(gr_poly_t poly, const fmpq_t c, gr_ctx_t ctx)

int gr_poly_set_coeff_scalar(gr_poly_t poly, slong n, gr_srcptr c, gr_ctx_t ctx)
int gr_poly_set_coeff_si(gr_poly_t poly, slong n, slong c, gr_ctx_t ctx)
int gr_poly_set_coeff_ui(gr_poly_t poly, slong n, ulong c, gr_ctx_t ctx)
int gr_poly_set_coeff_fmpz(gr_poly_t poly, slong n, const fmpz_t c, gr_ctx_t ctx)
int gr_poly_set_coeff_fmpq(gr_poly_t poly, slong n, const fmpq_t c, gr_ctx_t ctx)

int gr_poly_get_coeff_scalar(gr_ptr res, const gr_poly_t poly, slong n, gr_ctx_t ctx)

```

3.8.7 Arithmetic

```

int gr_poly_neg(gr_poly_t res, const gr_poly_t src, gr_ctx_t ctx)

int _gr_poly_add(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
int gr_poly_add(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)

int _gr_poly_sub(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
int gr_poly_sub(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)

int _gr_poly_mul(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t ctx)
int gr_poly_mul(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)

int _gr_poly_mullow_generic(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
                           slong len, gr_ctx_t ctx)
int _gr_poly_mullow(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, slong len,
                    gr_ctx_t ctx)
int gr_poly_mullow(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, slong len, gr_ctx_t
                  ctx)

```

Note: multiplication and low multiplication call `_gr_poly_mullow_generic()` by default, which currently always delegates to `_gr_poly_mullow_classical()`. This can be overridden by specific rings.

```

int _gr_poly_mulmid_generic(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
                           slong nlo, slong nhi, gr_ctx_t ctx)
int _gr_poly_mulmid(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, slong nlo,
                    slong nhi, gr_ctx_t ctx)
int gr_poly_mulmid(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, slong nlo, slong
                  nhi, gr_ctx_t ctx)

```


3.8.8 Multiplication algorithms

```
int _gr_poly_mulmid_classical(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,  
                             slong nlo, slong nhi, gr_ctx_t ctx)
```

```
int gr_poly_mulmid_classical(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,  
                             slong nlo, slong nhi, gr_ctx_t ctx)
```

```
int _gr_poly_mulalow_classical(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,  
                               slong n, gr_ctx_t ctx)
```

```
int gr_poly_mulalow_classical(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, slong  
                              n, gr_ctx_t ctx)
```

Multiply using the classical (schoolbook) algorithm, performing a sequence of dot products.

```
int _gr_poly_mulmid_bivariate_KS(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong  
                                 len2, slong nlo, slong nhi, gr_ctx_t ctx)
```

```
int gr_poly_mulmid_bivariate_KS(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong  
                                len2, slong nlo, slong nhi, gr_ctx_t ctx)
```

```
int _gr_poly_mulalow_bivariate_KS(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong  
                                  len2, slong n, gr_ctx_t ctx)
```

```
int gr_poly_mulalow_bivariate_KS(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,  
                                 slong n, gr_ctx_t ctx)
```

Assuming that the coefficients are polynomials of type `gr_poly`, reduce the bivariate product to a single univariate product using Kronecker substitution. Returns `GR_UNABLE` if the elements are not of type `gr_poly`.

```
int _gr_poly_mulmid_complex_reorder(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,  
                                    slong len2, slong nlo, slong nhi, int karatsuba, gr_ctx_t ctx,  
                                    gr_ctx_t real_ctx)
```

```
int gr_poly_mulmid_complex_reorder(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,  
                                   slong nlo, slong nhi, int karatsuba, gr_ctx_t ctx, gr_ctx_t  
                                   real_ctx)
```

```
int _gr_poly_mulalow_complex_reorder(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,  
                                     slong len2, slong n, int karatsuba, gr_ctx_t ctx, gr_ctx_t  
                                     real_ctx)
```

```
int gr_poly_mulalow_complex_reorder(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,  
                                    slong n, int karatsuba, gr_ctx_t ctx, gr_ctx_t real_ctx)
```

Assuming that the coefficients of the polynomials of type `ctx` are complex numbers represented in Cartesian form as pairs of elements of type `real_ctx`, multiply by reordering to obtain a set of real polynomial products.

If at least one polynomial is detected to be purely real or imaginary, one or two real multiplications are used. Otherwise, four real multiplications are used unless the *karatsuba* flag is set in which three multiplications are used. When squaring, two real squarings and one real multiplication are used unless the *karatsuba* flag is set in which case three real squarings are used.

```
int _gr_poly_mul_karatsuba(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,  
                           gr_ctx_t ctx)
```

```
int gr_poly_mul_karatsuba(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t  
                          ctx)
```

Karatsuba multiplication. Not optimized for unbalanced operands, and not memory-optimized for recursive calls. The underscore method requires positive lengths and does not support aliasing. This function calls `_gr_poly_mul()` recursively rather than itself, so to get a recursive algorithm with $O(n^{1.6})$ complexity, the ring must overload `_gr_poly_mul()` to dispatch to `_gr_poly_mul_karatsuba()` above some cutoff.

```
int _gr_poly_mul_toom33(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,  
                        gr_ctx_t ctx);
```



```
int gr_poly_mul_toom33(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx);
```

Balanced Toom-3 multiplication with interpolation in five points, using the Bodrato evaluation scheme. Assumes commutativity and that the ring supports exact division by 2 and 3. Not optimized for squaring. The underscore method requires positive lengths and does not support aliasing. This function calls `_gr_poly_mul()` recursively rather than itself, so to get a recursive algorithm with $O(n^{1.465})$ complexity, the ring must overload `_gr_poly_mul()` to dispatch to `_gr_poly_mul_toom33()` above some cutoff.

```
int _gr_poly_mullow_toom_serial(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong
                               len2, slong trunc, slong num_points, slong splitting_degree,
                               gr_ctx_t ctx)
```

```
int gr_poly_mullow_toom_serial(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,
                               slong n, slong num_points, slong splitting_degree, gr_ctx_t ctx)
```

Toom-Cook multiplication of arbitrary order: the inputs are segmented as polynomials in $t = x^m$ for some splitting degree $m \geq 1$ and the product is computed by evaluation and interpolation at $r \geq 1$ distinct values of t .

For convenience, the user can specify either the number of interpolation points r (`num_points`) or the splitting degree m (`splitting_degree`). The user must provide exactly one of these parameters, passing zero for the other; the optimal value of the omitted parameter will be computed automatically. For example, if `len1` and `len2` are roughly balanced, then providing either $r = 3, m = 0$ or $r = 0, m = \lceil \max(\text{len1}, \text{len2})/2 \rceil$ will result in Karatsuba multiplication with evaluation in 0, 1 and ∞ . For balanced operands, an odd number of evaluation points $r = 2k - 1$ corresponds to splitting both inputs into k segments; even values of d typically only make sense for unbalanced products.

The evaluation and interpolation points are chosen as $0, 1, -1, 2, -2, 3, -3, \dots, \infty$. Evaluation and interpolation is implicitly done by multiplying by the corresponding Vandermonde matrix and its inverse. In too small finite characteristic where the Vandermonde matrix cannot be inverted, `GR_UNABLE` is returned. In characteristic zero, the interpolation matrix is computed with a common denominator and this denominator is removed from the final result with an exact division.

The suffix “serial” indicates that the evaluations are done one point at a time and that the output is constructed incrementally with the goal to minimize memory allocation. This implementation is therefore well suited for huge memory-constrained multiplications, in contrast to a typical Toom-Cook implementation which will use several extra temporary buffers to minimize scalar operations. In total this implementation requires $4m$ elements of temporary storage for intermediate polynomials and $2r^2$ ring elements or integers for the evaluation and interpolation matrices.

The underscore method requires positive lengths and does not support aliasing.

3.8.9 Scalar arithmetic

```
int gr_poly_add_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
```

```
int gr_poly_add_ui(gr_poly_t res, const gr_poly_t poly, ulong c, gr_ctx_t ctx)
```

```
int gr_poly_add_si(gr_poly_t res, const gr_poly_t poly, slong c, gr_ctx_t ctx)
```

```
int gr_poly_add_fmpz(gr_poly_t res, const gr_poly_t poly, const fmpz c, gr_ctx_t ctx)
```

```
int gr_poly_add_fmpq(gr_poly_t res, const gr_poly_t poly, const fmpq c, gr_ctx_t ctx)
```

Sets `res` to `poly` plus the scalar `c` which must be an element of or coercible to the coefficient ring.

```
int gr_poly_sub_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
```

```
int gr_poly_sub_ui(gr_poly_t res, const gr_poly_t poly, ulong c, gr_ctx_t ctx)
```

```
int gr_poly_sub_si(gr_poly_t res, const gr_poly_t poly, slong c, gr_ctx_t ctx)
```

```
int gr_poly_sub_fmpz(gr_poly_t res, const gr_poly_t poly, const fmpz c, gr_ctx_t ctx)
```

```
int gr_poly_sub_fmpq(gr_poly_t res, const gr_poly_t poly, const fmpq c, gr_ctx_t ctx)
```

Sets `res` to `poly` minus the scalar `c` which must be an element of or coercible to the coefficient ring.

```
int gr_poly_mul_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
```

```
int gr_poly_scalar_mul(gr_poly_t res, gr_srcptr c, const gr_poly_t poly, gr_ctx_t ctx)
int gr_poly_mul_ui(gr_poly_t res, const gr_poly_t poly, ulong c, gr_ctx_t ctx)
int gr_poly_mul_si(gr_poly_t res, const gr_poly_t poly, slong c, gr_ctx_t ctx)
int gr_poly_mul_fmpz(gr_poly_t res, const gr_poly_t poly, const fmpz_t c, gr_ctx_t ctx)
int gr_poly_mul_fmpq(gr_poly_t res, const gr_poly_t poly, const fmpq_t c, gr_ctx_t ctx)
    Sets res to poly multiplied by the scalar c (or the scalar c multiplied by poly) which must be an
    element of or coercible to the coefficient ring.

int gr_poly_addmul_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
    Adds poly multiplied by the scalar c to res.

int gr_poly_submul_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
    Subtracts poly multiplied by the scalar c from res.
```

3.8.10 Powering

```
int _gr_poly_pow_series_ui_binexp(gr_ptr res, gr_srcptr f, slong flen, ulong exp, slong len,
                                   gr_ctx_t ctx)
int gr_poly_pow_series_ui_binexp(gr_poly_t res, const gr_poly_t poly, ulong exp, slong len,
                                   gr_ctx_t ctx)

int _gr_poly_pow_series_ui(gr_ptr res, gr_srcptr f, slong flen, ulong exp, slong len, gr_ctx_t ctx)
int gr_poly_pow_series_ui(gr_poly_t res, const gr_poly_t poly, ulong exp, slong len, gr_ctx_t ctx)

int _gr_poly_pow_ui_binexp(gr_ptr res, gr_srcptr f, slong flen, ulong exp, gr_ctx_t ctx)
int gr_poly_pow_ui_binexp(gr_poly_t res, const gr_poly_t poly, ulong exp, gr_ctx_t ctx)

int _gr_poly_pow_ui(gr_ptr res, gr_srcptr f, slong flen, ulong exp, gr_ctx_t ctx)
int gr_poly_pow_ui(gr_poly_t res, const gr_poly_t poly, ulong exp, gr_ctx_t ctx)

int gr_poly_pow_fmpz(gr_poly_t res, const gr_poly_t poly, const fmpz_t exp, gr_ctx_t ctx)

int _gr_poly_pow_series_fmpq_recurrence(gr_ptr h, gr_srcptr f, slong flen, const fmpq_t exp,
                                         slong len, int precomp, gr_ctx_t ctx)
int gr_poly_pow_series_fmpq_recurrence(gr_poly_t res, const gr_poly_t poly, const fmpq_t exp,
                                         slong len, gr_ctx_t ctx)
```

3.8.11 Shifting

```
int _gr_poly_shift_left(gr_ptr res, gr_srcptr poly, slong len, slong n, gr_ctx_t ctx)
int gr_poly_shift_left(gr_poly_t res, const gr_poly_t poly, slong n, gr_ctx_t ctx)

int _gr_poly_shift_right(gr_ptr res, gr_srcptr poly, slong len, slong n, gr_ctx_t ctx)
int gr_poly_shift_right(gr_poly_t res, const gr_poly_t poly, slong n, gr_ctx_t ctx)
```

3.8.12 Scalar division

```
int gr_poly_div_scalar(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
```

3.8.13 Division with remainder

```
int _gr_poly_divrem_divconquer_preinv1(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr
                                     B, slong lenB, gr_srcptr invB, slong cutoff, gr_ctx_t ctx)
int _gr_poly_divrem_divconquer_noinv(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B,
                                     slong lenB, slong cutoff, gr_ctx_t ctx)
int _gr_poly_divrem_divconquer(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong
                                lenB, slong cutoff, gr_ctx_t ctx)
int gr_poly_divrem_divconquer(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B,
                              slong cutoff, gr_ctx_t ctx)
int _gr_poly_divrem_basecase_preinv1(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B,
                                     slong lenB, gr_srcptr invB, gr_ctx_t ctx)
int _gr_poly_divrem_basecase_noinv(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B,
                                   slong lenB, gr_ctx_t ctx)
int _gr_poly_divrem_basecase(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong
                              lenB, gr_ctx_t ctx)
int gr_poly_divrem_basecase(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B,
                            gr_ctx_t ctx)
int _gr_poly_divrem_newton(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                           gr_ctx_t ctx)
int gr_poly_divrem_newton(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B,
                          gr_ctx_t ctx)
int _gr_poly_divrem(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                   gr_ctx_t ctx)
int gr_poly_divrem(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
```

These functions implement Euclidean division with remainder: given polynomials $A, B \in K[x]$ where K is a field, with $B \neq 0$, there is a unique quotient Q and remainder R such that $A = BQ + R$ and either $R = 0$ or $\deg(R) < \deg(B)$. If B is provably zero, `GR_DOMAIN` is returned.

When K is a commutative ring and $\text{lc}(B)$ is a unit in K , the situation is the same as over fields. In particular, Euclidean division with remainder always makes sense over commutative rings when B is monic. If $\text{lc}(B)$ is not a unit, the division still makes sense if the coefficient quotient $\text{lc}(r) / \text{lc}(B)$ exists for each partial remainder r . Indeed, the *basecase* and *divconquer* algorithms return `GR_DOMAIN` precisely when encountering a leading quotient $\text{lc}(r) / \text{lc}(B) \notin K$. However, the *newton* algorithm as currently implemented returns `GR_DOMAIN` when $\text{lc}(B)^{-1} \notin K$.

The underscore methods make the following assumptions:

- Q has room for `lenA - lenB + 1` coefficients.
- R has room for `lenB - 1` coefficients.
- `lenA >= lenB >= 1`.
- Q is not aliased with either A or B .
- R is not aliased with B .
- R may be aliased with A , in which case all `lenA` entries may be used as scratch space. Note that in this case, only the low `lenB - 1` coefficients of R actually represent valid coefficients on output: the higher scratch coefficients will not necessarily be zeroed.
- The divisor B is normalized to have nonzero leading coefficient. (The non-underscore methods check for leading coefficients that are not provably nonzero and return `GR_UNABLE`.)

The *preinv1* functions take a precomputed inverse of the leading coefficient as input. The *noinv* versions perform repeated checked divisions by the leading coefficient.

```
int _gr_poly_div_divconquer_preinv1(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                                   gr_srcptr invB, slong cutoff, gr_ctx_t ctx)
int _gr_poly_div_divconquer_noinv(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                                  slong cutoff, gr_ctx_t ctx)
int _gr_poly_div_divconquer(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, slong
                           cutoff, gr_ctx_t ctx)
int gr_poly_div_divconquer(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, slong cutoff,
                          gr_ctx_t ctx)
int _gr_poly_div_basecase_preinv1(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                                  gr_srcptr invB, gr_ctx_t ctx)
int _gr_poly_div_basecase_noinv(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                                gr_ctx_t ctx)
int _gr_poly_div_basecase(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t
                          ctx)
int gr_poly_div_basecase(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_div_newton(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int gr_poly_div_newton(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_div(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int gr_poly_div(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
```

Versions of the *divrem* functions which output only the quotient. These are generally faster.

```
int _gr_poly_rem(gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t ctx)
int gr_poly_rem(gr_poly_t R, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
```

Versions of the *divrem* functions which output only the remainder.

3.8.14 Division with remainder with full precomputed inverse

```
int _gr_poly_div_newton_n_preinv(gr_ptr Q, gr_srcptr A, slong lenA, gr_srcptr
                                FLINT_UNUSED(B), slong lenB, gr_srcptr Binv, slong
                                lenBinv, gr_ctx_t ctx)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but return only Q .

We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
int gr_poly_div_newton_n_preinv(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, const
                              gr_poly_t Binv, gr_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q .

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \cdot \text{the length of } B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
int _gr_poly_divrem_newton_n_preinv(gr_ptr Q, gr_ptr R, gr_srcptr A, slong lenA, gr_srcptr B,
                                   slong lenB, gr_srcptr Binv, slong lenBinv, gr_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$. The algorithm used is to call `_gr_poly_div_newton_n_preinv()` and then multiply out and compute the remainder.

```
int gr_poly_divrem_newton_n_preinv(gr_poly_t Q, gr_poly_t R, const gr_poly_t A, const
                                   gr_poly_t B, const gr_poly_t Binv, gr_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required (not checked) that the length of A is less than or equal to $2 \times \text{the length of } B - 2$.

3.8.15 Power series division

For divide-and-conquer (including Newton-like) algorithms, *cutoff* has the following meaning: we use the basecase algorithm for lengths $n < \text{cutoff}$ and the divide-and-conquer algorithm for $n \geq \text{cutoff}$. Using $\text{cutoff} = n$ thus results in exactly one divide-and-conquer step with a basecase length of $\lceil n/2 \rceil$. One should **avoid** calling the Newton methods with $n < \text{cutoff}$ as this may result in much worse performance if those methods do not have a specific escape check for that case.

The *newton* versions uses Newton iteration, switching to a basecase algorithm when the length is smaller than the specified *cutoff*. Division uses the Karp-Markstein algorithm.

```
int _gr_poly_inv_series_newton(gr_ptr res, gr_srcptr A, slong Alen, slong len, slong cutoff,
                              gr_ctx_t ctx)
```

```
int gr_poly_inv_series_newton(gr_poly_t res, const gr_poly_t A, slong len, slong cutoff, gr_ctx_t
                              ctx)
```

```
int _gr_poly_inv_series_basecase_preinv1(gr_ptr res, gr_srcptr A, slong Alen, gr_srcptr Ainv,
                                         slong len, gr_ctx_t ctx)
```

```
int _gr_poly_inv_series_basecase(gr_ptr res, gr_srcptr A, slong Alen, slong len, gr_ctx_t ctx)
```

```
int gr_poly_inv_series_basecase(gr_poly_t res, const gr_poly_t A, slong len, gr_ctx_t ctx)
```

```
int _gr_poly_inv_series(gr_ptr res, gr_srcptr A, slong Alen, slong len, gr_ctx_t ctx)
```

```
int gr_poly_inv_series(gr_poly_t res, const gr_poly_t A, slong len, gr_ctx_t ctx)
```

```
int _gr_poly_div_series_newton(gr_ptr res, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen, slong
                              len, slong cutoff, gr_ctx_t ctx)
```

```
int gr_poly_div_series_newton(gr_poly_t res, const gr_poly_t A, const gr_poly_t B, slong len,
                              slong cutoff, gr_ctx_t ctx)
```

```
int _gr_poly_div_series_divconquer(gr_ptr res, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen,
                                   slong len, slong cutoff, gr_ctx_t ctx)
```

```
int gr_poly_div_series_divconquer(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, slong len,
                                   slong cutoff, gr_ctx_t ctx)
```

```
int _gr_poly_div_series_invmul(gr_ptr res, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen, slong
                              len, gr_ctx_t ctx)
```

```
int gr_poly_div_series_invmul(gr_poly_t res, const gr_poly_t A, const gr_poly_t B, slong len,
                              gr_ctx_t ctx)
```

```
int _gr_poly_div_series_basecase_preinv1(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong
                                         Blen, gr_srcptr Binv, slong len, gr_ctx_t ctx)
```

```
int _gr_poly_div_series_basecase_noinv(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong
                                        Blen, slong len, gr_ctx_t ctx)
```

```
int _gr_poly_div_series_basecase(gr_ptr res, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen,
                                   slong len, gr_ctx_t ctx)
```

```
int gr_poly_div_series_basecase(gr_poly_t res, const gr_poly_t A, const gr_poly_t B, slong len,
                                gr_ctx_t ctx)
int _gr_poly_div_series(gr_ptr res, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen, slong len,
                        gr_ctx_t ctx)
int gr_poly_div_series(gr_poly_t res, const gr_poly_t A, const gr_poly_t B, slong len, gr_ctx_t
                        ctx)
```

3.8.16 Exact division

These functions compute a quotient $Q = A/B$ which is known to be exact (without remainder) in $R[x]$ (or in $R[[x]]/x^n$ in the case of series division). Given a nonexact division, they are allowed to set Q to an arbitrary polynomial and return `GR_SUCCESS` instead of returning an error flag.

R is assumed to be an integral domain (this is not checked).

For exact division, we have the choice of starting the division from the most significant terms (classical division) or the least significant (power series division). Which direction is more efficient depends in part on whether the leading or trailing coefficient of B is cheaper to use for divisions. In a generic setting, this is hard to predict.

The *bidirectional* algorithms combine two half-divisions from both ends. This halves the number of operations in the basecase regime, though an extra coefficient inversion may be needed.

The `noinv` versions perform repeated `divexact` operations in the scalar domain without attempting to invert the leading (or trailing) coefficient, while other versions check invertibility first. There are no `divexact_preinv1` versions because those are identical to the `div_preinv1` counterparts.

```
int _gr_poly_divexact_basecase_bidirectional(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B,
                                             slong Blen, gr_ctx_t ctx)
int gr_poly_divexact_basecase_bidirectional(gr_poly_t Q, const gr_poly_t A, const gr_poly_t
                                             B, gr_ctx_t ctx)
int _gr_poly_divexact_bidirectional(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen,
                                    gr_ctx_t ctx)
int gr_poly_divexact_bidirectional(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B,
                                    gr_ctx_t ctx)
int _gr_poly_divexact_basecase_noinv(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen,
                                      gr_ctx_t ctx)
int _gr_poly_divexact_basecase(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong Blen,
                                gr_ctx_t ctx)
int gr_poly_divexact_basecase(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_divexact_series_basecase_noinv(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B,
                                             slong Blen, slong len, gr_ctx_t ctx)
int _gr_poly_divexact_series_basecase(gr_ptr Q, gr_srcptr A, slong Alen, gr_srcptr B, slong
                                       Blen, slong len, gr_ctx_t ctx)
int gr_poly_divexact_series_basecase(gr_poly_t Q, const gr_poly_t A, const gr_poly_t B, slong
                                      len, gr_ctx_t ctx)
```


3.8.17 Square roots

```

int _gr_poly_sqrt_series_newton(gr_ptr res, gr_srcptr f, slong flen, slong len, slong cutoff,
                               gr_ctx_t ctx)
int gr_poly_sqrt_series_newton(gr_poly_t res, const gr_poly_t f, slong len, slong cutoff, gr_ctx_t
                               ctx)
int _gr_poly_sqrt_series_basecase(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_sqrt_series_basecase(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_sqrt_series_miller(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_sqrt_series_miller(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_sqrt_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_sqrt_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)

int _gr_poly_rsqrts_series_newton(gr_ptr res, gr_srcptr f, slong flen, slong len, slong cutoff,
                                  gr_ctx_t ctx)
int gr_poly_rsqrts_series_newton(gr_poly_t res, const gr_poly_t f, slong len, slong cutoff, gr_ctx_t
                                  ctx)
int _gr_poly_rsqrts_series_basecase(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_rsqrts_series_basecase(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_rsqrts_series_miller(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_rsqrts_series_miller(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_rsqrts_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_rsqrts_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)

```

3.8.18 Evaluation

```

int _gr_poly_evaluate_rectangular(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t
                                  ctx)
int gr_poly_evaluate_rectangular(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)
int _gr_poly_evaluate_modular(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_modular(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)

int _gr_poly_evaluate_horner(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate_horner(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)

int _gr_poly_evaluate(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr x, gr_ctx_t ctx)
int gr_poly_evaluate(gr_ptr res, const gr_poly_t poly, gr_srcptr x, gr_ctx_t ctx)
    Set res to poly evaluated at x.

int _gr_poly_evaluate_other_horner(gr_ptr res, gr_srcptr f, slong len, const gr_srcptr x, gr_ctx_t
                                   x_ctx, gr_ctx_t ctx)
int gr_poly_evaluate_other_horner(gr_ptr res, const gr_poly_t f, gr_srcptr x, gr_ctx_t x_ctx,
                                   gr_ctx_t ctx)
int _gr_poly_evaluate_other_rectangular(gr_ptr res, gr_srcptr f, slong len, const gr_srcptr x,
                                         gr_ctx_t x_ctx, gr_ctx_t ctx)
int gr_poly_evaluate_other_rectangular(gr_ptr res, const gr_poly_t f, gr_srcptr x, gr_ctx_t
                                         x_ctx, gr_ctx_t ctx)
int _gr_poly_evaluate_other(gr_ptr res, gr_srcptr f, slong len, const gr_srcptr x, gr_ctx_t x_ctx,
                             gr_ctx_t ctx)

```

```
int gr_poly_evaluate_other(gr_ptr res, const gr_poly_t f, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t
                           ctx)
```

Set *res* to *poly* evaluated at *x*, where the coefficients of *f* belong to *ctx* while both *x* and *res* belong to *x_ctx*.

3.8.19 Newton basis

Let $C = (c_0, c_1, c_2, \dots)$ be a sequence of values of the commutative ring R . Then the sequence of polynomials

$$\begin{aligned} N_0 &= 1, \\ N_1 &= (x - c_0), \\ N_2 &= (x - c_0)(x - c_1), \\ N_3 &= (x - c_0)(x - c_1)(x - c_2), \dots \end{aligned}$$

with roots in C forms a basis of $R[x]$ called the *Newton basis* associated with C . Given a polynomial $f = f_0 + f_1x + \dots + f_nx^n$ in the standard monomial basis, there are accordingly coefficients $g_0, \dots, g_n$ such that $f = g_0N_0 + g_1N_1 + \dots + g_nN_n$ and vice versa. In the following functions, some finite initial segment of C is represented by a vector called *basis*, and a polynomial in the Newton basis is represented formally as the polynomial $g = g_0 + g_1x + \dots + g_nx^n$.

The following functions all support aliasing between inputs and outputs except that no output may be aliased with *basis*.

```
int _gr_poly_newton_basis_from_monomial(gr_ptr res, gr_srcptr basis, gr_srcptr poly, slong len,
                                         gr_ctx_t ctx)
```

```
int gr_poly_newton_basis_from_monomial(gr_poly_t res, const gr_vec_t basis, const gr_poly_t
                                         poly, gr_ctx_t ctx)
```

```
int _gr_poly_newton_basis_to_monomial(gr_ptr res, gr_srcptr basis, gr_srcptr poly, slong len,
                                       gr_ctx_t ctx)
```

```
int gr_poly_newton_basis_to_monomial(gr_poly_t res, const gr_vec_t basis, const gr_poly_t poly,
                                       gr_ctx_t ctx)
```

Given *poly* of length *len* in the standard monomial basis, set *res* to the polynomial in the Newton basis and vice versa. The underscore methods require that *basis* is a pointer to at least *len* - 1 entries. The non-underscore methods return GR_UNABLE if *basis* does not contain at least *len* - 1 entries.

```
int _gr_poly_newton_basis_evaluate(gr_ptr res, gr_srcptr basis, gr_srcptr poly, slong len,
                                    gr_srcptr x, gr_ctx_t ctx)
```

```
int gr_poly_newton_basis_evaluate(gr_ptr res, const gr_vec_t basis, const gr_poly_t poly,
                                    gr_srcptr x, gr_ctx_t ctx)
```

Given *poly* of length *len* in the Newton basis, set *res* to the evaluation at *x*. The underscore method requires that *basis* is a pointer to at least *len* - 1 entries. The non-underscore method returns GR_UNABLE if *basis* does not contain at least *len* - 1 entries.

```
int _gr_poly_newton_basis_interpolate_exact(gr_ptr res, gr_srcptr basis, gr_srcptr ys, slong len,
                                             gr_ctx_t ctx)
```

```
int gr_poly_newton_basis_interpolate_exact(gr_poly_t res, const gr_vec_t basis, const gr_vec_t
                                             ys, gr_ctx_t ctx)
```

```
int _gr_poly_newton_basis_interpolate(gr_ptr res, gr_srcptr basis, gr_srcptr ys, slong len,
                                       gr_ctx_t ctx)
```

```
int gr_poly_newton_basis_interpolate(gr_poly_t res, const gr_vec_t basis, const gr_vec_t ys,
                                       gr_ctx_t ctx)
```

Given a vector *y* containing *len* values, set *res* to the Newton basis form of the unique interpolating polynomial *f* of length (up to) *len* such that $y_0 = f(c_0), \dots, y_{len-1} = f(c_{len-1})$ evaluated at the roots of the Newton basis.

We presume, but do not check, that the ring is an integral domain. These functions may succeed over non-integral domains, but the result need not be an interpolating polynomial.

We require that *basis* contains at least *len* initial roots which are pairwise distinct.

These functions return GR_DOMAIN (and/or GR_UNABLE) in either of the following situations:

- The vector *basis* does not have sufficiently many points (checked by the non-underscore functions only).
- The evaluation points *xs* are not pairwise distinct.
- The interpolating polynomial *f* has coefficients in the fraction field of *R* but not in *R* itself.

The *exact* versions presume that the evaluation points are distinct and that *f* has coefficients in *R*; they may silently output some arbitrary polynomial otherwise.

3.8.20 Multipoint evaluation and interpolation

```
gr_ptr *_gr_poly_tree_alloc(slong len, gr_ctx_t ctx)
```

```
void _gr_poly_tree_free(gr_ptr *tree, slong len, gr_ctx_t ctx)
```

```
int _gr_poly_tree_build(gr_ptr *tree, gr_srcptr roots, slong len, gr_ctx_t ctx)
```

Initialize a subproduct tree over the given roots.

```
int _gr_poly_product_roots(gr_ptr poly, gr_srcptr xs, slong n, gr_ctx_t ctx)
```

```
int gr_poly_product_roots(gr_poly_t poly, const gr_vec_t xs, gr_ctx_t ctx)
```

Set *poly* to the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

```
int _gr_poly_evaluate_vec_fast_precomp(gr_ptr vs, gr_srcptr poly, slong plen, const gr_ptr *tree,
                                       slong len, gr_ctx_t ctx)
```

```
int _gr_poly_evaluate_vec_fast(gr_ptr ys, gr_srcptr poly, slong plen, gr_srcptr xs, slong n,
                               gr_ctx_t ctx)
```

```
int gr_poly_evaluate_vec_fast(gr_vec_t ys, const gr_poly_t poly, const gr_vec_t xs, gr_ctx_t
                              ctx)
```

```
int _gr_poly_evaluate_vec_iter(gr_ptr ys, gr_srcptr poly, slong plen, gr_srcptr xs, slong n,
                              gr_ctx_t ctx)
```

```
int gr_poly_evaluate_vec_iter(gr_vec_t ys, const gr_poly_t poly, const gr_vec_t xs, gr_ctx_t
                              ctx)
```

```
int _gr_poly_interpolate_exact(gr_ptr res, gr_srcptr xs, gr_srcptr ys, slong len, gr_ctx_t ctx)
```

```
int gr_poly_interpolate_exact(gr_poly_t poly, const gr_vec_t xs, const gr_vec_t ys, gr_ctx_t
                              ctx)
```

```
int _gr_poly_interpolate(gr_ptr res, gr_srcptr xs, gr_srcptr ys, slong len, gr_ctx_t ctx)
```

```
int gr_poly_interpolate(gr_poly_t poly, const gr_vec_t xs, const gr_vec_t ys, gr_ctx_t ctx)
```

Given vectors *xs* and *ys* of length *len* where the entries of *xs* are pairwise distinct, set *res* to the interpolating polynomial *f* of length (up to) *len* such that $y_0 = f(x_0), \dots, y_{len-1} = f(x_{len-1})$.

We presume, but do not check, that the ring is an integral domain. These functions may succeed over non-integral domains, but the result might not be an interpolating polynomial.

These functions return GR_DOMAIN (and/or GR_UNABLE) in either of the following situations:

- The vectors *xs* and *ys* do not have the same length (checked by the non-underscore functions only).
- The evaluation points *xs* are not pairwise distinct.
- The interpolating polynomial *f* has coefficients in the fraction field of *R* but not in *R* itself.

The *exact* versions presume that the evaluation points are distinct and that f has coefficients in R ; they may silently output some arbitrary polynomial otherwise.

```
int _gr_poly_interpolation_weights(gr_ptr w, const gr_ptr *tree, slong len, gr_ctx_t ctx)
int _gr_poly_interpolate_fast_precomp(gr_ptr poly, gr_srcptr ys, const gr_ptr *tree, gr_srcptr
                                     weights, slong len, gr_ctx_t ctx)
int _gr_poly_interpolate_fast(gr_ptr res, gr_srcptr xs, gr_srcptr ys, slong len, gr_ctx_t ctx)
int gr_poly_interpolate_fast(gr_poly_t poly, const gr_vec_t xs, const gr_vec_t ys, gr_ctx_t ctx)
    Fast polynomial interpolation using a subproduct tree. The precomp version requires a precomputed
    subproduct tree generated using _gr_poly_tree_build() and precomputed interpolation weights
    generated using _gr_poly_interpolation_weights().
```

This currently requires a field.

3.8.21 Composition

```
int _gr_poly_taylor_shift_horner(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_poly_taylor_shift_horner(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
int _gr_poly_taylor_shift_divconquer(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t
                                     ctx)
int gr_poly_taylor_shift_divconquer(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t
                                     ctx)
int _gr_poly_taylor_shift_convolution(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c,
                                     gr_ctx_t ctx)
int gr_poly_taylor_shift_convolution(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t
                                     ctx)
int _gr_poly_taylor_shift(gr_ptr res, gr_srcptr poly, slong len, gr_srcptr c, gr_ctx_t ctx)
int gr_poly_taylor_shift(gr_poly_t res, const gr_poly_t poly, gr_srcptr c, gr_ctx_t ctx)
    Sets res to the Taylor shift  $f(x + c)$ , where  $f$  is given by poly, computed respectively using an
    optimized form of Horner's rule, divide-and-conquer, a single convolution, and an automatic choice
    between the three algorithms. The underscore methods support aliasing.
int _gr_poly_compose_horner(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
                             gr_ctx_t ctx)
int gr_poly_compose_horner(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t
                             ctx)
int _gr_poly_compose_divconquer(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong
                                len2, gr_ctx_t ctx)
int gr_poly_compose_divconquer(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,
                                gr_ctx_t ctx)
int _gr_poly_compose(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2, gr_ctx_t
                      ctx)
int gr_poly_compose(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, gr_ctx_t ctx)
    Sets res to the composition  $f(g(x))$  where  $f$  is given by poly1 and  $g$  is given by poly2, respectively
    using Horner's rule, divide-and-conquer, and an automatic choice between the two algorithms. The
    default algorithm also handles special-form input  $g = ax^n + c$  efficiently by performing a Taylor
    shift followed by a rescaling. The underscore methods do not support aliasing of the output with
    either input polynomial.
```

3.8.22 Power series composition and reversion

```

int _gr_poly_compose_series_horner(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong
    len2, slong n, gr_ctx_t ctx)
int gr_poly_compose_series_horner(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,
    slong n, gr_ctx_t ctx)
int _gr_poly_compose_series_brent_kung(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,
    slong len2, slong n, gr_ctx_t ctx)
int gr_poly_compose_series_brent_kung(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t
    poly2, slong n, gr_ctx_t ctx)
int _gr_poly_compose_series_divconquer(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,
    slong len2, slong n, gr_ctx_t ctx)
int gr_poly_compose_series_divconquer(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t
    poly2, slong n, gr_ctx_t ctx)
int _gr_poly_compose_series(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
    slong n, gr_ctx_t ctx)
int gr_poly_compose_series(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, slong n,
    gr_ctx_t ctx)

```

Sets *res* to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where f is given by *poly1* and g is given by *poly2*, respectively using Horner's rule, the Brent-Kung baby step-giant step algorithm [BrentKung1978], divide-and-conquer, and an automatic choice between the algorithms.

The default algorithm also handles short input and special-form input $g = ax^n$ efficiently.

We require that the constant term in $g(x)$ is exactly zero. The underscore methods do not support aliasing of the output with either input polynomial, and do not zero-pad the result.

```

int _gr_poly_revert_series_lagrange(gr_ptr res, gr_srcptr f, slong flen, slong n, gr_ctx_t ctx)
int gr_poly_revert_series_lagrange(gr_poly_t res, const gr_poly_t f, slong n, gr_ctx_t ctx)
int _gr_poly_revert_series_lagrange_fast(gr_ptr res, gr_srcptr f, slong flen, slong n, gr_ctx_t
    ctx)
int gr_poly_revert_series_lagrange_fast(gr_poly_t res, const gr_poly_t f, slong n, gr_ctx_t ctx)
int _gr_poly_revert_series_newton(gr_ptr res, gr_srcptr f, slong flen, slong n, gr_ctx_t ctx)
int gr_poly_revert_series_newton(gr_poly_t res, const gr_poly_t f, slong n, gr_ctx_t ctx)
int _gr_poly_revert_series(gr_ptr res, gr_srcptr f, slong flen, slong n, gr_ctx_t ctx)
int gr_poly_revert_series(gr_poly_t res, const gr_poly_t f, slong n, gr_ctx_t ctx)

```

Sets *res* to the power series reversion $f^{-1}(x)$ which satisfies $f^{-1}(f(x)) = f(f^{-1}(x)) = x \bmod x^n$. For the series reversion to exist, we require that the constant term in f is zero and that the linear coefficient is invertible. The flag `GR_DOMAIN` is returned otherwise.

The *lagrange* and *lagrange_fast* algorithms require the ability to divide by $2, 3, \dots, n-1$ and will return the `GR_UNABLE` flag in too small characteristic.

The underscore methods do not support aliasing of the output with the input.

The Newton method is described in [BrentKung1978]; the *lagrange* algorithm implements the Lagrange inversion formula, while the *lagrange_fast* algorithm implements the baby-step giant-step algorithm described in [Joh2015b].

3.8.23 Derivative and integral

```
int _gr_poly_derivative(gr_ptr res, gr_srcptr poly, slong len, gr_ctx_t ctx)
int gr_poly_derivative(gr_poly_t res, const gr_poly_t poly, gr_ctx_t ctx)

int _gr_poly_nth_derivative(gr_ptr res, gr_srcptr poly, ulong n, slong len, gr_ctx_t ctx)
int gr_poly_nth_derivative(gr_poly_t res, const gr_poly_t poly, ulong n, gr_ctx_t ctx)

int _gr_poly_integral(gr_ptr res, gr_srcptr poly, slong len, gr_ctx_t ctx)
int gr_poly_integral(gr_poly_t res, const gr_poly_t poly, gr_ctx_t ctx)
```

3.8.24 Monic polynomials

```
int _gr_poly_make_monic(gr_ptr res, gr_srcptr poly, slong len, gr_ctx_t ctx)
int gr_poly_make_monic(gr_poly_t res, const gr_poly_t src, gr_ctx_t ctx)

truth_t _gr_poly_is_monic(gr_srcptr poly, slong len, gr_ctx_t ctx)
truth_t gr_poly_is_monic(const gr_poly_t res, gr_ctx_t ctx)

int gr_poly_canonical_associate(gr_poly_t res, gr_poly_t u, const gr_poly_t src, gr_ctx_t ctx)
```

3.8.25 GCD

```
int _gr_poly_hgcd(gr_ptr r, slong *sgn, gr_ptr *M, slong *lenM, gr_ptr A, slong *lenA, gr_ptr B,
                 slong *lenB, gr_srcptr a, slong lena, gr_srcptr b, slong lenb, slong cutoff,
                 gr_ctx_t ctx)
```

Computes the HGCD of a and b , that is, a matrix M , a sign σ and two polynomials A and B such that

$$(A, B)^t = \sigma M^{-1}(a, b)^t.$$

Assumes that $\text{len}(a) > \text{len}(b) > 0$.

Assumes that A and B have space of size at least $\text{len}(a)$ and $\text{len}(b)$, respectively. On exit, $*lenA$ and $*lenB$ will contain the correct lengths of A and B .

Assumes that $M[0]$, $M[1]$, $M[2]$, and $M[3]$ each point to a vector of size at least $\text{len}(a)$.

If r is not NULL, writes to that variable the corresponding value for computing resultants using the HGCD algorithm.

```
int _gr_poly_gcd_hgcd(gr_ptr G, slong *_lenG, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                    slong inner_cutoff, slong cutoff, gr_ctx_t ctx)
int gr_poly_gcd_hgcd(gr_poly_t G, const gr_poly_t A, const gr_poly_t B, slong inner_cutoff, slong
                    cutoff, gr_ctx_t ctx)
int _gr_poly_gcd_euclidean(gr_ptr G, slong *_lenG, gr_srcptr A, slong lenA, gr_srcptr B, slong
                        lenB, gr_ctx_t ctx)
int gr_poly_gcd_euclidean(gr_poly_t G, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_gcd_subresultant(gr_ptr G, slong *_lenG, gr_srcptr A, slong lenA, gr_srcptr B, slong
                            lenB, gr_ctx_t ctx)
int gr_poly_gcd_subresultant(gr_poly_t G, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
int _gr_poly_gcd_generic(gr_ptr G, slong *_lenG, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB,
                        gr_ctx_t ctx)
int gr_poly_gcd(gr_ptr G, slong *_lenG, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, gr_ctx_t
                ctx)
```

```
int gr_poly_gcd(gr_poly_t G, const gr_poly_t A, const gr_poly_t B, gr_ctx_t ctx)
```

GCD in the polynomial ring $R[x]$.

The `gcd_subresultant` algorithm assumes that R is a unique factorization domain. The `euclidean` and `hgcd` algorithms assume that R is a field. The time complexity of the half-GCD algorithm is $\mathcal{O}(n \log^2 n)$ field operations. For further details, see [ThullYap1990].

The `gcd_generic` fallback implementation checks if R is a field or UFD and dispatches to an appropriate algorithm, and otherwise returns `GR_UNABLE`. The main `gcd` function is synonymous with `gcd_generic` unless overridden by a particular ring.

The non-underscore methods canonicalise the leading coefficient of the output by calling `gr_poly_canonical_associate`. Over a field, this corresponds to making the output monic. The underscore methods need not perform this canonicalisation.

The underscore methods assume `lenA >= lenB >= 1` and that both A and B have nonzero leading coefficient.

```
int _gr_poly_xgcd_euclidean(slong *lenG, gr_ptr G, gr_ptr S, gr_ptr T, gr_srcptr A, slong lenA,
                           gr_srcptr B, slong lenB, gr_ctx_t ctx)
```

```
int gr_poly_xgcd_euclidean(gr_poly_t G, gr_poly_t S, gr_poly_t T, const gr_poly_t A, const
                           gr_poly_t B, gr_ctx_t ctx)
```

```
int _gr_poly_xgcd_hgcd(slong *lenG, gr_ptr G, gr_ptr S, gr_ptr T, gr_srcptr A, slong lenA,
                      gr_srcptr B, slong lenB, slong hgcd_cutoff, slong cutoff, gr_ctx_t ctx)
```

```
int gr_poly_xgcd_hgcd(gr_poly_t G, gr_poly_t S, gr_poly_t T, const gr_poly_t A, const gr_poly_t
                      B, slong hgcd_cutoff, slong cutoff, gr_ctx_t ctx)
```

```
int _gr_poly_xgcd_generic(slong *lenG, gr_ptr G, gr_ptr S, gr_ptr T, gr_srcptr A, slong lenA,
                          gr_srcptr B, slong lenB, gr_ctx_t ctx)
```

```
int _gr_poly_xgcd(slong *lenG, gr_ptr G, gr_ptr S, gr_ptr T, gr_srcptr A, slong lenA, gr_srcptr B,
                  slong lenB, gr_ctx_t ctx)
```

```
int gr_poly_xgcd(gr_poly_t G, gr_poly_t S, gr_poly_t T, const gr_poly_t A, const gr_poly_t B,
                 gr_ctx_t ctx)
```

3.8.26 Resultant

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

```
int _gr_poly_resultant_euclidean(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong
                                len2, gr_ctx_t ctx)
```

```
int gr_poly_resultant_euclidean(gr_ptr res, const gr_poly_t f, const gr_poly_t g, gr_ctx_t ctx)
```

```
int _gr_poly_resultant_hgcd(gr_ptr res, gr_srcptr A, slong lenA, gr_srcptr B, slong lenB, slong
                            inner_cutoff, slong cutoff, gr_ctx_t ctx)
```

```
int gr_poly_resultant_hgcd(gr_ptr res, const gr_poly_t f, const gr_poly_t g, slong inner_cutoff,
                           slong cutoff, gr_ctx_t ctx)
```

```
int _gr_poly_resultant_sylvester(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong
                                 len2, gr_ctx_t ctx)
```

```
int gr_poly_resultant_sylvester(gr_ptr res, const gr_poly_t f, const gr_poly_t g, gr_ctx_t ctx)
```

```
int _gr_poly_resultant_small(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
                             gr_ctx_t ctx)
```

```
int gr_poly_resultant_small(gr_ptr res, const gr_poly_t f, const gr_poly_t g, gr_ctx_t ctx)
int _gr_poly_resultant(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, slong len2,
                      gr_ctx_t ctx)
```

```
int gr_poly_resultant(gr_ptr res, const gr_poly_t f, const gr_poly_t g, gr_ctx_t ctx)
```

Sets *res* to the resultant of *poly1* and *poly2*. The underscore methods assume that $len1 \geq len2 \geq 1$ and that the leading coefficients are nonzero.

The *euclidean* algorithm is the ordinary Euclidean algorithm. The *hgcd* version uses the quasilinear half-GCD algorithm. It requires two extra tuning parameters *inner_cutoff* (recursion threshold passed forward to the HGCD algorithm) and *cutoff*. Both algorithms can fail when run over non-fields; they will return *GR_DOMAIN* when encountering an impossible inverse.

The *small* version uses division-free straight-line programs optimized for short polynomials. It returns *GR_UNABLE* if the polynomials are too large. Currently this function handles the cases where $len1 \leq 2$ or $len2 \leq 3$.

The *sylvester* version constructs the Sylvester matrix and computes its determinant. This is useful over inexact rings and as a fallback for rings without division.

The default version attempts to choose an appropriate algorithm automatically.

Currently no algorithm has been implemented that is appropriate for integral domains.

3.8.27 Squarefree factorization

TODO: currently only fields of characteristic 0 are supported.

```
int gr_poly_factor_squarefree(gr_ptr c, gr_vec_t fac, gr_vec_t exp, const gr_poly_t poly,
                             gr_ctx_t ctx)
```

Computes a squarefree factorization of *poly*.

The constant *c* is set to an element of the scalar ring. The factors in *fac* are set to polynomials; the user must thus initialize it to a vector of polynomials of the same type as *poly* (and *not* to the parent *ctx*). The exponent vector *exp* must be initialized to the *fmpz* type.

```
int gr_poly_squarefree_part(gr_poly_t res, const gr_poly_t poly, gr_ctx_t ctx)
```

Sets *res* to the squarefree part of *poly*.

3.8.28 Shift equivalence

```
truth_t gr_poly_shift_equivalent(fmpz_t shift, const gr_poly_t p, const gr_poly_t q, gr_ctx_t
                                ctx)
```

Returns whether there exists an integer n such that $p(x + n) = q(x)$. If the result is *T_TRUE* and *shift* is not *NULL*, *shift* is set to such an n .

```
int gr_poly_leading_taylor_shift(gr_ptr shift, const gr_poly_t p, const gr_poly_t q, gr_ctx_t
                                ctx)
```

Computes (if possible) s such that $p(x + s) = q(x)(1 + O(x^2))$.

```
int gr_poly_dispersion_resultant(fmpz_t disp, gr_vec_t disp_set, const gr_poly_t f, const
                                gr_poly_t g, gr_ctx_t ctx);
```

```
int gr_poly_dispersion_factor(fmpz_t disp, gr_vec_t disp_set, const gr_poly_t f, const gr_poly_t
                              g, gr_ctx_t ctx);
```

```
int gr_poly_dispersion(fmpz_t disp, gr_vec_t disp_set, const gr_poly_t f, const gr_poly_t g,
                      gr_ctx_t ctx);
```

Computes the dispersion and/or the dispersion set of *f* and *g*.

The dispersion set of two polynomials f and g (over a unique factorization domain of characteristic zero) is the set of nonnegative integers n such that $f(x+n)$ and $g(x)$ have a nonconstant common factor. The dispersion is the largest element of the dispersion set.

The output variables `disp` and/or `disp_set` can be NULL, in which case the corresponding result is not stored. When the dispersion set is empty, `disp` is left unchanged. The elements of `disp_set` are sorted in increasing order.

The *factor* version uses the algorithm described in [ManWright1994]. The *resultant* version computes the integer roots of a bivariate resultant and is mainly intended for testing.

```
int gr_poly_dispersion_from_factors(fmpz_t disp, gr_vec_t disp_set, const gr_vec_t ffac, const
                                   gr_vec_t gfac, gr_ctx_t ctx);
```

Same as `gr_poly_dispersion_factor()` for nonzero f and g but takes as input their nonconstant irreducible factors (without multiplicities) instead of the polynomials themselves.

```
int gr_poly_shiftless_decomposition_factor(gr_ptr c, gr_vec_t slfac, gr_vec_t slshifts,
                                           gr_vec_t slmult, const gr_poly_t f, gr_ctx_t ctx)

int gr_poly_shiftless_decomposition(gr_ptr c, gr_vec_t slfac, gr_vec_t slshifts, gr_vec_t slmult,
                                   const gr_poly_t f, gr_ctx_t ctx)
```

Computes a decomposition of f of the form

$$c \prod_i \prod_j g_i(x + h_{i,j})^{e_{i,j}}$$

where

- c is a constant,
- the g_i are squarefree polynomials of degree at least one,
- $g_i(x)$ and $g_j(x+h)$ (with $i \neq j$) are coprime for all $h \in \mathbb{Z}$,
- $g_i(x)$ and $g_i(x+h)$ are coprime for all nonzero $h \in \mathbb{Z}$,
- $e_{i,j}$ and $h_{i,j}$ are integers with $e_{i,j} \geq 1$ and $0 = h_{i,1} < h_{i,2} < \dots$.

The output variable `slfac` must be initialized to a vector of polynomials of the same type as f . The other two output vectors `slshift` and `slmult` must be initialized to vectors of *vectors* with entries of type `fmpz`.

The *factor* version computes an irreducible factorization and sorts the factors into shift-equivalence classes.

No algorithm avoiding a full irreducible factorization is currently implemented.

```
int _gr_poly_shiftless_decomposition_from_factors(gr_vec_t slfac, gr_vec_t slshifts, gr_vec_t
                                                  slmult, const gr_vec_t fac, const gr_vec_t
                                                  mult, gr_ctx_t ctx)

int gr_poly_shiftless_decomposition_from_factors(gr_vec_t slfac, gr_vec_t slshifts, gr_vec_t
                                                  slmult, const gr_vec_t fac, const gr_vec_t
                                                  mult, gr_ctx_t ctx)
```

Same as `gr_poly_shiftless_decomposition_factor()` but takes as input an irreducible factorization (`fac`, `mult`) of f (without the prefactor c). The underscore method does not support aliasing of `slfac` with `fac`.

3.8.29 Roots

```
int gr_poly_roots(gr_vec_t roots, gr_vec_t mult, const gr_poly_t poly, int flags, gr_ctx_t ctx)
int gr_poly_roots_other(gr_vec_t roots, gr_vec_t mult, const gr_poly_t poly, gr_ctx_t poly_ctx,
                        int flags, gr_ctx_t ctx)
```

Finds all roots of the given polynomial in the ring defined by *ctx*, storing the roots without duplication in *roots* (a vector with elements of type *ctx*) and the corresponding multiplicities in *mult* (a vector with elements of type *fmpz*).

If the target ring is not an algebraically closed field, then the sum of multiplicities can be smaller than the degree of the polynomial. For example, with *fmpz* coefficients, we only find integer roots. The *other* version of this function takes as input a polynomial with entries in a different ring *poly_ctx*. For example, we can compute *qqbar* or *arb* roots for a polynomial with *fmpz* coefficients.

Whether the roots are sorted in any particular order is ring-dependent.

We consider roots of the zero polynomial to be ill-defined and return *GR_DOMAIN* in that case.

```
int _gr_poly_refine_roots_aberth(gr_ptr w, gr_srcptr f, gr_srcptr f_prime, slong deg, gr_srcptr
                                z, int progressive, gr_ctx_t ctx)
int _gr_poly_refine_roots_wdk(gr_ptr w, gr_srcptr f, slong deg, gr_srcptr z, int progressive,
                              gr_ctx_t ctx)
```

Given a vector of approximate complex roots $z_1, \dots, z_{deg}$ of $f = \sum_{i=0}^{deg} f_i x^i$, computes a vector of corrections $w_1, \dots, w_{deg}$ such that $z_k - w_k$ is a closer approximation of the respective root provided that the initial approximations are close enough and that the polynomial evaluation is numerically accurate. The user will typically call these methods in a loop.

The *wdk* version performs the Weierstrass-Durand-Kerner update

$$w_k = \frac{f(z_k)}{\prod_{j \neq k} (z_k - z_j)}, \quad k = 1, \dots, deg.$$

The *aberth* version performs the Aberth-Ehrlich update

$$w_k = \frac{g(z_k)}{1 - g(z_k) \sum_{j \neq k} (z_k - z_j)^{-1}}, \quad g(z_k) = \frac{f(z_k)}{f'(z_k)} \quad k = 1, \dots, deg.$$

requiring the coefficients of f' as an extra input *f_prime*.

If *progressive* flag is set, corrected roots $z_j - w_j$ that have already been computed are used in place of z_j in the update loop, which can improve the rate of convergence.

3.8.30 Power series special functions

```
int _gr_poly_asin_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_asin_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_asinh_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_asinh_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_acos_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_acos_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_acosh_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_acosh_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_atan_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_atan_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_atanh_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
```



```

int gr_poly_atanh_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)

int _gr_poly_log_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_log_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)
int _gr_poly_log1p_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_log1p_series(gr_poly_t res, const gr_poly_t f, slong len, gr_ctx_t ctx)

int _gr_poly_exp_series_basecase(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_exp_series_basecase(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_exp_series_basecase_mul(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_exp_series_basecase_mul(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_exp_series_newton(gr_ptr f, gr_ptr g, gr_srcptr h, slong hlen, slong n, slong cutoff,
                               gr_ctx_t ctx)
int gr_poly_exp_series_newton(gr_poly_t f, const gr_poly_t h, slong n, slong cutoff, gr_ctx_t ctx)
int _gr_poly_exp_series_generic(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int _gr_poly_exp_series(gr_ptr res, gr_srcptr f, slong flen, slong len, gr_ctx_t ctx)
int gr_poly_exp_series(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)

int _gr_poly_sin_cos_series_basecase(gr_ptr s, gr_ptr c, gr_srcptr h, slong hlen, slong n, int
                                     times_pi, gr_ctx_t ctx)
int gr_poly_sin_cos_series_basecase(gr_poly_t s, gr_poly_t c, const gr_poly_t h, slong n, int
                                     times_pi, gr_ctx_t ctx)
int _gr_poly_sin_cos_series_tangent(gr_ptr s, gr_ptr c, gr_srcptr h, slong hlen, slong n, int
                                    times_pi, gr_ctx_t ctx)
int gr_poly_sin_cos_series_tangent(gr_poly_t s, gr_poly_t c, const gr_poly_t h, slong n, int
                                   times_pi, gr_ctx_t ctx)

```

The *basecase* version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where m is the length of h .

The *tangent* version uses the tangent half-angle formulas to compute the sine and cosine via `_acb_poly_tan_series()`. This requires $O(M(n))$ operations. When $h = h_0 + h_1$ where the constant term h_0 is nonzero, the evaluation is done as $\sin(h_0 + h_1) = \cos(h_0) \sin(h_1) + \sin(h_0) \cos(h_1)$, $\cos(h_0 + h_1) = \cos(h_0) \cos(h_1) - \sin(h_0) \sin(h_1)$.

The *basecase* and *tangent* versions take a flag `times_pi` specifying that the input is to be multiplied by π .

```

int _gr_poly_tan_series_basecase(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_tan_series_basecase(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)
int _gr_poly_tan_series_newton(gr_ptr f, gr_srcptr h, slong hlen, slong n, slong cutoff, gr_ctx_t
                               ctx)
int gr_poly_tan_series_newton(gr_poly_t f, const gr_poly_t h, slong n, slong cutoff, gr_ctx_t ctx)
int _gr_poly_tan_series(gr_ptr f, gr_srcptr h, slong hlen, slong n, gr_ctx_t ctx)
int gr_poly_tan_series(gr_poly_t f, const gr_poly_t h, slong n, gr_ctx_t ctx)

```

3.8.31 Modular arithmetic and composition

int **_gr_poly_mulmod**(*gr_ptr* res, *gr_srcptr* poly1, *slong* len1, *gr_srcptr* poly2, *slong* len2, *gr_srcptr* f, *slong* lenf, *gr_ctx_t* ctx)

Sets **res** to the remainder of the product of **poly1** and **poly2** upon polynomial division by **f**.

It is required that $\text{len1} + \text{len2} - \text{lenf} > 0$, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use **_fq_poly_mul** instead.

Aliasing of **f** and **res** is not permitted.

int **gr_poly_mulmod**(*gr_poly_t* res, const *gr_poly_t* poly1, const *gr_poly_t* poly2, const *gr_poly_t* f, *gr_ctx_t* ctx)

Sets **res** to the remainder of the product of **poly1** and **poly2** upon polynomial division by **f**.

int **_gr_poly_mulmod_preinv**(*gr_ptr* res, *gr_srcptr* poly1, *slong* len1, *gr_srcptr* poly2, *slong* len2, *gr_srcptr* f, *slong* lenf, *gr_srcptr* finv, *slong* lenfinv, *gr_ctx_t* ctx)

Sets **res** to the remainder of the product of **poly1** and **poly2** upon polynomial division by **f**.

It is required that **finv** is the inverse of the reverse of **f** mod x^{lenf} .

Aliasing of **res** with any of the inputs is not permitted.

int **gr_poly_mulmod_preinv**(*gr_poly_t* res, const *gr_poly_t* poly1, const *gr_poly_t* poly2, const *gr_poly_t* f, const *gr_poly_t* finv, *gr_ctx_t* ctx)

Sets **res** to the remainder of the product of **poly1** and **poly2** upon polynomial division by **f**. **finv** is the inverse of the reverse of **f**.

int **_gr_poly_powmod_fmpz_binexp**(*gr_ptr* res, *gr_srcptr* poly, const *fmpz_t* e, *gr_srcptr* f, *slong* lenf, *gr_ctx_t* ctx)

int **gr_poly_powmod_fmpz_binexp**(*gr_poly_t* res, const *gr_poly_t* poly, const *fmpz_t* e, const *gr_poly_t* f, *gr_ctx_t* ctx)

int **_gr_poly_powmod_fmpz_binexp_preinv**(*gr_ptr* res, *gr_srcptr* poly, const *fmpz_t* e, *gr_srcptr* f, *slong* lenf, *gr_srcptr* finv, *slong* lenfinv, *gr_ctx_t* ctx)

int **gr_poly_powmod_fmpz_binexp_preinv**(*gr_poly_t* res, const *gr_poly_t* poly, const *fmpz_t* e, const *gr_poly_t* f, const *gr_poly_t* finv, *gr_ctx_t* ctx)

int **_gr_poly_powmod_x_fmpz_preinv**(*gr_ptr* res, const *fmpz_t* e, *gr_srcptr* f, *slong* lenf, *gr_srcptr* finv, *slong* lenfinv, *gr_ctx_t* ctx)

int **gr_poly_powmod_x_fmpz_preinv**(*gr_poly_t* res, const *fmpz_t* e, const *gr_poly_t* f, const *gr_poly_t* finv, *gr_ctx_t* ctx)

int **_gr_poly_powmod_ui_binexp**(*gr_ptr* res, *gr_srcptr* poly, *ulong* e, *gr_srcptr* f, *slong* lenf, *gr_ctx_t* ctx)

int **gr_poly_powmod_ui_binexp**(*gr_poly_t* res, const *gr_poly_t* poly, *ulong* e, const *gr_poly_t* f, *gr_ctx_t* ctx)

int **_gr_poly_powmod_ui_binexp_preinv**(*gr_ptr* res, *gr_srcptr* poly, *ulong* e, *gr_srcptr* f, *slong* lenf, *gr_srcptr* finv, *slong* lenfinv, *gr_ctx_t* ctx)

int **gr_poly_powmod_ui_binexp_preinv**(*gr_poly_t* res, const *gr_poly_t* poly, *ulong* e, const *gr_poly_t* f, const *gr_poly_t* finv, *gr_ctx_t* ctx)

int **_gr_poly_powmod_fmpz_sliding_preinv**(*gr_ptr* res, *gr_srcptr* poly, const *fmpz_t* e, *ulong* k, *gr_srcptr* f, *slong* lenf, *gr_srcptr* finv, *slong* lenfinv, *gr_ctx_t* ctx)

int **gr_poly_powmod_fmpz_sliding_preinv**(*gr_poly_t* res, const *gr_poly_t* poly, const *fmpz_t* e, *ulong* k, const *gr_poly_t* f, const *gr_poly_t* finv, *gr_ctx_t* ctx)

Sets **res** to poly raised to the power **e** modulo **f**.

int **_gr_poly_compose_mod_horner**(*gr_ptr* res, *gr_srcptr* poly1, *slong* len1, *gr_srcptr* poly2, *gr_srcptr* poly3, *slong* len3, *gr_ctx_t* ctx)

```

int gr_poly_compose_mod_horner(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,
                               const gr_poly_t poly3, gr_ctx_t ctx)
int _gr_poly_compose_mod_brent_kung(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,
                                     gr_srcptr poly3, slong len3, gr_ctx_t ctx)
int gr_poly_compose_mod_brent_kung(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,
                                   const gr_poly_t poly3, gr_ctx_t ctx)
int _gr_poly_compose_mod(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2, gr_srcptr poly3,
                        slong len3, gr_ctx_t ctx)
int gr_poly_compose_mod(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2, const
                        gr_poly_t poly3, gr_ctx_t ctx)

```

Sets `res` to the composition $f(g)$ modulo h where f , g and h are given by `poly1`, `poly2` and `poly3`.

The underscore methods require that the lengths are nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . The underscore methods do not support aliasing.

```

int _gr_poly_compose_mod_horner_preinv(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,
                                       gr_srcptr poly3, slong len3, gr_srcptr poly3inv, slong
                                       inv3len, gr_ctx_t ctx)
int gr_poly_compose_mod_horner_preinv(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t
                                       poly2, const gr_poly_t poly3, const gr_poly_t poly3inv,
                                       gr_ctx_t ctx)
int _gr_poly_compose_mod_brent_kung_preinv(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr
                                           poly2, gr_srcptr poly3, slong len3, gr_srcptr
                                           poly3inv, slong inv3len, gr_ctx_t ctx)
int gr_poly_compose_mod_brent_kung_preinv(gr_poly_t res, const gr_poly_t poly1, const
                                           gr_poly_t poly2, const gr_poly_t poly3, const
                                           gr_poly_t poly3inv, gr_ctx_t ctx)
int _gr_poly_compose_mod_preinv(gr_ptr res, gr_srcptr poly1, slong len1, gr_srcptr poly2,
                                gr_srcptr poly3, slong len3, gr_srcptr poly3inv, slong inv3len,
                                gr_ctx_t ctx)
int gr_poly_compose_mod_preinv(gr_poly_t res, const gr_poly_t poly1, const gr_poly_t poly2,
                               const gr_poly_t poly3, const gr_poly_t poly3inv, gr_ctx_t ctx)

```

Versions accepting an additional precomputed argument `poly3inv` representing the inverse of the reverse of `poly3`.

```

int _gr_poly_reduce_matrix_mod_poly(gr_mat_t A, const gr_mat_t B, const gr_poly_t f,
                                    gr_ctx_t ctx)

```

Sets the i th row of A to the reduction of the i th row of B modulo f for $i = 1, \dots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```

int _gr_poly_precompute_matrix(gr_mat_t A, gr_srcptr f, gr_srcptr g, slong leng, gr_srcptr ginv,
                              slong lenginv, gr_ctx_t ctx)

```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g and g to be nonzero.

```

int gr_poly_precompute_matrix(gr_mat_t A, const gr_poly_t f, const gr_poly_t g, const gr_poly_t
                              ginv, gr_ctx_t ctx)

```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g .

```

int _gr_poly_compose_mod_brent_kung_precomp_preinv(gr_ptr res, gr_srcptr f, slong lenf, const
                                                    gr_mat_t A, gr_srcptr h, slong lenh,
                                                    gr_srcptr hinv, slong lenhinv, gr_ctx_t
                                                    ctx)

```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. We require that the i th

row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
int gr_poly_compose_mod_brent_kung_precomp_preinv(gr_poly_t res, const gr_poly_t f, const
                                                    gr_mat_t A, const gr_poly_t h, const
                                                    gr_poly_t hinv, gr_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h .

3.8.32 Test functions

The following functions run *iters* test iterations, generating polynomials up to length *maxn*. If *ctx* is set to NULL, a random ring is generated on each test iteration, otherwise the given ring is used.

```
void _gr_poly_test_mullof(gr_method_poly_binary_trunc_op mullof_impl,
                          gr_method_poly_binary_trunc_op mullof_ref, flint_rand_t state,
                          slong iters, slong maxn, gr_ctx_t ctx)
```

```
void _gr_poly_test_mulmid(gr_method_poly_binary_trunc_op mulmid_impl,
                          gr_method_poly_binary_trunc_op mulmid_ref, flint_rand_t state,
                          slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `mullof_impl` (`mulmid_impl`) for correctness as an implementation of `_gr_poly_mullof()` (`_gr_poly_mulmid()`). A reference implementation to compare against can be provided as `mullof_ref` (`mulmid_ref`); if NULL, classical multiplication is used.

```
void _gr_poly_test_divrem(gr_method_poly_binary_op divrem_impl, flint_rand_t state,
                          slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `divrem_impl` for correctness as an implementation of `_gr_poly_divrem()`.

```
void _gr_poly_test_div(gr_method_poly_binary_op div_impl, flint_rand_t state, slong iters,
                      slong maxn, gr_ctx_t ctx)
```

Tests the given function `div_impl` for correctness as an implementation of `_gr_poly_div()`.

```
void _gr_poly_test_inv_series(gr_method_poly_unary_trunc_op inv_series_impl, flint_rand_t
                             state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `inv_series_impl` for correctness as an implementation of `_gr_poly_inv_series()`.

```
void _gr_poly_test_div_series(gr_method_poly_binary_trunc_op div_series_impl, flint_rand_t
                              state, slong iters, slong maxn, gr_ctx_t ctx)
```

Tests the given function `div_series_impl` for correctness as an implementation of `_gr_poly_div_series()`.

```
void _gr_poly_test_gcd(gr_method_poly_gcd_op gcd_impl, flint_rand_t state, slong iters, slong
                      maxn, gr_ctx_t ctx)
```

Tests the given function `gcd_impl` for correctness as an implementation of `_gr_poly_gcd()`.

```
void _gr_poly_test_xgcd(gr_method_poly_xgcd_op xgcd_impl, flint_rand_t state, slong iters,
                       slong maxn, gr_ctx_t ctx)
```

Tests the given function `xgcd_impl` for correctness as an implementation of `_gr_poly_xgcd()`.

```
void _gr_poly_test_approx_mulmid_pos_entrywise_accurate(gr_method_poly_binary_trunc2_op
mulmid_impl,
gr_method_poly_binary_trunc2_op
mulmid_ref, gr_srcptr rel_tol,
flint_rand_t state, slong iters, slong
maxn, gr_ctx_t ctx)
```

Tests the given implementation of polynomial multiplication for accuracy over an approximate numerical ring by generating nonnegative polynomials and checking that the entrywise relative error compared to the reference implementation does not exceed *rel_tol*.

3.9 gr_mpoly.h – sparse multivariate polynomials over generic rings

This module implements multivariate polynomials with `gr` coefficients.

3.9.1 Types, macros and constants

type `gr_mpoly_struct`

type `gr_mpoly_t`

Represents a multivariate polynomial $f \in R[X_1, \dots, X_n]$ as an array of coefficients in a generic ring R together with an array of packed exponents. The two arrays always have the same length.

A `gr_mpoly_t` is always normalised by removing zero coefficients. For rings without decidable equality (e.g. rings with inexact representation), only coefficients that are provably zero will be removed, and there can thus be spurious zeros in the internal representation. For example, with ball coefficients one can have the polynomial $3xy^2 + [\pm 0.01]xy$. Over rings with this issue, the represented lengths or degrees are thus upper bounds, and methods that depend on knowing the exact term structure of a polynomial will return `GR_UNABLE` when encountering such input.

A `gr_mpoly_t` is defined as an array of length one of type `gr_mpoly_struct`, permitting a `gr_mpoly_t` to be passed by reference.

type `gr_mpoly_ctx_struct`

type `gr_mpoly_ctx_t`

Context object representing a multivariate polynomial ring $R[X_1, \dots, X_n]$. This subtypes `gr_ctx_t`, allowing generic `gr` and `gr_ctx` methods to be used interchangeably with `gr_mpoly` and `gr_mpoly_ctx` methods. For example, `gr_add()` with `gr_mpoly_t` and `gr_mpoly_ctx_t` arguments is equivalent to `gr_mpoly_add()`. A context object contains the following data:

- A pointer `mctx` to a `gr_ctx_t` representing the coefficient type R . The coefficient context object is not considered owned by the `gr_mpoly_ctx` and the user must ensure that it stays alive as long as the `gr_mpoly_ctx` is alive.
- A pointer `cctx` to a `mpoly_ctx_t` defining the number of variables and term ordering. This object is considered owned by and will automatically be initialized and cleared along with the `gr_mpoly_ctx`.
- An optional pointer `vars` to an array of strings specifying names of the generators $X_1, \dots, X_n$. This can be set with `gr_mpoly_ctx_set_gen_names()`. By default, `vars` will be initialized to NULL in which case some default names are used. Names are used for printing and parsing from strings with `gr_set_str()` and in some cases for coercions between different rings.

`GR_MPOLY_MCTX`(ctx)

Access the mpoly context object `mctx`.

`GR_MPOLY_CCTX`(ctx)

Access the coefficient context object `cctx`.

`GR_MPOLY_VARS`(ctx)

Access the array of variable names `vars`.

`GR_MPOLY_NVARS`(ctx)

Access the number of variables of this context object.

3.9.2 Context object methods

void **gr_mpoly_ctx_init**(*gr_mpoly_ctx_t* ctx, *gr_ctx_t* base_ring, *slong* nvars, const *ordering_t* ord)

Initializes *ctx* to represent a polynomial ring with coefficients in *base_ring*, with *nvars* variables and term ordering *ord*.

void **gr_mpoly_ctx_clear**(*gr_mpoly_ctx_t* ctx)

Clears the context object *ctx*.

void **gr_mpoly_ctx_init_rand**(*gr_mpoly_ctx_t* ctx, *flint_rand_t* state, *gr_ctx_t* base_ring, *slong* max_nvars)

Initializes *ctx* with a random number of variables up to *max_nvars* inclusive and with a random term ordering.

The following methods implement parts of the standard interface for **gr** context objects.

int **gr_mpoly_ctx_set_gen_names**(*gr_mpoly_ctx_t* ctx, const char **s)

Sets the names of the generators to the strings in *s*.

int **gr_mpoly_ctx_write**(*gr_stream_t* out, *gr_mpoly_ctx_t* ctx)

truth_t **gr_mpoly_ctx_is_ring**(*gr_mpoly_ctx_t* ctx)

truth_t **gr_mpoly_ctx_is_zero_ring**(*gr_mpoly_ctx_t* ctx)

truth_t **gr_mpoly_ctx_is_commutative_ring**(*gr_mpoly_ctx_t* ctx)

truth_t **gr_mpoly_ctx_is_integral_domain**(*gr_mpoly_ctx_t* ctx)

truth_t **gr_mpoly_ctx_is_field**(*gr_mpoly_ctx_t* ctx)

truth_t **gr_mpoly_ctx_is_threadsafe**(*gr_mpoly_ctx_t* ctx)

3.9.3 Memory management

void **gr_mpoly_init**(*gr_mpoly_t* A, *gr_mpoly_ctx_t* ctx)

Initializes and sets *A* to the zero polynomial.

void **gr_mpoly_init3**(*gr_mpoly_t* A, *slong* alloc, *flint_bitcnt_t* bits, *gr_mpoly_ctx_t* ctx)

void **gr_mpoly_init2**(*gr_mpoly_t* A, *slong* alloc, *gr_mpoly_ctx_t* ctx)

Initializes *A* with space allocated for the given number of coefficients and exponents with the given number of bits.

void **gr_mpoly_clear**(*gr_mpoly_t* A, *gr_mpoly_ctx_t* ctx)

Clears *A*, freeing all allocated data.

3.9.4 Basic manipulation

void **_gr_mpoly_normalise**(*gr_mpoly_t* A, *gr_mpoly_ctx_t* ctx)

Removes provably zero coefficients from *A* and updates the length. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **gr_mpoly_swap**(*gr_mpoly_t* A, *gr_mpoly_t* B, *gr_mpoly_ctx_t* ctx)

Swaps *A* and *B* efficiently.

void **gr_mpoly_set_shallow**(*gr_mpoly_t* A, const *gr_mpoly_t* B, *gr_mpoly_ctx_t* ctx)

Sets *A* to a shallow copy of *B* (unsafe).

int **gr_mpoly_set**(*gr_mpoly_t* A, const *gr_mpoly_t* B, *gr_mpoly_ctx_t* ctx)

Sets *A* to *B*.

int **gr_mpoly_zero**(*gr_mpoly_t* A, *gr_mpoly_ctx_t* ctx)

Sets A to the zero polynomial.

truth_t **gr_mpoly_is_zero**(const *gr_mpoly_t* A, *gr_mpoly_ctx_t* ctx)

Returns whether A is the zero polynomial.

slong **gr_mpoly_length**(const *gr_mpoly_t* A, *gr_mpoly_ctx_t* ctx)

Returns the number of terms in A .

3.9.5 Generators

int **gr_mpoly_gen**(*gr_mpoly_t* A, *slong* var, *gr_mpoly_ctx_t* ctx)

Sets A to the generator with index var (indexed from zero).

truth_t **gr_mpoly_is_gen**(const *gr_mpoly_t* A, *slong* var, *gr_mpoly_ctx_t* ctx)

Returns whether A is the generator with index var (indexed from zero).

int **gr_mpoly_gens**(*gr_vec_t* res, *gr_mpoly_ctx_t* ctx)

Sets the vector res to a list of the generators $X_1, \dots, X_n$.

int **gr_mpoly_gens_recursive**(*gr_vec_t* vec, *gr_mpoly_ctx_t* ctx)

Sets the vector res to a list of the recursive generators of R (as constant elements of $R[X_1, \dots, X_n]$) followed by the generators $X_1, \dots, X_n$.

3.9.6 Conversions

int **gr_mpoly_set_scalar**(*gr_mpoly_t* A, *gr_srcptr* c, *gr_mpoly_ctx_t* ctx)

int **gr_mpoly_set_ui**(*gr_mpoly_t* A, *ulong* c, *gr_mpoly_ctx_t* ctx)

int **gr_mpoly_set_si**(*gr_mpoly_t* A, *slong* c, *gr_mpoly_ctx_t* ctx)

int **gr_mpoly_set_fmpz**(*gr_mpoly_t* A, const *fmpz_t* c, *gr_mpoly_ctx_t* ctx)

int **gr_mpoly_set_fmpq**(*gr_mpoly_t* A, const *fmpq_t* c, *gr_mpoly_ctx_t* ctx)

Sets A to the given scalar c .

int **gr_mpoly_set_other**(*gr_mpoly_t* res, *gr_srcptr* A, *gr_ctx_t* A_ctx, *gr_mpoly_ctx_t* ctx)

Sets res to A (an element of A_ctx) converted to the multivariate polynomial ring ctx .

If A_ctx is a multivariate polynomial ring, this attempts to coerce the coefficients and translate the generators. If both rings have named generators, we find all used generators in A and match them to generators with the same names in ctx . If both rings have the same number of unnamed generators and the same term ordering, we perform a direct conversion. Other cases are not currently supported.

Otherwise, we attempt to interpret A as a scalar.

Currently, absorbing generators from nested rings is not supported, e.g. converting between $R[x, y][s, t]$ and $R[x, y, s, t]$ is likely to fail.

3.9.7 Comparisons

truth_t **gr_mpoly_equal**(const *gr_mpoly_t* A, const *gr_mpoly_t* B, *gr_mpoly_ctx_t* ctx)

Returns whether A and B are equal.

3.9.8 Random generation

```
int gr_mpoly_randtest_bits(gr_mpoly_t A, flint_rand_t state, slong length, flint_bitcnt_t
                        exp_bits, gr_mpoly_ctx_t ctx)
```

Sets A to a random polynomial with up to $length$ terms and up to exp_bits bits in the exponents.

3.9.9 Input and output

Note: `gr_set_str()` can be used for parsing.

```
int gr_mpoly_write_pretty(gr_stream_t out, const gr_mpoly_t A, gr_mpoly_ctx_t ctx)
int gr_mpoly_print_pretty(const gr_mpoly_t A, gr_mpoly_ctx_t ctx)
```

Prints A , using the variable names stored in the context.

3.9.10 Coefficient and exponent access

```
int gr_mpoly_get_coeff_scalar_fmpz(gr_ptr c, const gr_mpoly_t A, const fmpz_t *exp,
                                gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_get_coeff_scalar_ui(gr_ptr c, const gr_mpoly_t A, const ulong_t *exp,
                                gr_mpoly_ctx_t ctx)
```

Sets c to the coefficient in A with exponents exp .

```
int gr_mpoly_set_coeff_scalar_fmpz(gr_mpoly_t A, gr_srcptr c, const fmpz_t *exp,
                                gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_ui_fmpz(gr_mpoly_t A, ulong_t c, const fmpz_t *exp, gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_si_fmpz(gr_mpoly_t A, slong_t c, const fmpz_t *exp, gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_fmpz_fmpz(gr_mpoly_t A, const fmpz_t c, const fmpz_t *exp,
                                gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_fmpz_fmpq(gr_mpoly_t A, const fmpq_t c, const fmpz_t *exp,
                                gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_scalar_ui(gr_mpoly_t poly, gr_srcptr c, const ulong_t *exp,
                                gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_ui_ui(gr_mpoly_t A, ulong_t c, const ulong_t *exp, gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_si_ui(gr_mpoly_t A, slong_t c, const ulong_t *exp, gr_mpoly_ctx_t ctx)
```

```
int gr_mpoly_set_coeff_fmpz_ui(gr_mpoly_t A, const fmpz_t c, const ulong_t *exp, gr_mpoly_ctx_t
                                ctx)
```

```
int gr_mpoly_set_coeff_fmpq_ui(gr_mpoly_t A, const fmpq_t c, const ulong_t *exp, gr_mpoly_ctx_t
                                ctx)
```

Sets the coefficient with exponents exp in A to the scalar c which must be an element of or coercible to the coefficient ring.

3.9.11 Arithmetic

```
int gr_mpoly_neg(gr_mpoly_t A, const gr_mpoly_t B, gr_mpoly_ctx_t ctx)
```

Sets A to the negation of B .

```
int gr_mpoly_add(gr_mpoly_t A, const gr_mpoly_t B, const gr_mpoly_t C, gr_mpoly_ctx_t ctx)
```

Sets A to the difference of B and C .

int `gr_mpoly_sub`(*gr_mpoly_t* A, const *gr_mpoly_t* B, const *gr_mpoly_t* C, *gr_mpoly_ctx_t* ctx)
Sets *A* to the difference of *B* and *C*.

int `gr_mpoly_mul`(*gr_mpoly_t* A, const *gr_mpoly_t* B, const *gr_mpoly_t* C, *gr_mpoly_ctx_t* ctx)

int `gr_mpoly_mul_johnson`(*gr_mpoly_t* A, const *gr_mpoly_t* B, const *gr_mpoly_t* C,
gr_mpoly_ctx_t ctx)

int `gr_mpoly_mul_monomial`(*gr_mpoly_t* A, const *gr_mpoly_t* B, const *gr_mpoly_t* C,
gr_mpoly_ctx_t ctx)

Sets *A* to the product of *B* and *C*. The *monomial* version assumes that *C* is a monomial.

int `gr_mpoly_mul_scalar`(*gr_mpoly_t* A, const *gr_mpoly_t* B, *gr_srcptr* c, *gr_mpoly_ctx_t* ctx)

int `gr_mpoly_mul_si`(*gr_mpoly_t* A, const *gr_mpoly_t* B, *slong* c, *gr_mpoly_ctx_t* ctx)

int `gr_mpoly_mul_ui`(*gr_mpoly_t* A, const *gr_mpoly_t* B, *ulong* c, *gr_mpoly_ctx_t* ctx)

int `gr_mpoly_mul_fmpz`(*gr_mpoly_t* A, const *gr_mpoly_t* B, const *fmpz_t* c, *gr_mpoly_ctx_t* ctx)

int `gr_mpoly_mul_fmpq`(*gr_mpoly_t* A, const *gr_mpoly_t* B, const *fmpq_t* c, *gr_mpoly_ctx_t* ctx)

Sets *A* to *B* multiplied by the scalar *c* which must be an element of or coercible to the coefficient ring.

int `gr_mpoly_inv`(*gr_mpoly_t* res, const *gr_mpoly_t* src, *gr_mpoly_ctx_t* ctx)

3.9.12 Derivative and integral

int `gr_mpoly_derivative`(*gr_mpoly_t* A, const *gr_mpoly_t* B, *slong* var, *gr_mpoly_ctx_t* ctx)

Set *A* to the derivative of *B* with respect to the variable of index *var*.

int `gr_mpoly_integral`(*gr_mpoly_t* A, const *gr_mpoly_t* B, *slong* var, *gr_mpoly_ctx_t* ctx)

Set *A* to the integral of *B* with respect to the variable of index *var*.

3.9.13 Other operations

int `gr_mpoly_canonical_associate`(*gr_mpoly_t* res, *gr_mpoly_t* u, const *gr_mpoly_t* src,
gr_mpoly_ctx_t ctx)

3.9.14 Container operations

Mostly intended for internal use.

void `_gr_mpoly_fit_length`(*gr_ptr* *coeffs, *slong* *coeffs_alloc, *ulong* **exps, *slong* *exps_alloc,
slong N, *slong* length, *gr_mpoly_ctx_t* ctx)

void `gr_mpoly_fit_length`(*gr_mpoly_t* A, *slong* len, *gr_mpoly_ctx_t* ctx)

Ensures that *A* has space for *len* coefficients and exponents.

void `gr_mpoly_fit_bits`(*gr_mpoly_t* A, *flint_bitcnt_t* bits, *gr_mpoly_ctx_t* ctx)

void `gr_mpoly_fit_length_fit_bits`(*gr_mpoly_t* A, *slong* len, *flint_bitcnt_t* bits, *gr_mpoly_ctx_t* ctx)

void `gr_mpoly_fit_length_reset_bits`(*gr_mpoly_t* A, *slong* len, *flint_bitcnt_t* bits,
gr_mpoly_ctx_t ctx)

void `_gr_mpoly_set_length`(*gr_mpoly_t* A, *slong* newlen, *gr_mpoly_ctx_t* ctx)

void `_gr_mpoly_push_exp_ui`(*gr_mpoly_t* A, const *ulong* *exp, *gr_mpoly_ctx_t* ctx)

```
int gr_mpoly_push_term_scalar_ui(gr_mpoly_t A, gr_srcptr c, const ulong *exp, gr_mpoly_ctx_t
                                ctx)

void _gr_mpoly_push_exp_fmpz(gr_mpoly_t A, const fmpz *exp, gr_mpoly_ctx_t ctx)

int gr_mpoly_push_term_scalar_fmpz(gr_mpoly_t A, gr_srcptr c, const fmpz *exp,
                                   gr_mpoly_ctx_t ctx)

void gr_mpoly_sort_terms(gr_mpoly_t A, gr_mpoly_ctx_t ctx)

int gr_mpoly_combine_like_terms(gr_mpoly_t A, gr_mpoly_ctx_t ctx)

truth_t gr_mpoly_is_canonical(const gr_mpoly_t A, gr_mpoly_ctx_t ctx)

void gr_mpoly_assert_canonical(const gr_mpoly_t A, gr_mpoly_ctx_t ctx)
```

3.10 `gr_series.h` – formal power series over generic rings

We provide two implementations of formal power series:

- `gr_series_mod` - truncated power series $R[[x]]/x^n$, with exact representation of elements.
- `gr_series` - infinite power series $R[[x]]$, with elements represented as finite truncations with an error term $O(x^n)$ where n is tracked per-element. Polynomials of small degree can be represented exactly without error term.

It should be noted that these are different algebraic structures. For example, $\mathbb{Q}[[x]]$ is an integral domain, but $\mathbb{Q}[[x]]/x^n$ is not. Some operations are admissible for `gr_series` but not for `gr_series_mod`, and vice versa. Note that for `gr_equal()`, $0 = 0$ is true in both $R[[x]]$ and $R[[x]]/x^n$, but $0 + O(x^n) = 0 + O(x^n)$ (or even $0 + O(x^n) = 0$) is unknown in $R[[x]]$ since we do not know whether the big-O term hides some nonzero term. For a computation where either representation may be used, it is recommended to choose `gr_series_mod` as this will have less overhead.

3.10.1 Context constructors

`void gr_series_mod_ctx_init(gr_ctx_t ctx, gr_ctx_t base_ring, slong prec)`

Initializes *ctx* to a ring of truncated power series $R[[x]]/\langle x^n \rangle$ over the given *base_ring*. Elements have type `gr_poly_t`. It is assumed that all inputs are already truncated to length n , and this invariant is enforced for all outputs.

The truncation order n of a `gr_series_mod` context is fixed. If $m \leq n$, then elements belonging to a context object for $R[[x]]/x^m$ can be given as input to a context for $R[[x]]/x^n$, but not vice versa.

`void gr_series_ctx_init(gr_ctx_t ctx, gr_ctx_t base_ring, slong prec)`

Initializes *ctx* to a ring of power series $R[[x]]$ over the given *base_ring*. Elements have type `gr_series_t`.

Elements are generally inexact, having an error term $O(x^n)$ where n may be different for each element. The context parameter *prec* defines the default precision: tails $c_n x^n + c_{n+1} x^{n+1} + \dots$ with $n \geq \text{prec}$ will be replaced by the error term $O(x^n)$.

The default precision is mutable. Elements can be shared freely between two context objects with the same base ring, even if those context objects have different precision.

3.10.2 Truncated power series

Types and macros

`GR_SERIES_MOD_ELEM_CTX(ctx)`

Given a `gr_series_mod` context object, accesses the context object of the base ring.

`GR_SERIES_MOD_N(ctx)`

Given a `gr_series_mod` context object, accesses the truncation order n .

Generic methods

Elements of a `gr_series_mod` ring can be manipulated using the standard `gr` interface. The following is a list of implemented functions such that `gr_series_mod_foo` overloads `gr_foo` in the method table for `gr_series_mod`; the user can optionally call `gr_series_mod_foo` directly.

```
void gr_series_mod_ctx_clear(gr_ctx_t ctx)
int gr_series_mod_ctx_write(gr_stream_t out, gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_ring(gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_commutative_ring(gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_integral_domain(gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_rational_vector_space(gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_real_vector_space(gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_complex_vector_space(gr_ctx_t ctx)
truth_t gr_series_mod_ctx_is_field(gr_ctx_t ctx)
int gr_series_mod_ctx_set_gen_name(gr_ctx_t ctx, const char *s)
int gr_series_mod_ctx_set_gen_names(gr_ctx_t ctx, const char **s)
int gr_series_mod_gens_recursive(gr_vec_t vec, gr_ctx_t ctx)
void gr_series_mod_init(gr_poly_t res, gr_ctx_t ctx)
void gr_series_mod_clear(gr_poly_t res, gr_ctx_t ctx)
void gr_series_mod_swap(gr_poly_t x, gr_poly_t y, gr_ctx_t ctx)
int gr_series_mod_randtest(gr_poly_t res, flint_rand_t state, gr_ctx_t ctx)
int gr_series_mod_write(gr_stream_t out, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_zero(gr_poly_t res, gr_ctx_t ctx)
int gr_series_mod_one(gr_poly_t res, gr_ctx_t ctx)
int gr_series_mod_gen(gr_poly_t res, gr_ctx_t ctx)
int gr_series_mod_set(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_set_other(gr_poly_t res, gr_srcptr_t x, gr_ctx_t x_ctx, gr_ctx_t ctx)
void gr_series_mod_set_shallow(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_set_si(gr_poly_t res, slong c, gr_ctx_t ctx)
int gr_series_mod_set_ui(gr_poly_t res, ulong c, gr_ctx_t ctx)
int gr_series_mod_set_fmpz(gr_poly_t res, const fmpz_t c, gr_ctx_t ctx)
int gr_series_mod_set_fmpq(gr_poly_t res, const fmpq_t c, gr_ctx_t ctx)
truth_t gr_series_mod_is_zero(const gr_poly_t x, gr_ctx_t ctx)
truth_t gr_series_mod_is_one(const gr_poly_t x, gr_ctx_t ctx)
truth_t gr_series_mod_equal(const gr_poly_t x, const gr_poly_t y, gr_ctx_t ctx)
int gr_series_mod_neg(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_add(gr_poly_t res, const gr_poly_t x, const gr_poly_t y, gr_ctx_t ctx)
int gr_series_mod_sub(gr_poly_t res, const gr_poly_t x, const gr_poly_t y, gr_ctx_t ctx)
int gr_series_mod_mul(gr_poly_t res, const gr_poly_t x, const gr_poly_t y, gr_ctx_t ctx)
int gr_series_mod_inv(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_div(gr_poly_t res, const gr_poly_t x, const gr_poly_t y, gr_ctx_t ctx)
int gr_series_mod_exp(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_log(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_sqrt(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_rsqr(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_tan(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_asin(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_acos(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
```

```
int gr_series_mod_atan(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_asinh(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_acosh(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_mod_atanh(gr_poly_t res, const gr_poly_t x, gr_ctx_t ctx)
```

3.10.3 Power series

Types and macros

type **gr_series_struct**

type **gr_series_t**

A structure containing a *gr_poly_t* followed by a *slong* representing the exponent in the error term.

type **gr_series_vec_struct**

type **gr_series_vec_t**

Alias for *gr_vec_t* with *gr_series_t* elements, provided for convenience.

GR_SERIES_POLY(x)

Macro accessing the polynomial part of a *gr_series_t*.

GR_SERIES_ERROR(x)

Macro accessing the error of a *gr_series_t*.

GR_SERIES_ERR_MAX

The maximum allowed n in an error term $O(x^n)$.

GR_SERIES_ERR_EXACT

A special value of n used to indicate that a series is exact.

GR_SERIES_ELEM_CTX(ctx)

Given a *gr_series* context object, accesses the context object of the base ring.

GR_SERIES_PREC(ctx)

Given a *gr_series* context object, accesses the default precision.

Error term manipulation

slong **_gr_series_get_error**(const *gr_series_t* f, *gr_ctx_t* ctx)

Return the exponent n of the error term $O(x^n)$ of f . If f is exact, returns **GR_SERIES_ERR_EXACT**.

truth_t **_gr_series_is_exact**(const *gr_series_t* f, *gr_ctx_t* ctx)

Returns whether f is exact as a power series, i.e. lacks error term $O(x^n)$. This does not recursively check exactness of the underlying elements.

void **_gr_series_set_error**(*gr_series_t* f, *slong* n, *gr_ctx_t* ctx)

Add an error term $O(x^n)$ in-place to f . The exponent n is clamped between 0 and **GR_SERIES_ERR_MAX**. Terms of order higher than or equal to n will be truncated.

void **_gr_series_make_exact**(*gr_series_t* f, *gr_ctx_t* ctx)

Remove the $O(x^n)$ error term (if any) from f .

Generic methods

Elements of a `gr_series` ring can be manipulated using the standard `gr` interface. The following is a list of implemented functions such that `gr_series_foo` overloads `gr_foo` in the method table for `gr_series`; the user can optionally call `gr_series_foo` directly.

```
void gr_series_ctx_clear(gr_ctx_t ctx)
int gr_series_ctx_write(gr_stream_t out, gr_ctx_t ctx)
truth_t gr_series_ctx_is_ring(gr_ctx_t ctx)
truth_t gr_series_ctx_is_commutative_ring(gr_ctx_t ctx)
truth_t gr_series_ctx_is_integral_domain(gr_ctx_t ctx)
truth_t gr_series_ctx_is_rational_vector_space(gr_ctx_t ctx)
truth_t gr_series_ctx_is_real_vector_space(gr_ctx_t ctx)
truth_t gr_series_ctx_is_complex_vector_space(gr_ctx_t ctx)
int gr_series_ctx_set_gen_name(gr_ctx_t ctx, const char *s)
int gr_series_ctx_set_gen_names(gr_ctx_t ctx, const char **s)
void gr_series_init(gr_series_t res, gr_ctx_t ctx)
void gr_series_clear(gr_series_t res, gr_ctx_t ctx)
void gr_series_swap(gr_series_t x, gr_series_t y, gr_ctx_t ctx)
void gr_series_set_shallow(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_randtest(gr_series_t res, flint_rand_t state, gr_ctx_t ctx)
int gr_series_write(gr_stream_t out, const gr_series_t x, gr_ctx_t ctx)
int gr_series_zero(gr_series_t res, gr_ctx_t ctx)
int gr_series_one(gr_series_t res, gr_ctx_t ctx)
int gr_series_set(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_gen(gr_series_t res, gr_ctx_t ctx)
int gr_series_gens_recursive(gr_vec_t vec, gr_ctx_t ctx)
int gr_series_neg(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_set_gr_poly(gr_series_t res, const gr_poly_t x, gr_ctx_t ctx)
int gr_series_set_scalar(gr_series_t res, gr_srcptr x, gr_ctx_t ctx)
int gr_series_set_si(gr_series_t res, slong c, gr_ctx_t ctx)
int gr_series_set_ui(gr_series_t res, ulong c, gr_ctx_t ctx)
int gr_series_set_fmpz(gr_series_t res, const fmpz_t c, gr_ctx_t ctx)
int gr_series_set_fmpq(gr_series_t res, const fmpq_t c, gr_ctx_t ctx)
int gr_series_set_other(gr_series_t res, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t ctx)
truth_t gr_series_is_zero(const gr_series_t x, gr_ctx_t ctx)
truth_t gr_series_is_one(const gr_series_t x, gr_ctx_t ctx)
truth_t gr_series_coeff_is_zero(const gr_series_t x, slong i, gr_ctx_t ctx)
truth_t gr_series_equal(const gr_series_t x, const gr_series_t y, gr_ctx_t ctx)
int gr_series_add(gr_series_t res, const gr_series_t x, const gr_series_t y, gr_ctx_t ctx)
int gr_series_sub(gr_series_t res, const gr_series_t x, const gr_series_t y, gr_ctx_t ctx)
int gr_series_mul(gr_series_t res, const gr_series_t x, const gr_series_t y, gr_ctx_t ctx)
int gr_series_inv(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_div(gr_series_t res, const gr_series_t x, const gr_series_t y, gr_ctx_t ctx)
int gr_series_divexact(gr_series_t res, const gr_series_t x, const gr_series_t y, gr_ctx_t ctx)
int gr_series_sqrt(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_rsqrtn(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_exp(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_log(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
```



```
int gr_series_tan(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_asin(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_acos(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_atan(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_asinh(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_acosh(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_atanh(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_gamma(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_rgamma(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_lgamma(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_digamma(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_erf(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_erfc(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_erfi(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_exp_integral_ei(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_cos_integral(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_cosh_integral(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_sin_integral(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_sinh_integral(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_fresnel(gr_series_t res1, gr_series_t res2, const gr_series_t x, int normalized,
                      gr_ctx_t ctx)
int gr_series_fresnel_s(gr_series_t res, const gr_series_t x, int normalized, gr_ctx_t ctx)
int gr_series_fresnel_c(gr_series_t res, const gr_series_t x, int normalized, gr_ctx_t ctx)
int gr_series_airy(gr_series_t res1, gr_series_t res2, gr_series_t res3, gr_series_t res4, const
                  gr_series_t x, gr_ctx_t ctx)
int gr_series_airy_ai(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_airy_ai_prime(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_airy_bi(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_airy_bi_prime(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)
int gr_series_log_integral(gr_series_t res, const gr_series_t x, int offset, gr_ctx_t ctx)
int gr_series_gamma_upper(gr_series_t res, const gr_series_t s, const gr_series_t x, int
                          regularized, gr_ctx_t ctx)
int gr_series_gamma_lower(gr_series_t res, const gr_series_t s, const gr_series_t x, int
                          regularized, gr_ctx_t ctx)
int gr_series_beta_lower(gr_series_t res, const gr_series_t a, const gr_series_t b, const
                        gr_series_t x, int regularized, gr_ctx_t ctx)
int gr_series_polylog(gr_series_t res, const gr_series_t s, const gr_series_t z, gr_ctx_t ctx)
int gr_series_hurwitz_zeta(gr_series_t res, const gr_series_t s, const gr_series_t z, gr_ctx_t
                          ctx)
int gr_series_dirichlet_l(gr_series_t res, const dirichlet_group_t G, const dirichlet_char_t chi,
                        const gr_series_t x, gr_ctx_t ctx)
int gr_series_dirichlet_hardy_theta(gr_series_t res, const dirichlet_group_t G, const
                                   dirichlet_char_t chi, const gr_series_t x, gr_ctx_t ctx)
int gr_series_dirichlet_hardy_z(gr_series_t res, const dirichlet_group_t G, const
                                dirichlet_char_t chi, const gr_series_t x, gr_ctx_t ctx)
int gr_series_jacobi_theta(gr_series_t res1, gr_series_t res2, gr_series_t res3, gr_series_t res4,
                          const gr_series_t x, const gr_series_t tau, gr_ctx_t ctx)
int gr_series_jacobi_theta_1(gr_series_t res, const gr_series_t x, const gr_series_t tau,
                             gr_ctx_t ctx)
```



```
int gr_series_jacobi_theta_2(gr_series_t res, const gr_series_t x, const gr_series_t tau,  
                             gr_ctx_t ctx)  
int gr_series_jacobi_theta_3(gr_series_t res, const gr_series_t x, const gr_series_t tau,  
                             gr_ctx_t ctx)  
int gr_series_jacobi_theta_4(gr_series_t res, const gr_series_t x, const gr_series_t tau,  
                             gr_ctx_t ctx)  
int gr_series_agm1(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)  
int gr_series_elliptic_k(gr_series_t res, const gr_series_t x, gr_ctx_t ctx)  
int gr_series_weierstrass_p(gr_series_t res, const gr_series_t x, const gr_series_t tau, gr_ctx_t  
                           ctx)  
int gr_series_hypgeom_pfq(gr_series_t res, const gr_series_vec_t a, const gr_series_vec_t b,  
                          const gr_series_t x, int regularized, gr_ctx_t ctx)
```


INTEGERS

4.1 `ulong_extras.h` – arithmetic and number-theoretic functions for single-word integers

This module implements functions for single limb unsigned integers, including arithmetic with a precomputed inverse and modular arithmetic.

The module includes functions for square roots, factorisation and primality testing. Almost all the functions in this module are highly developed and extremely well optimised.

The basic type is the `ulong` as defined by GMP. Functions which take a precomputed inverse either have the suffix `preinv` and take an `ulong` precomputed inverse as computed by `n_preinvert_limb` or have the suffix `_precomp` and accept a `double` precomputed inverse as computed by `n_precompute_inverse`.

Sometimes three functions with similar names are provided for the same task, e.g. `n_mod_precomp`, `n_mod2_precomp` and `n_mod2_preinv`. If the part of the name that designates the functionality ends in 2 then the function has few if any limitations on its inputs. Otherwise the function may have limitations such as being limited to 52 or 53 bits. In practice we found that the `preinv` functions are generally faster anyway, so most times it pays to just use the `n_blah2_preinv` variants.

Some functions with the `n_11_` or `n_111_` prefix accept parameters of two or three limbs respectively.

4.1.1 Simple example

The following example computes $ab \pmod n$ using a precomputed inverse, where $a = 12345678$, $b = 87654321$ and $n = 111111111$.

```
#include <stdio.h>
#include "ulong_extras.h"
int main()
{
    ulong r, a, b, n, ninv;

    a = UWORD(12345678);
    b = UWORD(87654321);
    n = UWORD(111111111);
    ninv = n_preinvert_limb(n);

    r = n_mulmod2_preinv(a, b, n, ninv);

    flint_printf("%wu*%wu mod %wu is %wu\n", a, b, n, r);
}
```

The output is:

`12345678*87654321 mod 111111111 is 23456790`

4.1.2 Random functions

ulong **n_randlimb**(*flint_rand_t* state)

ulong **_n_randlimb**(*flint_rand_t* state)

Returns a uniformly pseudo random limb.

The algorithm generates two random half limbs s_j , $j = 0, 1$, by iterating respectively $v_{i+1} = (v_i a + b) \bmod p_j$ for some initial seed v_0 , randomly chosen values a and b and $p_0 = 4294967311 = \text{nextprime}(2^{32})$ on a 64-bit machine and $p_0 = \text{nextprime}(2^{16})$ on a 32-bit machine and $p_1 = \text{nextprime}(p_0)$.

The function **_n_randlimb** is defined inline, which can allow for better performance when generating many random numbers.

ulong **n_randbits**(*flint_rand_t* state, unsigned int bits)

Returns a uniformly pseudo random number with the given number of bits. The most significant bit is always set, unless zero is passed, in which case zero is returned.

ulong **n_randtest_bits**(*flint_rand_t* state, int bits)

Returns a uniformly pseudo random number with the given number of bits. The most significant bit is always set, unless zero is passed, in which case zero is returned. The probability of a value with a sparse binary representation being returned is increased. This function is intended for use in test code.

ulong **n_randint**(*flint_rand_t* state, *ulong* limit)

ulong **n_urandint**(*flint_rand_t* state, *ulong* limit)

ulong **_n_randint**(*flint_rand_t* state, *ulong* limit)

Returns a uniformly pseudo random number up to but not including the given limit. If zero is passed as a parameter, an entire random limb is returned.

The functions **n_randint** and **n_urandint** are identical. The function **_n_randint** is defined inline, which can allow for better performance when generating many random numbers.

ulong **n_randtest**(*flint_rand_t* state)

Returns a pseudo random number with a random number of bits, from 0 to FLINT_BITS. The probability of the special values 0, 1, COEFF_MAX and WORD_MAX is increased as is the probability of a value with sparse binary representation. This random function is mainly used for testing purposes. This function is intended for use in test code.

ulong **n_randtest_not_zero**(*flint_rand_t* state)

As for **n_randtest()**, but does not return 0. This function is intended for use in test code.

ulong **n_randprime**(*flint_rand_t* state, *ulong* bits, int proved)

Returns a random prime number (**proved** = 1) or probable prime (**proved** = 0) with **bits** bits, where **bits** must be at least 2 and at most FLINT_BITS.

ulong **n_randtest_prime**(*flint_rand_t* state, int proved)

Returns a random prime number (**proved** = 1) or probable prime (**proved** = 0) with size randomly chosen between 2 and FLINT_BITS bits. This function is intended for use in test code.

4.1.3 Basic arithmetic

ulong **n_pow**(*ulong* n, *ulong* exp)

Returns n^{exp} . No checking is done for overflow. The exponent may be zero. We define $0^0 = 1$.

The algorithm simply uses a for loop. Repeated squaring is unlikely to speed up this algorithm.

ulong **n_flog**(*ulong* n, *ulong* b)

Returns $\lfloor \log_b n \rfloor$.

Assumes that $n \geq 1$ and $b \geq 2$.

ulong **n_clog**(*ulong* n, *ulong* b)

Returns $\lceil \log_b n \rceil$.

Assumes that $n \geq 1$ and $b \geq 2$.

ulong **n_clog_2exp**(*ulong* n, *ulong* b)

Returns $\lceil \log_b 2^n \rceil$.

Assumes that $b \geq 2$.

4.1.4 Miscellaneous

ulong **n_revbin**(*ulong* n, *ulong* b)

Returns the binary reverse of n , assuming it is b bits in length, e.g. **n_revbin**(10110, 6) will return 110100.

int **n_sizeinbase**(*ulong* n, int base)

Returns the exact number of digits needed to represent n as a string in base **base** assumed to be between 2 and 36. Returns 1 when $n = 0$.

4.1.5 Basic arithmetic with precomputed inverses

ulong **n_preinvert_limb_prenorm**(*ulong* n)

Computes an approximate inverse **invx1** of the limb **x1**, with an implicit leading `1``. More formally it computes:

$$\text{invx1} = (B^2 - B \cdot x - 1) / x = (B^2 - 1) / x - B$$

Note that x must be normalised, i.e. with msb set. This inverse makes use of Lemma 8.1 in [GraMon1994]:

Let d be normalised, $d < B$, i.e. it fits in a word, and suppose that $md < B^2 \leq (m+1)d$. Let $0 \leq n \leq Bd - 1$. Write $n = n_2B + n_1B/2 + n_0$ with $n_1 = 0$ or 1 and $n_0 < B/2$. Suppose $q_1B + q_0 = n_2B + (n_2 + n_1)(m - B) + n_1(d - B/2) + n_0$ and $0 \leq q_0 < B$. Then $0 \leq q_1 < B$ and $0 \leq n - q_1d < 2d$.

In the theorem, m is the inverse of d . If we let $m = \text{invx1} + B$ and $d = x$ we have $md = B^2 - 1 < B^2$ and $(m+1)x = B^2 + d - 1 \geq B^2$.

The theorem is often applied as follows: note that n_0 and $n_1(d - B/2)$ are both less than $B/2$. Also note that $n_1(m - B) < B$. Thus the sum of all these terms contributes at most 1 to q_1 . We are left with $n_2B + n_2(m - B)$. But note that $(m - B)$ is precisely our precomputed inverse **invx1**. If we write $q_1B + q_0 = n_2B + n_2(m - B)$, then from the theorem, we have $0 \leq n - q_1d < 3d$, i.e. the quotient is out by at most 2 and is always either correct or too small.

ulong **n_preinvert_limb**(*ulong* n)

Returns a precomputed inverse of n , as defined in [GraMol2010]. This precomputed inverse can be used with all of the functions that take a precomputed inverse whose names are suffixed by `_preinv`.

We require $n > 0$.

double **n_precompute_inverse**(*ulong* n)

Returns a precomputed inverse of n with double precision value $1/n$. This precomputed inverse can be used with all of the functions that take a precomputed inverse whose names are suffixed by `_precomp`.

We require $n > 0$.

ulong **n_mod_precomp**(*ulong* a, *ulong* n, double ninv)

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{\text{FLINT\_D\_BITS}}$ and $a < 2^{(\text{FLINT\_BITS}-1)}$ and $0 \leq a < n^2$.

We assume the processor is in the standard round to nearest mode. Thus `ninv` is correct to 53 binary bits, the least significant bit of which we shall call a place, and can be at most half a place out. When a is multiplied by `ninv`, the binary representation of a is exact and the mantissa is less than 2, thus we see that $a * \text{ninv}$ can be at most one out in the mantissa. We now truncate $a * \text{ninv}$ to the nearest integer, which is always a round down. Either we already have an integer, or we need to make a change down of at least 1 in the last place. In the latter case we either get precisely the exact quotient or below it as when we rounded the product to the nearest place we changed by at most half a place. In the case that truncating to an integer takes us below the exact quotient, we have rounded down by less than 1 plus half a place. But as the product is less than n and n is less than 2^{53} , half a place is less than 1, thus we are out by less than 2 from the exact quotient, i.e. the quotient we have computed is the quotient we are after or one too small. That leaves only the case where we had to round up to the nearest place which happened to be an integer, so that truncating to an integer didn't change anything. But this implies that the exact quotient a/n is less than 2^{-54} from an integer. We deal with this rare case by subtracting 1 from the quotient. Then the quotient we have computed is either exactly what we are after, or one too small.

ulong **n_mod2_precomp**(*ulong* a, *ulong* n, double ninv)

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()`. There are no restrictions on a or on n .

As for `n_mod_precomp()` for $n < 2^{53}$ and $a < n^2$ the computed quotient is either what we are after or one too large or small. We deal with these cases. Otherwise we can be sure that the top 52 bits of the quotient are computed correctly. We take the remainder and adjust the quotient by multiplying the remainder by `ninv` to compute another approximate quotient as per `mod_precomp()`. Now the remainder may be either negative or positive, so the quotient we compute may be one out in either direction.

ulong **n_divrem2_preinv**(*ulong* *q, *ulong* a, *ulong* n, *ulong* ninv)

ulong **n_divrem_preinv**(*ulong* *q, *ulong* a, *ulong* n, *ulong* ninv, unsigned int norm)

ulong **n_divrem_preinv_unnorm**(*ulong* *q, *ulong* a, *ulong* n, *ulong* ninv, unsigned int norm)

Returns $a \bmod n$ and sets q to the quotient of a by n , given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a and the only restriction on n is that it be nonzero.

This uses the algorithm of Granlund and Möller [GraMol2010]. First n is normalised and a is shifted into two limbs to compensate. Then their algorithm is applied verbatim and the remainder shifted back.

The difference between the legacy `n_divrem2_preinv()` and `n_divrem_preinv()` is that the latter accepts the `norm` parameter specifying the number of leading zero bits of n as input and hence avoids computing this on the fly. The version `n_divrem_preinv_unnorm()` requires positive `norm`.

ulong **n_divrem_norm**(*ulong* *q, *ulong* a, *ulong* n)

Returns $a \bmod n$ and sets q to the quotient of a by n , assuming that n is normalised. This is a trivial operation as the quotient is either 0 or 1.

ulong **n_div2_preinv**(*ulong* a, *ulong* n, *ulong* ninv)

Returns the Euclidean quotient of a by n given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a and the only restriction on n is that it be nonzero.

This uses the algorithm of Granlund and Möller [GraMol2010]. First n is normalised and a is shifted into two limbs to compensate. Then their algorithm is applied verbatim.

ulong **n_mod2_preinv**(*ulong* a, *ulong* n, *ulong* ninv)

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a and the only restriction on n is that it be nonzero.

This uses the algorithm of Granlund and Möller [GraMol2010]. First n is normalised and a is shifted into two limbs to compensate. Then their algorithm is applied verbatim and the result shifted back.

ulong **n_divrem2_precomp**(*ulong* *q, *ulong* a, *ulong* n, double npre)

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()` and sets q to the quotient. There are no restrictions on a or on n .

This is as for `n_mod2_precomp()` with some additional care taken to retain the quotient information. There are also special cases to deal with the case where a is already reduced modulo n and where n is 64 bits and a is not reduced modulo n .

ulong **n_ll_mod_preinv**(*ulong* a_hi, *ulong* a_lo, *ulong* n, *ulong* ninv)

Returns $a \bmod n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a , which will be two limbs (a_{hi} , a_{lo}), or on n .

The old version of this function merely reduced the top limb a_{hi} modulo n so that `udiv_qrnnnd_preinv()` could be used.

The new version reduces the top limb modulo n as per `n_mod2_preinv()` and then the algorithm of Granlund and Möller [GraMol2010] is used again to reduce modulo n .

ulong **n_lll_mod_preinv**(*ulong* a_hi, *ulong* a_mi, *ulong* a_lo, *ulong* n, *ulong* ninv)

Returns $a \bmod n$, where a has three limbs (a_{hi} , a_{mi} , a_{lo}), given a precomputed inverse of n computed by `n_preinvert_limb()`. It is assumed that a_{hi} is reduced modulo n . There are no restrictions on n .

This function uses the algorithm of Granlund and Möller [GraMol2010] to first reduce the top two limbs modulo n , then does the same on the bottom two limbs.

ulong **n_mulmod_precomp**(*ulong* a, *ulong* b, *ulong* n, double ninv)

Returns $ab \bmod n$ given a precomputed inverse of n computed by `n_precompute_inverse()`. We require $n < 2^{\text{FLINT\_D\_BITS}}$ and $0 \leq a, b < n$.

We assume the processor is in the standard round to nearest mode. Thus `ninv` is correct to 53 binary bits, the least significant bit of which we shall call a place, and can be at most half a place out. The product of a and b is computed with error at most half a place. When $a * b$ is multiplied by `ninv` we find that the exact quotient and computed quotient differ by less than two places. As the quotient is less than n this means that the exact quotient is at most 1 away from the computed quotient. We truncate this quotient to an integer which reduces the value by less than 1. We end up with a value which can be no more than two above the quotient we are after and no less than two below. However an argument similar to that for `n_mod_precomp()` shows that the truncated computed quotient cannot be two smaller than the truncated exact quotient. In other words the computed integer quotient is at most two above and one below the quotient we are after.

ulong **n_mulmod2_preinv**(*ulong* a, *ulong* b, *ulong* n, *ulong* ninv)

Returns $ab \bmod n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. There are no restrictions on a , b or on n . This is implemented by multiplying using `umul_ppmm()` and then reducing using `n_ll_mod_preinv()`.

ulong **n_mulmod2**(*ulong* a, *ulong* b, *ulong* n)

Returns $ab \bmod n$. There are no restrictions on a , b or on n . This is implemented by multiplying using `umul_ppmm()` and then reducing using `n_ll_mod_preinv()` after computing a precomputed inverse.

ulong **n_mulmod_preinv**(*ulong* a, *ulong* b, *ulong* n, *ulong* ninv, *ulong* norm)

Returns $ab \pmod n$ given a precomputed inverse of n computed by `n_preinvert_limb()`, assuming a and b are reduced modulo n and n is normalised, i.e. with most significant bit set. There are no other restrictions on a , b or n .

The value `norm` is provided for convenience. As n is required to be normalised, it may be that a and b have been shifted to the left by `norm` bits before calling the function. Their product then has an extra factor of 2^{norm} . Specifying a nonzero `norm` will shift the product right by this many bits before reducing it.

The algorithm used is that of Granlund and Möller [GraMol2010].

4.1.6 Greatest common divisor

ulong **n_gcd**(*ulong* x, *ulong* y)

Returns the greatest common divisor g of x and y . No assumptions are made about the values x and y .

This function wraps GMP's `mpn_gcd_1`.

ulong **n_gcdinv**(*ulong* *a, *ulong* x, *ulong* y)

Returns the greatest common divisor g of x and y and computes a such that $0 \leq a < y$ and $ax = \gcd(x, y) \bmod y$, when this is defined. We require $x < y$.

When $y = 1$ the greatest common divisor is set to 1 and a is set to 0.

This is merely an adaption of the extended Euclidean algorithm computing just one cofactor and reducing it modulo y .

ulong **n_xgcd**(*ulong* *a, *ulong* *b, *ulong* x, *ulong* y)

Returns the greatest common divisor g of x and y and unsigned values a and b such that $ax - by = g$. We require $x \geq y$.

We claim that computing the extended greatest common divisor via the Euclidean algorithm always results in cofactor $|a| < x/2$, $|b| < x/2$, with perhaps some small degenerate exceptions.

We proceed by induction.

Suppose we are at some step of the algorithm, with $x_n = qy_n + r$ with $r \geq 1$, and suppose $1 = sy_n - tr$ with $s < y_n/2$, $t < y_n/2$ by hypothesis.

Write $1 = sy_n - t(x_n - qy_n) = (s + tq)y_n - tx_n$.

It suffices to show that $(s + tq) < x_n/2$ as $t < y_n/2 < x_n/2$, which will complete the induction step.

But at the previous step in the backsubstitution we would have had $1 = sr - cd$ with $s < r/2$ and $c < r/2$.

Then $s + tq < r/2 + y_n/2q = (r + qy_n)/2 = x_n/2$.

See the documentation of `n_gcd()` for a description of the branching in the algorithm, which is faster than using division.

4.1.7 Jacobi and Kronecker symbols

int **n_jacobi**(*slong* x, *ulong* y)

Computes the Jacobi symbol $\left(\frac{x}{y}\right)$ for any x and odd y .

int **n_jacobi_unsigned**(*ulong* x, *ulong* y)

Computes the Jacobi symbol, allowing x to go up to a full limb.

4.1.8 Modular arithmetic

ulong **n_addmod**(*ulong* a, *ulong* b, *ulong* n)

Returns $(a + b) \bmod n$.

ulong **n_submod**(*ulong* a, *ulong* b, *ulong* n)

Returns $(a - b) \bmod n$.

ulong **n_invmod**(*ulong* x, *ulong* y)

Returns the inverse of x modulo y , if it exists. Otherwise an exception is thrown.

This is merely an adaption of the extended Euclidean algorithm with appropriate normalisation.

ulong **n_binvert**(*ulong* n)

Assuming that n is odd, returns the multiplicative inverse modulo $2^{\text{FLINT\_BITS}}$. If n is even, some arbitrary value is returned.

ulong **n_powmod_precomp**(*ulong* a, *slong* exp, *ulong* n, double npre)

Returns a^{exp} modulo n given a precomputed inverse of n computed by **n_precompute_inverse()**. We require $n < 2^{53}$ and $0 \leq a < n$. There are no restrictions on **exp**, i.e. it can be negative.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

ulong **n_powmod_ui_precomp**(*ulong* a, *ulong* exp, *ulong* n, double npre)

Returns a^{exp} modulo n given a precomputed inverse of n computed by **n_precompute_inverse()**. We require $n < 2^{53}$ and $0 \leq a < n$. The exponent **exp** is unsigned and so can be larger than allowed by **n_powmod_precomp()**.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

ulong **n_powmod**(*ulong* a, *slong* exp, *ulong* n)

Returns a^{exp} modulo n . We require $n < 2^{\text{FLINT\_D\_BITS}}$ and $0 \leq a < n$. There are no restrictions on **exp**, i.e. it can be negative.

This is implemented by precomputing an inverse and calling the **precomp** version of this function.

ulong **n_powmod2_preinv**(*ulong* a, *slong* exp, *ulong* n, *ulong* ninv)

Returns $(a^{\text{exp}}) \% n$ given a precomputed inverse of n computed by **n_preinvert_limb()**. We require $0 \leq a < n$, but there are no restrictions on n or on **exp**, i.e. it can be negative.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

If **exp** is negative but a is not invertible modulo n , an exception is raised.

ulong **n_powmod2**(*ulong* a, *slong* exp, *ulong* n)

Returns $(a^{\text{exp}}) \% n$. We require $0 \leq a < n$, but there are no restrictions on n or on **exp**, i.e. it can be negative.

This is implemented by precomputing an inverse limb and calling the **preinv** version of this function.

If **exp** is negative but a is not invertible modulo n , an exception is raised.

ulong **n_powmod2_ui_preinv**(*ulong* a, *ulong* exp, *ulong* n, *ulong* ninv)

Returns $(a^{\text{exp}}) \% n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. We require $0 \leq a < n$, but there are no restrictions on n . The exponent `exp` is unsigned and so can be larger than allowed by `n_powmod2_preinv()`.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

ulong **n_powmod2_fmpz_preinv**(*ulong* a, const *fmpz_t* exp, *ulong* n, *ulong* ninv)

Returns $(a^{\text{exp}}) \% n$ given a precomputed inverse of n computed by `n_preinvert_limb()`. We require $0 \leq a < n$, but there are no restrictions on n . The exponent `exp` must not be negative.

This is implemented as a standard binary powering algorithm using repeated squaring and reducing modulo n at each step.

ulong **n_sqrtmod**(*ulong* a, *ulong* p)

If p is prime, compute a square root of a modulo p if a is a quadratic residue modulo p , otherwise return 0.

If p is not prime the result is with high probability 0, indicating that p is not prime, or a is not a square modulo p . Otherwise the result is meaningless.

Assumes that a is reduced modulo p .

slong **n_sqrtmod_2pow**(*ulong* **sqrt, *ulong* a, *slong* exp)

Computes all the square roots of a modulo 2^{exp} . The roots are stored in an array which is created and whose address is stored in the location pointed to by `sqrt`. The array of roots is allocated by the function but must be cleaned up by the user by calling `flint_free`. The number of roots is returned by the function. If a is not a quadratic residue modulo 2^{exp} then 0 is returned by the function and the location `sqrt` points to is set to NULL.

slong **n_sqrtmod_primepow**(*ulong* **sqrt, *ulong* a, *ulong* p, *slong* exp)

Computes all the square roots of a modulo p^{exp} . The roots are stored in an array which is created and whose address is stored in the location pointed to by `sqrt`. The array of roots is allocated by the function but must be cleaned up by the user by calling `flint_free`. The number of roots is returned by the function. If a is not a quadratic residue modulo p^{exp} then 0 is returned by the function and the location `sqrt` points to is set to NULL.

slong **n_sqrtmodn**(*ulong* **sqrt, *ulong* a, *n_factor_t* *fac)

Computes all the square roots of a modulo m given the factorisation of m in `fac`. The roots are stored in an array which is created and whose address is stored in the location pointed to by `sqrt`. The array of roots is allocated by the function but must be cleaned up by the user by calling `flint_free()`. The number of roots is returned by the function. If a is not a quadratic residue modulo m then 0 is returned by the function and the location `sqrt` points to is set to NULL.

4.1.9 Modular arithmetic with fixed operand

This is about computing several modular multiplications where one operand and the modulus are fixed, that is, computing $ab_i \bmod n$ for several b_i 's and fixed a and n . Most functions below require $a < n < 2^{\text{FLINT\_BITS}-1}$ but have no constraint on b_i (it can be $\geq n$).

Some explanations about the method can be found after the functions descriptions. This has been introduced in NTL and is attributed to Victor Shoup. For references, see

- NTL code (in particular file FFT.cpp, consulted in NTL v11.5.1);
- David Harvey, Faster arithmetic for number-theoretic transforms, 2014 J.Symbolic Computation (<http://dx.doi.org/10.1016/j.jsc.2013.09.002>);
- Victor Shoup, "Arithmetic Software Libraries", 2021 (<https://doi.org/10.1017/9781108854207.012>) (chapter 9 of the book <https://doi.org/10.1017/9781108854207>).

`ulong n_mulmod_precomp_shoup(ulong a, ulong n)`

Returns `a_precomp`, a scaled approximation of a/n . This requires $a < n$. Precisely, `a_precomp` is the integer $\lfloor a \cdot 2^{\text{FLINT\_BITS}}/n \rfloor$, and is intended to be used as the precomputed data for `n_mulmod_shoup()`, which requires $n < 2^{\text{FLINT\_BITS}-1}$.

`void n_mulmod_precomp_shoup_quo_rem(ulong *a_pr_quo, ulong *a_pr_rem, ulong a, ulong n)`

Sets `a_pr_quo` to the above scaled approximation of $\lfloor a \cdot 2^{\text{FLINT\_BITS}}/n \rfloor$, which is the quotient in the integer division of $a \cdot 2^{\text{FLINT\_BITS}}$ by n , and also sets `a_pr_rem` to the remainder in this division. This requires $a < n$ and is intended to be used as the precomputed data for `n_mulmod_and_precomp_shoup()`, which requires $n < 2^{\text{FLINT\_BITS}-1}$.

`ulong n_mulmod_precomp_shoup_rem_from_quo(ulong a_pr_quo, ulong n)`

Returns the remainder in the integer division of $a \cdot 2^{\text{FLINT\_BITS}}$ by n . This requires $a < n$ and is intended to be used as the precomputed data for `n_mulmod_and_precomp_shoup()`, which requires $n < 2^{\text{FLINT\_BITS}-1}$. This is faster than `n_mulmod_precomp_shoup_quo_rem()` when `a_pr_quo` is already known.

`ulong n_mulmod_shoup(ulong a, ulong b, ulong a_precomp, ulong n)`

Returns $ab \bmod n$ given `a_precomp`, a precomputed scaled approximation of a/n equal to $\lfloor a \cdot 2^{\text{FLINT\_BITS}}/n \rfloor$. This requires $n < 2^{\text{FLINT\_BITS}-1}$ and $a < n$, there is no restriction on b . Works faster than other mulmod routines (e.g. `n_mulmod2_preinv()`) in situations where one repeats modular multiplications with fixed a and n and several values of b 's, which amortizes the time for precomputing `a_precomp`. Examples are scalar multiplication of vectors such as `nmod_vec_scalar_mul_nmod()` or of matrices such as `nmod_mat_scalar_mul()`.

`void n_mulmod_and_precomp_shoup(ulong *ab, ulong *ab_precomp, ulong a, ulong b, ulong a_pr_quo, ulong a_pr_rem, ulong b_precomp, ulong n)`

Sets `ab` to $ab \bmod n$ and sets `ab_precomp` to the precomputed scaled approximation for ab/n , that is, $\lfloor (ab \bmod n) \cdot 2^{\text{FLINT\_BITS}}/n \rfloor$. This requires $n < 2^{\text{FLINT\_BITS}-1}$, $a < n$, and $b < n$. The input `a_pr_quo` and `a_pr_rem` is as in the description of `n_mulmod_precomp_shoup_quo_rem()`, and `b_precomp` is as in the description of `n_mulmod_precomp_shoup()`. This can be used for example when seeking a list of powers $[a \bmod n, a^2 \bmod n, a^3 \bmod n, \dots]$ along with associated precomputed data to speed up repeated modular multiplications by these fixed powers.

Here are some explanations. Let $B = \text{FLINT\_BITS}$, and $W = 2^B$. We have as input a, b, n , and the goal is mainly to output $ab \bmod n$. Constraints are: $a < n$, and n has $< B$ bits, i.e. $0 \leq a < n < 2^{B-1}$. There is no restriction on b .

This is intended for repeated multiplications with fixed a and n , and varying b ; hence the initial step is seen as a precomputation (it depends on a and n only).

Precomputation step: The main function is `n_mulmod_precomp_shoup`, which computes `a_precomp` = $\lfloor aW/n \rfloor$. The requirement $a < n$ ensures `a_precomp` fits in a word (i.e. `a_precomp` $< W$). The variant ending in `_quo_rem` computes the same quotient `a_pr_quo` = `a_precomp` but also stores the remainder `a_pr_rem` in the division of aW by n ; the variant ending in `_rem_from_quo` deduces the remainder from the quotient if the latter is already known.

Modular multiplication: The main function is `n_mulmod_shoup`, which takes $a, b, \hat{a}, n$ as input and returns $ab \bmod n$, where $\hat{a} = \text{a\_precomp}$. The steps are:

1. find p_{hi}, p_{lo} such that $\hat{a}b = p_{hi}W + p_{lo}$ (high part of a double-word multiplication, p_{lo} will not be used)
2. compute $c = ab - p_{hi}n$ (two single-word multiplications)
3. if $c \geq n$, return $c - n$, else return c

Step 1. One has $p_{hi} = \lfloor ab/n \rfloor$ or $p_{hi} = \lfloor ab/n \rfloor - 1$.

Proof: Write $aW = \hat{a}n + r$, with $0 \leq r < n$. Thus $\frac{\hat{a}b}{W} = \frac{ab}{n} - \frac{rb}{nW}$. So $p_{hi} = \lfloor \frac{\hat{a}b}{W} \rfloor = \lfloor \frac{ab}{n} - \frac{rb}{nW} \rfloor$. Clearly $p_{hi} \leq \lfloor \frac{ab}{n} \rfloor$. And $rb < nW$ (since $r < n$ and $b < W$) so $-\frac{rb}{nW} > -1$, hence $p_{hi} \geq \lfloor \frac{ab}{n} \rfloor - 1$.

Step 2. It follows that either $c = ab \bmod n$ or $c = (ab \bmod n) + n$.

This is where the restriction on n comes into play: allowing n to have B bits prevents us from detecting which case we are in (a comparison such as $c \geq n$ would not tell). Since n has $< B$ bits, $c \geq n$ if and only if $c = (ab \bmod n) + n$.

Step 3. we use this to detect which case we are in and correct the possible excess.

Combining precomputation and multiplication: The main function is `n_mulmod_and_precomp_shoup`. The goal is to compute not only $ab \bmod n$, but also the corresponding quotient `ab_precomp` = $\lfloor (ab \bmod n) \cdot W/n \rfloor$. For this we require as input both the precomputed quotient and remainder for a (that is, `a_pr_quo` and `a_pr_rem`), as well as the precomputed quotient for b (`b_precomp`). This adds the constraint that $b < n$.

To simplify notation, write the integer division $aW = \hat{a}n + \check{a}$ so that $\hat{a} = \text{a\_pr\_quo}$ and $\check{a} = \text{a\_pr\_rem}$, and similarly for $bW = \hat{b}n + \check{b}$ with $\hat{b} = \text{b\_precomp}$ (we will not use $\check{b}$).

The first three steps follow the above-described approach to compute $ab \bmod n$ using $\hat{a}$ (but this time, the lower part of the double-word multiplication is kept):

1. find $d, \hat{c}$ such that $\hat{a}\hat{b} = dW + \hat{c}$ (double-word multiplication)
2. compute $c = ab - dn$ (two single-word multiplications)
3. if $c \geq n$ then $c \leftarrow c - n$

At this stage, $c = ab \bmod n$, and $\hat{c} = \hat{a}\hat{b} \bmod W$. Now suppose we have the quotient $q = \lfloor \frac{\hat{a}\hat{b}}{n} \rfloor$ in the division of $\hat{a}\hat{b}$ by n . The claim is that $\hat{c} + q$ is the sought precomputation quotient for $ab \bmod n$ up to a multiple of W , that is, $\hat{c} + q = \lfloor \frac{cW}{n} \rfloor \bmod W$. Indeed, one has $\lfloor \frac{cW}{n} \rfloor = \lfloor \frac{abW}{n} \rfloor \bmod W$, and it remains to observe that $\lfloor \frac{abW}{n} \rfloor = \lfloor \frac{(\hat{a}n + \check{a})b}{n} \rfloor = \hat{a}\hat{b} + \lfloor \frac{\check{a}b}{n} \rfloor = \hat{c} + q \bmod W$.

Therefore the remaining steps consist in computing the quotient q , which can be done via the modular multiplication of $\check{a}$ by b using the precomputed quotient for b :

4. find q, e such that $\hat{b}\check{a} = qW + e$ (high-part of double word multiplication, e will not be used)
5. compute $h = b\check{a} - qn$ (two single-word multiplications)
6. if $h \geq n$ then $q \leftarrow q + 1$

4.1.10 Divisibility testing

`int n_divides(ulong *q, ulong n, ulong p)`

Returns 1 if p divides n and sets q to the quotient, otherwise returns 0 and sets q to 0.

4.1.11 Prime number generation and counting

`void n_primes_init(n_primes_t iter)`

Initialises the prime number iterator `iter` for use.

`void n_primes_clear(n_primes_t iter)`

Clears memory allocated by the prime number iterator `iter`.

`ulong n_primes_next(n_primes_t iter)`

Returns the next prime number and advances the state of `iter`. The first call returns 2.

Small primes are looked up from `flint_small_primes`. When this table is exhausted, primes are generated in blocks by calling `n_primes_sieve_range()`.

`void n_primes_jump_after(n_primes_t iter, ulong n)`

Changes the state of `iter` to start generating primes after n (excluding n itself).

`void n_primes_extend_small(n_primes_t iter, ulong bound)`

Extends the table of small primes in `iter` to contain at least two primes larger than or equal to `bound`.

void **n_primes_sieve_range**(n_primes_t iter, *ulong* a, *ulong* b)

Sets the block endpoints of *iter* to the smallest and largest odd numbers between *a* and *b* inclusive, and sieves to mark all odd primes in this range. The iterator state is changed to point to the first number in the sieved range.

void **n_compute_primes**(*ulong* num_primes)

Precomputes at least *num_primes* primes and their **double** precomputed inverses and stores them in an internal cache. Assuming that FLINT has been built with support for thread-local storage, each thread has its own cache.

const *ulong* ***n_primes_arr_readonly**(*ulong* num_primes)

Returns a pointer to a read-only array of the first *num_primes* prime numbers. The computed primes are cached for repeated calls. The pointer is valid until the user calls **n_cleanup_primes()** in the same thread.

const double ***n_prime_inverses_arr_readonly**(*ulong* n)

Returns a pointer to a read-only array of inverses of the first *num_primes* prime numbers. The computed primes are cached for repeated calls. The pointer is valid until the user calls **n_cleanup_primes()** in the same thread.

void **n_cleanup_primes**()

Frees the internal cache of prime numbers used by the current thread. This will invalidate any pointers returned by **n_primes_arr_readonly()** or **n_prime_inverses_arr_readonly()**.

ulong **n_nextprime**(*ulong* n, int proved)

Returns the next prime after *n*. Assumes the result will fit in an *ulong*. If *proved* is 0, i.e. false, the prime is not proven prime, otherwise it is.

ulong **n_prime_pi**(*ulong* n)

Returns the value of the prime counting function $\pi(n)$, i.e. the number of primes less than or equal to *n*. The invariant **n_prime_pi**(**n_nth_prime**(*n*)) == *n*.

Currently, this function simply extends the table of cached primes up to an upper limit and then performs a binary search.

void **n_prime_pi_bounds**(*ulong* *lo, *ulong* *hi, *ulong* n)

Calculates lower and upper bounds for the value of the prime counting function $lo \leq \pi(n) \leq hi$. If *lo* and *hi* point to the same location, the high value will be stored.

This does a table lookup for small values, then switches over to some proven bounds.

The upper approximation is $1.25506n/\ln n$, and the lower is $n/\ln n$. These bounds are due to Rosser and Schoenfeld [RosSch1962] and valid for $n \geq 17$.

We use the number of bits in *n* (or one less) to form an approximation to $\ln n$, taking care to use a value too small or too large to maintain the inequality.

ulong **n_nth_prime**(*ulong* n)

Returns the *n*th prime number p_n , using the mathematical indexing convention $p_1 = 2, p_2 = 3, \dots$

This function simply ensures that the table of cached primes is large enough and then looks up the entry.

void **n_nth_prime_bounds**(*ulong* *lo, *ulong* *hi, *ulong* n)

Calculates lower and upper bounds for the *n*th prime number p_n , $lo \leq p_n \leq hi$. If *lo* and *hi* point to the same location, the high value will be stored. Note that this function will overflow for sufficiently large *n*.

We use the following estimates, valid for $n > 5$:

$$\begin{aligned} p_n &> n(\ln n + \ln \ln n - 1) \\ p_n &< n(\ln n + \ln \ln n) \\ p_n &< n(\ln n + \ln \ln n - 0.9427) \quad (n \geq 15985) \end{aligned}$$

The first inequality was proved by Dusart [Dus1999], and the last is due to Massias and Robin [MasRob1996]. For a further overview, see <http://primes.utm.edu/howmany.shtml>.

We bound $\ln n$ using the number of bits in n as in `n_prime_pi_bounds()`, and estimate $\ln \ln n$ to the nearest integer; this function is nearly constant.

4.1.12 Primality testing

int `n_is_prime`(*ulong* n)

Returns 1 if n is prime and 0 otherwise. This function is intended for general input.

For integers up to 15 bits, we look up the result in a bit array. For integers between 16 and 64 bits, we do some trial division and then perform a single base-2 strong probable prime test to detect most composites. For numbers that pass this test, primality is certified by checking that n is not one of the 31894014 base-2 strong pseudoprimes $n < 2^{64}$ which have been tabulated exhaustively by Feitsma [FeiGal2013].

The 2314 pseudoprimes up to 32 bits are simply looked up in a hash table.

For the pseudoprimes up to 64 bits we modify the approach of Forisek and Jancina [ForJan2015]. We first eliminate a class of pseudoprimes that are difficult to find reliable witnesses to. These are of the form pq where p and q are primes and $q = k \cdot p(-1) + 1$ where k is in the interval $[2, 9]$ and 12. A semiprime check algorithm using a single floating-point sqrt, and the multiplicative inverses of the sqrt of k eliminates this class in approximately 1/20 of the runtime of a fermat test. The remaining pseudoprimes are partitioned into 32768 sets using a fast multiplicative hash, and then tested against a precomputed witness smaller than 16-bit that is reliable to each set.

The total witness table requires 64KiB or 1/8 of the Forisek-Jancina table of 262144 16-bit bases. Due to the semiprime check this test is slightly less efficient in the case of primes, and equivalent in the average case but it's low memory usage means it performs better in memory intensive computations.

To check this implementation against Feitsma's table, one can run the `examples/check_n_is_prime` program.

int `n_is_prime_odd_no_trial`(*ulong* n)

As `n_is_prime()`, but does not perform trial division. Requires that n is odd. This function is much slower than `n_is_prime()` for random input but slightly faster for certifying primality or compositeness if n has already passed preliminary trial division or sieving done by the user.

int `n_is_prime_pocklington`(*ulong* n , *ulong* iterations)

Tests if n is a prime using the Pocklington–Lehmer primality test. If 1 is returned n has been proved prime. If 0 is returned n is composite. However -1 may be returned if nothing was proved either way due to the number of iterations being too small.

The most time consuming part of the algorithm is factoring $n - 1$. For this reason `n_factor_partial()` is used, which uses a combination of trial factoring and Hart's one line factor algorithm [Har2012] to try to quickly factor $n - 1$. Additionally if the cofactor is less than the square root of $n - 1$ the algorithm can still proceed.

One can also specify a number of iterations if less time should be taken. Simply set this to `WORD(0)` if this is irrelevant. In most cases a greater number of iterations will not significantly affect timings as most of the time is spent factoring.

See <https://mathworld.wolfram.com/PocklingtonsTheorem.html> for a description of the algorithm.

int `n_is_prime_pseudosquare`(*ulong* n)

Tests if n is a prime according to Theorem 2.7 [LukPatWil1996].

We first factor N using trial division up to some limit B . In fact, the number of primes used in the trial factoring is at most `FLINT_PSEUDOSQUARES_CUTOFF`.

Next we compute N/B and find the next pseudosquare L_p above this value, using a static table as per <https://oeis.org/A002189/b002189.txt>.

As noted in the text, if p is prime then Step 3 will pass. This test rejects many composites, and so by this time we suspect that p is prime. If N is 3 or 7 modulo 8, we are done, and N is prime.

We now run a probable prime test, for which no known counterexamples are known, to reject any composites. We then proceed to prove N prime by executing Step 4. In the case that N is 1 modulo 8, if Step 4 fails, we extend the number of primes p_i at Step 3 and hope to find one which passes Step 4. We take the test one past the largest p for which we have pseudosquares L_p tabulated, as this already corresponds to the next L_p which is bigger than 2^{64} and hence larger than any prime we might be testing.

As explained in the text, Condition 4 cannot fail if N is prime.

The possibility exists that the probable prime test declares a composite prime. However in that case an error is printed, as that would be of independent interest.

int **n_is_strong_probabprime_precomp**(ulong n, double npre, ulong a, ulong d)

Tests if n is a strong probable prime to the base a . We require that d is set to the largest odd factor of $n - 1$ and $npre$ is a precomputed inverse of n computed with `n_precompute_inverse()`. We also require that $n < 2^{53}$, a to be reduced modulo n and not 0 and n to be odd.

If we write $n - 1 = 2^s d$ where d is odd then n is a strong probable prime to the base a , i.e. an a -SPRP, if either $a^d = 1 \pmod{n}$ or $(a^d)^{2^r} = -1 \pmod{n}$ for some r less than s .

A description of strong probable primes is given here: <https://mathworld.wolfram.com/StrongPseudoprime.html>

int **n_is_strong_probabprime2_preinv**(ulong n, ulong ninv, ulong a, ulong d)

Tests if n is a strong probable prime to the base a . We require that d is set to the largest odd factor of $n - 1$ and $npre$ is a precomputed inverse of n computed with `n_preinvert_limb()`. We require a to be reduced modulo n and non-zero, and n to be odd.

If we write $n - 1 = 2^s d$ where d is odd then n is a strong probable prime to the base a (an a -SPRP) if either $a^d = 1 \pmod{n}$ or $(a^d)^{2^r} = -1 \pmod{n}$ for some r less than s .

A description of strong probable primes is given here: <https://mathworld.wolfram.com/StrongPseudoprime.html>

int **n_is_probabprime_fermat**(ulong n, ulong i)

Returns 1 if n is a base i Fermat probable prime. Requires $1 < i < n$ and that i does not divide n .

By Fermat's Little Theorem if i^{n-1} is not congruent to 1 then n is not prime.

int **n_is_probabprime_fibonacci**(ulong n)

Let F_j be the j th element of the Fibonacci sequence 0, 1, 1, 2, 3, 5, ..., starting at $j = 0$. Then if n is prime we have $F_{n-(n/5)} = 0 \pmod{n}$, where $(n/5)$ is the Jacobi symbol.

For further details, see pp. 142 [CraPom2005].

We require that n is not divisible by 2 or 5.

int **n_is_probabprime_BPSW**(ulong n)

Implements a Baillie–Pomerance–Selfridge–Wagstaff probable primality test. This is a variant of the usual BPSW test (which only uses strong base-2 probable prime and Lucas–Selfridge tests, see Baillie and Wagstaff [BaiWag1980]).

This implementation makes use of a weakening of the usual Baillie–PSW test given in [Chen2003], namely replacing the Lucas test with a Fibonacci test when $n \equiv 2, 3 \pmod{5}$ (see also the comment on page 143 of [CraPom2005]), regarding Fibonacci pseudoprimes.

There are no known counterexamples to this being a primality test.

Up to 2^{64} the test we use has been checked against tables of pseudoprimes. Thus it is a primality test up to this limit.

int **n_is_probabprime_lucas**(*ulong* n)

For details on Lucas pseudoprimes, see [pp. 143] [CraPom2005].

We implement a variant of the Lucas pseudoprime test similar to that described by Baillie and Wagstaff [BaiWag1980].

int **n_is_probabprime**(*ulong* n)

This function is obsolete and currently just wraps *n_is_prime()*.

4.1.13 Chinese remaindering

ulong **n_CRT**(*ulong* r1, *ulong* m1, *ulong* r2, *ulong* m2)

Use the Chinese Remainder Theorem to return the unique value $0 \leq x < M$ congruent to r_1 modulo m_1 and r_2 modulo m_2 , where $M = m_1 \times m_2$ is assumed to fit a *ulong*.

It is assumed that m_1 and m_2 are positive integers greater than 1 and coprime. It is assumed that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$.

4.1.14 Square root and perfect power testing

ulong **n_sqrt**(*ulong* a)

Computes the integer truncation of the square root of a .

The implementation uses a call to the IEEE floating point sqrt function. The integer itself is represented by the nearest double and its square root is computed to the nearest place. If a is one below a square, the rounding may be up, whereas if it is one above a square, the rounding will be down. Thus the square root may be one too large in some instances which we then adjust by checking if we have the right value. We also have to be careful when the square of this too large value causes an overflow. The same assumptions hold for a single precision float provided the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$.

ulong **n_sqrtrem**(*ulong* *r, *ulong* a)

Computes the integer truncation of the square root of a .

The integer itself is represented by the nearest double and its square root is computed to the nearest place. If a is one below a square, the rounding may be up, whereas if it is one above a square, the rounding will be down. Thus the square root may be one too large in some instances which we then adjust by checking if we have the right value. We also have to be careful when the square of this too large value causes an overflow. The same assumptions hold for a single precision float provided the square root itself can be represented in a single float, i.e. for $a < 281474976710656 = 2^{46}$.

The remainder is computed by subtracting the square of the computed square root from a .

int **n_is_square**(*ulong* x)

Returns 1 if x is a square, otherwise 0.

This code first checks if x is a square modulo 64, $63 = 3 \times 3 \times 7$ and $65 = 5 \times 13$, using lookup tables, and if so it then takes a square root and checks that the square of this equals the original value.

int **n_is_perfect_power235**(*ulong* n)

Returns 1 if n is a perfect square, cube or fifth power.

This function uses a series of modular tests to reject most non 235-powers. Each modular test returns a value from 0 to 7 whose bits respectively indicate whether the value is a square, cube or fifth power modulo the given modulus. When these are logically AND-ed together, this gives a powerful test which will reject most non-235 powers.

If a bit remains set indicating it may be a square, a standard square root test is performed. Similarly a cube root or fifth root can be taken, if indicated, to determine whether the power of that root is exactly equal to n .

`int n_is_perfect_power(ulong *root, ulong n)`

If $n = r^k$, return k and set `root` to r . Note that 0 and 1 are considered squares. No guarantees are made about r or k being the minimum possible value.

`ulong n_root(ulong n, ulong root)`

`ulong n_rootrem(ulong *remainder, ulong n, ulong root)`

Returns the integer part of $n^{1/\text{root}}$. Remainder is set as $n - \text{base}^{\text{root}}$. In case $n < 1$ or $\text{root} < 1$, 0 is returned.

`ulong n_cbrt(ulong n)`

This function returns the integer truncation of the cube root of n . First approximation is calculated by an algorithm mentioned in this article: https://en.wikipedia.org/wiki/Fast_inverse_square_root. Instead of the inverse square root, the cube root is calculated. This functions uses different algorithms to calculate the cube root, depending upon the size of n . For numbers greater than 2^{46} , it uses `n_cbrt_chebyshev_approx()`. Otherwise, it makes use of the iteration, $x \leftarrow x - (x \cdot x \cdot x - a) \cdot x / (2 \cdot x \cdot x \cdot x + a)$ for getting a good estimate, as mentioned in the paper by W. Kahan [Kahan1991].

`ulong n_cbrt_binary_search(ulong n)`

This function returns the integer truncation of the cube root of n . Uses binary search to get the correct value.

`ulong n_cbrt_chebyshev_approx(ulong n)`

This function returns the integer truncation of the cube root of n . The number is first expressed in the form $x \cdot 2^{\text{exp}}$. This ensures x is in the range $[0.5, 1]$. Cube root of x is calculated using Chebyshev's approximation polynomial for the function $y = x^{1/3}$. The values of the coefficient are calculated from the Python module `mpmath`, <https://mpmath.org>, using the function `chebyfit`. x is multiplied by 2^{exp} and the cube root of 1, 2 or 4 (according to `exp%3`). Requires that n is nonzero.

`ulong n_cbrtrem(ulong *remainder, ulong n)`

This function returns the integer truncation of the cube root of n . Remainder is set as n minus the cube of the value returned.

4.1.15 Factorisation

`type n_factor_t`

Represents a set of distinct prime factors and their corresponding exponents.

`int num`

The number of distinct prime factors.

`ulong p[]`

An array containing the distinct prime factors.

`int exp[]`

An array containing the corresponding exponents.

`void n_factor_init(n_factor_t *factors)`

Initializes factors.

`ulong n_factor_evaluate(const n_factor_t *factors)`

Returns the evaluation of `factors`, i.e. $p_1^{e_1} \cdots p_n^{e_n}$ assuming that it fits in a limb. In case the evaluation does not fit in a limb, it returns 0.

`int n_remove(ulong *n, ulong p)`

Removes the highest possible power of p from n , replacing n with the quotient. The return value is the highest power of p that divided n . Assumes n is not 0.

For $p = 2$ trailing zeroes are counted. For other primes p is repeatedly squared and stored in a table of powers with the current highest power of p removed at each step until no higher power can be removed. The algorithm then proceeds down the power tree again removing powers of p until none remain.

int **n_remove2_precomp**(ulong *n, ulong p, double ppre)

Removes the highest possible power of p from n , replacing n with the quotient. The return value is the highest power of p that divided n . Assumes n is not 0. We require `ppre` to be set to a precomputed inverse of p computed with `n_precompute_inverse()`.

For $p = 2$ trailing zeroes are counted. For other primes p we make repeated use of `n_divrem2_precomp()` until division by p is no longer possible.

void **n_factor_insert**(n_factor_t *factors, ulong p, ulong exp)

Inserts the given prime power factor p^{exp} into `factors`.

The algorithm performs a simple search to see if p already exists as a prime factor in the structure. If so the exponent there is increased by the supplied exponent. Otherwise a new factor p^{exp} is added to the end of the structure.

There is no test code for this function other than its use by the various factoring functions, which have test code.

ulong **n_factor_trial_range**(n_factor_t *factors, ulong n, ulong start, ulong num_primes)

Trial factor n with the first `num_primes` primes, but starting at the prime with index `start` (counting from zero).

One requires an initialised `n_factor_t` structure, but factors will be added by default to an already used `n_factor_t`. Use the function `n_factor_init()` defined in `ulong_extras` if initialisation has not already been completed on factors.

The return value is the unfactored cofactor after trial factoring is done.

The function calls `n_compute_primes()` automatically. See the documentation for that function regarding limits.

The algorithm stops when the current prime has a square exceeding n , as no prime factor of n can exceed this unless n is prime.

The precomputed inverses of all the primes computed by `n_compute_primes()` are utilised with the `n_remove2_precomp()` function.

ulong **n_factor_trial**(n_factor_t *factors, ulong n, ulong num_primes)

This function calls `n_factor_trial_range()`, with the value of 0 for `start`. By default this adds factors to an already existing `n_factor_t` or to a newly initialised one.

ulong **n_factor_power235**(ulong *exp, ulong n)

Returns 0 if n is not a perfect square, cube or fifth power. Otherwise it returns the root and sets `exp` to either 2, 3 or 5 appropriately.

This function uses a series of modular tests to reject most non 235-powers. Each modular test returns a value from 0 to 7 whose bits respectively indicate whether the value is a square, cube or fifth power modulo the given modulus. When these are logically AND-ed together, this gives a powerful test which will reject most non-235 powers.

If a bit remains set indicating it may be a square, a standard square root test is performed. Similarly a cube root or fifth root can be taken, if indicated, to determine whether the power of that root is exactly equal to n .

ulong **n_factor_one_line**(ulong n, ulong iters)

This implements Bill Hart's one line factoring algorithm [Har2012]. It is a variant of Fermat's algorithm which cycles through a large number of multipliers instead of incrementing the square root. It is faster than SQUFOF for n less than about 2^{40} .

ulong **n_factor_lehman**(*ulong* n)

Lehman's factoring algorithm. Currently works up to 10^{16} , but is not particularly efficient and so is not used in the general factor function. Always returns a factor of n .

ulong **n_factor_SQUFOF**(*ulong* n, *ulong* iters)

Attempts to split n using the given number of iterations of SQUFOF. Simply set `iters` to `WORD(0)` for maximum persistence.

The version of SQUFOF implemented here is as described by Gower and Wagstaff [GowWag2008].

We start by trying SQUFOF directly on n . If that fails we multiply it by each of the primes in `flint_primes_small` in turn. As this multiplication may result in a two limb value we allow this in our implementation of SQUFOF. As SQUFOF works with values about half the size of n it only needs single limb arithmetic internally.

If SQUFOF fails to factor n we return 0, however with `iters` large enough this should never happen.

void **n_factor**(*n_factor_t* *factors, *ulong* n, int proved)

Factors n with no restrictions on n . If the prime factors are required to be checked with a primality test, one may set `proved` to 1, otherwise set it to 0, and they will only be probable primes. NB: at the present there is no difference because the probable prime tests have been exhaustively tested up to 2^{64} .

However, in future, this flag may produce and separately check a primality certificate. This may be quite slow (and probably no less reliable in practice).

This function first tries trial factoring with a number of primes specified by the constant `FLINT_FACTOR_TRIAL_PRIMES`. If the cofactor is 1 or prime the function returns with all the factors.

Otherwise, the cofactor is placed in the array `factor_arr`. Whilst there are factors remaining in there which have not been split, the algorithm continues. At each step each factor is first checked to determine if it is a perfect power. If so it is replaced by the power that has been found. Next if the factor is small enough and composite, in particular, less than `FLINT_FACTOR_ONE_LINE_MAX` then `n_factor_one_line()` is called with `FLINT_FACTOR_ONE_LINE_ITERS` to try and split the factor. If that fails or the factor is too large for `n_factor_one_line()` then `n_factor_SQUFOF()` is called, with `FLINT_FACTOR_SQUFOF_ITERS`. If that fails an error results and the program aborts. However this should not happen in practice.

ulong **n_factor_trial_partial**(*n_factor_t* *factors, *ulong* n, *ulong* *prod, *ulong* num_primes, *ulong* limit)

Attempts trial factoring of n with the first `num_primes` primes, but stops when the product of prime factors so far exceeds `limit`.

One requires an initialised `n_factor_t` structure, but factors will be added by default to an already used `n_factor_t`. Use the function `n_factor_init()` defined in `ulong_extras` if initialisation has not already been completed on `factors`.

Once completed, `num` will contain the number of distinct prime factors found. The field `p` is an array of `ulong`s containing the distinct prime factors, `exp` an array containing the corresponding exponents.

The return value is the unfactored cofactor after trial factoring is done. The value `prod` will be set to the product of the factors found.

The function calls `n_compute_primes()` automatically. See the documentation for that function regarding limits.

The algorithm stops when the current prime has a square exceeding n , as no prime factor of n can exceed this unless n is prime.

The precomputed inverses of all the primes computed by `n_compute_primes()` are utilised with the `n_remove2_precomp()` function.

ulong **n_factor_partial**(*n_factor_t* *factors, *ulong* n, *ulong* limit, int proved)

Factors n , but stops when the product of prime factors so far exceeds *limit*.

One requires an initialised *n_factor_t* structure, but factors will be added by default to an already used *n_factor_t*. Use the function *n_factor_init*() defined in *ulong_extras* if initialisation has not already been completed on *factors*.

On exit, *num* will contain the number of distinct prime factors found. The field *p* is an array of *ulong*s containing the distinct prime factors, *exp* an array containing the corresponding exponents.

The return value is the unfactored cofactor after factoring is done.

The factors are proved prime if *proved* is 1, otherwise they are merely probably prime.

ulong **n_factor_pp1**(*ulong* n, *ulong* B1, *ulong* c)

Factors n using Williams' $p + 1$ factoring algorithm, with prime limit set to $B1$. We require c to be set to a random value. Each trial of the algorithm with a different value of c gives another chance to factor n , with roughly exponentially decreasing chance of finding a missing factor. If $p + 1$ (or $p - 1$) is not smooth for any factor p of n , the algorithm will never succeed. The value c should be less than n and greater than 2.

If the algorithm succeeds, it returns the factor, otherwise it returns 0 or 1 (the trivial factors modulo n).

ulong **n_factor_pp1_wrapper**(*ulong* n)

A simple wrapper around *n_factor_pp1* which works in the range 31-64 bits. Below this point, trial factoring will always succeed. This function mainly exists for *n_factor* and is tuned to minimise the time for *n_factor* on numbers that reach the *n_factor_pp1* stage, i.e. after trial factoring and one line factoring.

int **n_factor_pollard_brent_single**(*ulong* *factor, *ulong* n, *ulong* ai, *ulong* xi, *ulong* max_iters)

Pollard Rho algorithm (with Brent modification) for integer factorization. Assumes that the n is not prime. *factor* is set as the factor if found. It is not assured that the factor found will be prime. Does not compute the complete factorization, just one factor. Returns 1 if factorization is successful (non trivial factor is found), else returns 0.

ai is the constant of the polynomial used, *xi* is the initial value. *max_iters* is the number of iterations tried in process of finding the cycle.

The algorithm used is a modification of the original Pollard Rho algorithm, suggested by Richard Brent in the paper, available at <https://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf>.

int **n_factor_pollard_brent**(*ulong* *factor, *flint_rand_t* state, *ulong* n_in, *ulong* max_tries, *ulong* max_iters)

Pollard Rho algorithm, modified as suggested by Richard Brent. Makes a call to *n_factor_pollard_brent_single*(). The input parameters *ai* and *xi* for *n_factor_pollard_brent_single*() are selected at random.

If the algorithm fails to find a non trivial factor in one call, it tries again (this time with a different set of random values). This process is repeated a maximum of *max_tries* times.

Assumes n is not prime. *factor* is set as the factor found, if factorization is successful. In such a case, 1 is returned. Otherwise, 0 is returned. Factor discovered is not necessarily prime.

4.1.16 Arithmetic functions

int **n_moebius_mu**(*ulong* n)

Computes the Moebius function $\mu(n)$, which is defined as $\mu(n) = 0$ if n has a prime factor of multiplicity greater than 1, $\mu(n) = -1$ if n has an odd number of distinct prime factors, and $\mu(n) = 1$ if n has an even number of distinct prime factors. By convention, $\mu(0) = 0$.

For even numbers, we use the identities $\mu(4n) = 0$ and $\mu(2n) = -\mu(n)$. Odd numbers up to a cutoff are then looked up from a precomputed table storing $\mu(n) + 1$ in groups of two bits.

For larger n , we first check if n is divisible by a small odd square and otherwise call **n_factor**() and count the factors.

void **n_moebius_mu_vec**(int *mu, *ulong* len)

Computes $\mu(n)$ for $n = 0, 1, \dots, \text{len} - 1$. This is done by sieving over each prime in the range, flipping the sign of $\mu(n)$ for every multiple of a prime p and setting $\mu(n) = 0$ for every multiple of p^2 .

int **n_is_squarefree**(*ulong* n)

Returns 0 if n is divisible by some perfect square, and 1 otherwise. This simply amounts to testing whether $\mu(n) \neq 0$. As special cases, 1 is considered squarefree and 0 is not considered squarefree.

ulong **n_euler_phi**(*ulong* n)

Computes the Euler totient function $\phi(n)$, counting the number of positive integers less than or equal to n that are coprime to n .

4.1.17 Factorials

ulong **n_factorial_fast_mod2_preinv**(*ulong* n, *ulong* p, *ulong* pinv)

Returns $n! \bmod p$ given a precomputed inverse of p as computed by **n_preinvert_limb**(). p is not required to be a prime, but no special optimisations are made for composite p . Uses fast multipoint evaluation, running in about $O(n^{1/2})$ time.

ulong **n_factorial_mod2_preinv**(*ulong* n, *ulong* p, *ulong* pinv)

Returns $n! \bmod p$ given a precomputed inverse of p as computed by **n_preinvert_limb**(). p is not required to be a prime, but no special optimisations are made for composite p .

Uses a lookup table for small n , otherwise computes the product if n is not too large, and calls the fast algorithm for extremely large n .

4.1.18 Primitive roots and discrete logarithms

ulong **n_primitive_root_prime_prefactor**(*ulong* p, *n_factor_t* *factors)

Returns a primitive root for the multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$ where p is prime given the factorisation (**factors**) of $p - 1$.

ulong **n_primitive_root_prime**(*ulong* p)

Returns a primitive root for the multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$ where p is prime.

ulong **n_discrete_log_bsgs**(*ulong* b, *ulong* a, *ulong* n)

Returns the discrete logarithm of b with respect to a in the multiplicative subgroup of $\mathbb{Z}/n\mathbb{Z}$ when $\mathbb{Z}/n\mathbb{Z}$ is cyclic. That is, it returns a number x such that $a^x = b \bmod n$. The multiplicative subgroup is only cyclic when n is 2, 4, p^k , or $2p^k$ where p is an odd prime and k is a positive integer.

4.1.19 Elliptic curve method for factorization of `ulong`

void **n_factor_ecm_double**(*ulong* *x, *ulong* *z, *ulong* x0, *ulong* z0, *ulong* n, *n_ecm_t* n_ecm_inf)

Sets the point $(x : z)$ to two times $(x_0 : z_0)$ modulo n according to the formula

$$x = (x_0 + z_0)^2 \cdot (x_0 - z_0)^2 \pmod n,$$

$$z = 4x_0z_0((x_0 - z_0)^2 + 4a_{24}x_0z_0) \pmod n.$$

This group doubling is valid only for points expressed in Montgomery projective coordinates.

void **n_factor_ecm_add**(*ulong* *x, *ulong* *z, *ulong* x1, *ulong* z1, *ulong* x2, *ulong* z2, *ulong* x0, *ulong* z0, *ulong* n, *n_ecm_t* n_ecm_inf)

Sets the point $(x : z)$ to the sum of $(x_1 : z_1)$ and $(x_2 : z_2)$ modulo n , given the difference $(x_0 : z_0)$ according to the formula

This group doubling is valid only for points expressed in Montgomery projective coordinates.

void **n_factor_ecm_mul_montgomery_ladder**(*ulong* *x, *ulong* *z, *ulong* x0, *ulong* z0, *ulong* k, *ulong* n, *n_ecm_t* n_ecm_inf)

Montgomery ladder algorithm for scalar multiplication of elliptic points.

Sets the point $(x : z)$ to $k(x_0 : z_0)$ modulo n .

Valid only for points expressed in Montgomery projective coordinates.

int **n_factor_ecm_select_curve**(*ulong* *f, *ulong* sigma, *ulong* n, *n_ecm_t* n_ecm_inf)

Selects a random elliptic curve given a random integer `sigma`, according to Suyama's parameterization. If the factor is found while selecting the curve, 1 is returned. In case the curve found is not suitable, 0 is returned.

Also selects the initial point x_0 , and the value of $(a + 2)/4$, where a is a curve parameter. Sets z_0 as 1 (shifted left by `n_ecm_inf->normbits`). All these are stored in the `n_ecm_t` struct.

The curve selected is of Montgomery form, the points selected satisfy the curve and are projective coordinates.

int **n_factor_ecm_stage_I**(*ulong* *f, const *ulong* *prime_array, *ulong* num, *ulong* B1, *ulong* n, *n_ecm_t* n_ecm_inf)

Stage I implementation of the ECM algorithm.

`f` is set as the factor if found. `num` is number of prime numbers $\leq$ the bound `B1`. `prime_array` is an array of first `B1` primes. `n` is the number being factored.

If the factor is found, 1 is returned, otherwise 0.

int **n_factor_ecm_stage_II**(*ulong* *f, *ulong* B1, *ulong* B2, *ulong* P, *ulong* n, *n_ecm_t* n_ecm_inf)

Stage II implementation of the ECM algorithm.

`f` is set as the factor if found. `B1`, `B2` are the two bounds. `P` is the primorial (approximately equal to $\sqrt{B2}$). `n` is the number being factored.

If the factor is found, 1 is returned, otherwise 0.

int **n_factor_ecm**(*ulong* *f, *ulong* curves, *ulong* B1, *ulong* B2, *flint_rand_t* state, *ulong* n)

Outer wrapper function for the ECM algorithm. It factors n which must fit into a `ulong`.

The function calls stage I and II, and the precomputations (builds `prime_array` for stage I, `GCD_table` and `prime_table` for stage II).

`f` is set as the factor if found. `curves` is the number of random curves being tried. `B1`, `B2` are the two bounds or stage I and stage II. `n` is the number being factored.

If a factor is found in stage I, 1 is returned. If a factor is found in stage II, 2 is returned. If a factor is found while selecting the curve, -1 is returned. Otherwise 0 is returned.

4.2 fmpz.h – integers

By default, an *fmpz_t* is implemented as an array of *fmpz*'s of length one to allow passing by reference as one can do with GMP's *mpz_t* type. The *fmpz_t* type is simply a single limb, though the user does not need to be aware of this except in one specific case outlined below.

In all respects, *fmpz_t*'s act precisely like GMP's *mpz_t*'s, with automatic memory management, however, in the first place only one limb is used to implement them. Once an *fmpz_t* overflows a limb then a multiprecision integer is automatically allocated and instead of storing the actual integer data the *slong* which implements the type becomes an index into a FLINT wide array of *mpz_t*'s.

These internal implementation details are not important for the user to understand, except for three important things.

Firstly, *fmpz_t*'s will be more efficient than *mpz_t*'s for single limb operations, or more precisely for signed quantities whose absolute value does not exceed `FLINT_BITS - 2` bits.

Secondly, for small integers that fit into `FLINT_BITS - 2` bits much less memory will be used than for an *mpz_t*. When very many *fmpz_t*'s are used, there can be important cache benefits on account of this.

Thirdly, it is important to understand how to deal with arrays of *fmpz_t*'s. As for *mpz_t*'s, there is an underlying type, an *fmpz*, which can be used to create the array, e.g.

```
fmpz myarr[100];
```

Now recall that an *fmpz_t* is an array of length one of *fmpz*'s. Thus, a pointer to an *fmpz* can be used in place of an *fmpz_t*. For example, to find the sign of the third integer in our array we would write

```
int sign = fmpz_sgn(myarr + 2);
```

The *fmpz* module provides routines for memory management, basic manipulation and basic arithmetic.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

4.2.1 Simple example

The following example computes the square of the integer 7 and prints the result.

```
#include "fmpz.h"

int main()
{
    fmpz_t x, y;
    fmpz_init(x);
    fmpz_init(y);
    fmpz_set_ui(x, 7);
    fmpz_mul(y, x, x);
    fmpz_print(x);
    flint_printf("^2 = ");
    fmpz_print(y);
    flint_printf("\n");
    fmpz_clear(x);
    fmpz_clear(y);
}
```

```
7^2 = 49
```


4.2.2 Types, macros and constants

type **mpz**

The FLINT multi-precision integer type uses an inline representation for small integers, specifically when the absolute value is at most $2^{62} - 1$ (on 64-bit machines) or $2^{30} - 1$ (on 32-bit machines). It switches automatically to a GMP integer for larger values.

An **mpz** is implemented as an **slong**. When its second most significant bit is 0 the **mpz** represents an ordinary **slong** integer whose absolute value is at most $\text{FLINT\_BITS} - 2$ bits.

When the second most significant bit is 1 then the value represents a pointer (the pointer is shifted right 2 bits and the second most significant bit is set to 1. This relies on the fact that **malloc** always allocates memory blocks on a 4 or 8 byte boundary).

type **mpz_t**

An array of length 1 of **mpz**'s. This is used to pass **mpz**'s around by reference without fuss, similar to the way **mpz_t** works.

COEFF_MAX

The largest (positive) value an **mpz** can be if just an **slong**.

COEFF_MIN

The smallest (negative) value an **mpz** can be if just an **slong**.

mpz **PTR_TO_COEFF**(*mpz_ptr* ptr)

A macro to convert an **mpz_t** (or more generally any **mpz_ptr**) to an **mpz** (shifts the pointer right by 2 and sets the second most significant bit).

mpz_ptr **COEFF_TO_PTR**(*mpz* f)

A macro to convert an **mpz** which represents a pointer into an actual pointer to an **__mpz_struct** (i.e. to an **mpz_t**).

COEFF_IS_MPZ(f)

A macro which returns 1 if *f* represents an **mpz_t**, otherwise 0 is returned.

MPZ_MIN_ALLOC

A constant determining the minimum number of limbs the *mpz* memory manager will allocate for each *mpz*. This constant is currently 2.

mpz_ptr **_mpz_new_mpz**(void)

Returns a pointer to an initialised *mpz* with at least **MPZ_MIN_ALLOC** limbs allocated. This is only used internally.

Note: Requires that **gmp.h** has been included before any FLINT header is included.

void **_mpz_clear_mpz**(*mpz* f)

Clears the *mpz* "pointed to" by the *mpz* *f*. This is only used internally.

Note: As of FLINT 3.2.0, it is required that the *mpz* pointed to by *f* has at least **MPZ_MIN_ALLOC** limbs allocated when **_mpz_clear_mpz()** is called. Note that GMP functions never reduce the number of allocated limbs, apart from a bug in **mpz_remove** which as of GMP 6.3.0 may reduce the number of allocated limbs in the output *mpz*. Hence, as long as **mpz_realloc** never is called with a second argument less than **MPZ_MIN_ALLOC** on an *mpz* received from **_mpz_new_mpz()**, any other usage should be fine.

void **_mpz_cleanup_mpz_content**()

This function does nothing in the reentrant version of **mpz**.

void **_mpz_cleanup**()

This function does nothing in the reentrant version of **mpz**.

`mpz_ptr _fmpz_promote(fmpz_t f)`

If f doesn't represent an `mpz_t`, initialise one and associate it to f .

Note: Requires that `gmp.h` has been included before any FLINT header is included.

`mpz_ptr _fmpz_promote_val(fmpz_t f)`

If f doesn't represent an `mpz_t`, initialise one and associate it to f , but preserve the value of f .

This function is for internal use. The resulting `fmpz` will be backed by an `mpz_t` that can be passed to GMP, but the `fmpz` will be in an inconsistent state with respect to the other FLINT `fmpz` functions such as `fmpz_is_zero`, etc.

Note: Requires that `gmp.h` has been included before any FLINT header is included.

`void _fmpz_demote(fmpz_t f)`

If f represents an `mpz_t` clear it and make f just represent an `slong`.

`void _fmpz_demote_val(fmpz_t f)`

If f represents an `mpz_t` and its value will fit in an `slong`, preserve the value in f which we make to represent an `slong`, and clear the `mpz_t`.

`int _fmpz_is_canonical(const fmpz_t f)`

Returns 1 if the internal representation of f is correctly normalised and demoted; 0 otherwise.

4.2.3 Memory management

`void fmpz_init(fmpz_t f)`

A small `fmpz_t` is initialised, i.e. just a `slong`. The value is set to zero.

`void fmpz_init2(fmpz_t f, ulong limbs)`

Initialises the given `fmpz_t` to have space for the given number of limbs.

If `limbs` is zero then a small `fmpz_t` is allocated, i.e. just a `slong`. The value is also set to zero. It is not necessary to call this function except to save time. A call to `fmpz_init` will do just fine.

`void fmpz_clear(fmpz_t f)`

Clears the given `fmpz_t`, releasing any memory associated with it, either back to the stack or the OS, depending on whether the reentrant or non-reentrant version of FLINT is built.

`void fmpz_init_set(fmpz_t f, const fmpz_t g)`

`void fmpz_init_set_ui(fmpz_t f, ulong g)`

`void fmpz_init_set_si(fmpz_t f, slong g)`

Initialises f and sets it to the value of g .

4.2.4 Random generation

For thread-safety, the randomisation methods take as one of their parameters an object of type `flint_rand_t`. Before calling any of the randomisation functions such an object first has to be initialised with a call to `flint_rand_init()`. When one is finished generating random numbers, one should call `flint_rand_clear()` to clean up.

`void fmpz_randbits_unsigned(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)`

`void fmpz_randbits(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)`

Generates a random integer whose absolute value has precisely the given number of bits.

`void fmpz_randtest_unsigned(fmpz_t f, flint_rand_t state, flint_bitcnt_t bits)`

void **fmpz_randtest**(*fmpz_t* f, *flint_rand_t* state, *flint_bitcnt_t* bits)

Generates a random (unsigned) integer whose absolute value has a number of bits which is random from 0 up to *bits* inclusive.

void **fmpz_randtest_not_zero**(*fmpz_t* f, *flint_rand_t* state, *flint_bitcnt_t* bits)

As per **fmpz_randtest**, but the result will not be 0. If *bits* is set to 0, an exception will result.

void **fmpz_randm**(*fmpz_t* f, *flint_rand_t* state, const *fmpz_t* m)

Generates a random integer in the range 0 to $m - 1$ inclusive.

void **fmpz_randtest_mod**(*fmpz_t* f, *flint_rand_t* state, const *fmpz_t* m)

Generates a random integer in the range 0 to $m - 1$ inclusive, with an increased probability of generating values close to the endpoints.

void **fmpz_randtest_mod_signed**(*fmpz_t* f, *flint_rand_t* state, const *fmpz_t* m)

Generates a random integer in the range $(-m/2, m/2]$, with an increased probability of generating values close to the endpoints or close to zero.

void **fmpz_randprime**(*fmpz_t* f, *flint_rand_t* state, *flint_bitcnt_t* bits, int proved)

Generates a random prime number with the given number of bits.

The generation is performed by choosing a random number and then finding the next largest prime, and therefore does not quite give a uniform distribution over the set of primes with that many bits.

Random number generation is performed using the standard FLINT random number generator, which is not suitable for cryptographic use.

If *proved* is nonzero, then the integer returned is guaranteed to actually be prime.

4.2.5 Conversion

slong **fmpz_get_si**(const *fmpz_t* f)

Returns *f* as a *slong*. The result is undefined if *f* does not fit into a *slong*.

ulong **fmpz_get_ui**(const *fmpz_t* f)

Returns *f* as an *ulong*. The result is undefined if *f* does not fit into an *ulong* or is negative.

void **fmpz_get_uiui**(*ulong* *hi, *ulong* *low, const *fmpz_t* f)

If *f* consists of two limbs, then *hi and *low are set to the high and low limbs, otherwise *low is set to the low limb and *hi is set to 0.

ulong **fmpz_get_nmod**(const *fmpz_t* f, *nmod_t* mod)

Returns *f* modulo the integer given by *mod*.

double **fmpz_get_d**(const *fmpz_t* f)

Returns *f* as a *double*, rounding down towards zero if *f* cannot be represented exactly. The outcome is undefined if *f* is too large to fit in the normal range of a *double*.

void **fmpz_set_mpf**(*fmpz_t* f, const *mpf_t* x)

Sets *f* to the *mpf_t* *x*, rounding down towards zero if the value of *x* is fractional.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **fmpz_get_mpf**(*mpf_t* x, const *fmpz_t* f)

Sets the value of the *mpf_t* *x* to the value of *f*.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **fmpz_get_mpfr**(*mpfr_t* x, const *fmpz_t* f, *mpfr_rnd_t* rnd)

Sets the value of *x* from *f*, rounded toward the given direction *rnd*.

Note: Requires that *mpfr.h* has been included before any FLINT header is included.

double **fmpz_get_d_2exp**(*slong* *exp, const *fmpz_t* f)

Returns f as a normalized double along with a 2-exponent exp , i.e. if r is the return value then $f = r2^{\text{exp}}$, to within 1 ULP.

void **fmpz_get_mpz**(*mpz_t* x, const *fmpz_t* f)

Sets the *mpz_t* x to the same value as f .

Note: Requires that `gmp.h` has been included before any FLINT header is included.

int **fmpz_get_mpn**(*nn_ptr* *n, *fmpz_t* n_in)

Sets the *nn_ptr* n to the same value as n_{in} . Returned integer is number of limbs allocated to n , minimum number of limbs required to hold the value stored in n_{in} .

char ***fmpz_get_str**(char *str, int b, const *fmpz_t* f)

Returns the representation of f in base b , which can vary between 2 and 62, inclusive.

If *str* is NULL, the result string is allocated by the function. Otherwise, it is up to the caller to ensure that the allocated block of memory is sufficiently large.

void **fmpz_set_si**(*fmpz_t* f, *slong* val)

Sets f to the given *slong* value.

void **fmpz_set_ui**(*fmpz_t* f, *ulong* val)

Sets f to the given *ulong* value.

void **fmpz_set_d**(*fmpz_t* f, double c)

Sets f to the double c , rounding down towards zero if the value of c is fractional. The outcome is undefined if c is infinite, not-a-number, or subnormal.

void **fmpz_set_d_2exp**(*fmpz_t* f, double d, *slong* exp)

Sets f to the nearest integer to $d2^{\text{exp}}$.

void **fmpz_neg_ui**(*fmpz_t* f, *ulong* val)

Sets f to the given *ulong* value, and then negates f .

void **fmpz_set_uiui**(*fmpz_t* f, *ulong* hi, *ulong* lo)

Sets f to lo , plus hi shifted to the left by `FLINT_BITS`.

void **fmpz_neg_uiui**(*fmpz_t* f, *ulong* hi, *ulong* lo)

Sets f to lo , plus hi shifted to the left by `FLINT_BITS`, and then negates f .

void **fmpz_set_signed_uiui**(*fmpz_t* f, *ulong* hi, *ulong* lo)

Sets f to lo , plus hi shifted to the left by `FLINT_BITS`, interpreted as a signed two's complement integer with $2 * \text{FLINT\_BITS}$ bits.

void **fmpz_set_signed_uiuiui**(*fmpz_t* f, *ulong* hi, *ulong* mid, *ulong* lo)

Sets f to lo , plus mid shifted to the left by `FLINT_BITS`, plus hi shifted to the left by $2 * \text{FLINT\_BITS}$ bits, interpreted as a signed two's complement integer with $3 * \text{FLINT\_BITS}$ bits.

void **fmpz_set_ui_array**(*fmpz_t* out, const *ulong* *in, *slong* n)

Sets *out* to the nonnegative integer $\text{in}[0] + \text{in}[1]*X + \dots + \text{in}[n-1]*X^{(n-1)}$ where $X = 2^{\text{FLINT\_BITS}}$. It is assumed that $n > 0$.

void **fmpz_set_signed_ui_array**(*fmpz_t* out, const *ulong* *in, *slong* n)

Sets *out* to the integer represented in $\text{in}[0], \dots, \text{in}[n-1]$ as a signed two's complement integer with $n * \text{FLINT\_BITS}$ bits. It is assumed that $n > 0$. The function operates as a call to `fmpz_set_ui_array()` followed by a symmetric remainder modulo $2^{n * \text{FLINT\_BITS}}$.

void **fmpz_get_ui_array**(*ulong* *out, *slong* n, const *fmpz_t* in)

Assuming that the nonnegative integer *in* can be represented in the form $\text{out}[0] + \text{out}[1]*X + \dots + \text{out}[n-1]*X^{(n-1)}$, where $X = 2^{\text{FLINT\_BITS}}$, sets the corresponding elements of *out* so that this is true. It is assumed that $n > 0$.

void **fmpz_get_signed_ui_array**(*ulong* *out, *slong* n, const *fmpz_t* in)

Retrieves the value of *in* modulo $2^{n*FLINT\_BITS}$ and puts the *n* words of the result in *out*[0], . . . , *out*[*n*-1]. This will give a signed two's complement representation of *in* (assuming *in* doesn't overflow the array).

void **fmpz_set_mpn_large**(*fmpz_t* z, *nn_srcptr* src, *slong* n, int negative)

Sets *z* to the integer represented by the *n* limbs in the array *src*, or minus this value if *negative* is 1. Requires $n \geq 2$ and that the top limb of *src* is nonzero. Note that *fmpz_set_ui*, *fmpz_neg_ui* can be used for single-limb integers.

void **fmpz_get_signed_uui**(*ulong* *hi, *ulong* *lo, const *fmpz_t* in)

Retrieves the value of *in* modulo $2^{2*FLINT\_BITS}$ and puts the high and low words into *hi and *lo respectively.

void **fmpz_set_mpz**(*fmpz_t* f, const *mpz_t* x)

Sets *f* to the given *mpz_t* value.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

int **fmpz_set_str**(*fmpz_t* f, const char *str, int b)

Sets *f* to the value given in the null-terminated string *str*, in base *b*. The base *b* can vary between 2 and 62, inclusive. Returns 0 if the string contains a valid input and -1 otherwise.

void **fmpz_set_ui_smod**(*fmpz_t* f, *ulong* x, *ulong* m)

Sets *f* to the signed remainder $y \equiv x \bmod m$ satisfying $-m/2 < y \leq m/2$, given *x* which is assumed to satisfy $0 \leq x < m$.

void **flint_mpz_init_set_readonly**(*mpz_t* z, const *fmpz_t* f)

Sets the uninitialised *mpz_t* *z* to the value of the readonly *fmpz_t* *f*.

Note that it is assumed that *f* does not change during the lifetime of *z*.

The integer *z* has to be cleared by a call to *flint_mpz_clear_readonly*() .

The suggested use of the two functions is as follows:

```
fmpz_t f;
...
{
    mpz_t z;

    flint_mpz_init_set_readonly(z, f);
    foo(..., z);
    flint_mpz_clear_readonly(z);
}
```

This provides a convenient function for user code, only requiring to work with the types *fmpz_t* and *mpz_t*.

In critical code, the following approach may be favourable:

```
fmpz_t f;
...
{
    mpz_ptr z;

    z = _fmpz_promote_val(f);
    foo(..., z);
    _fmpz_demote_val(f);
}
```

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **flint_mpz_clear_readonly**(mpz_t z)

Clears the readonly mpz_t z.

Note: Requires that `gmp.h` has been included before any FLINT header is included.

void **fmpz_init_set_readonly**(fmpz_t f, const mpz_t z)

Sets the uninitialised fmpz_t f to a readonly version of the integer z.

Note that the value of z is assumed to remain constant throughout the lifetime of f.

The fmpz_t f has to be cleared by calling the function `fmpz_clear_readonly()`.

The suggested use of the two functions is as follows:

```
mpz_t z;
...
{
    fmpz_t f;

    fmpz_init_set_readonly(f, z);
    foo(..., f);
    fmpz_clear_readonly(f);
}
```

Note: Requires that `gmp.h` has been included before any FLINT header is included.

void **fmpz_clear_readonly**(fmpz_t f)

Clears the readonly fmpz_t f.

4.2.6 Input and output

int **fmpz_read**(fmpz_t f)

Reads a multiprecision integer from `stdin`. The format is an optional minus sign, followed by one or more digits. The first digit should be non-zero unless it is the only digit.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `scanf` from the standard library and `mpz_inp_str` from GMP.

int **fmpz_fread**(FILE *file, fmpz_t f)

Reads a multiprecision integer from the stream `file`. The format is an optional minus sign, followed by one or more digits. The first digit should be non-zero unless it is the only digit.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

This convention is adopted in light of the return values of `scanf` from the standard library and `mpz_inp_str` from GMP.

size_t **fmpz_inp_raw**(fmpz_t x, FILE *fin)

Reads a multiprecision integer from the stream `file`. The format is raw binary format write by `fmpz_out_raw()`.

In case of success, return a positive number, indicating number of bytes read. In case of failure 0.

This function calls the `mpz_inp_raw` function in library `gmp`. So that it can read the raw data written by `mpz_inp_raw` directly.

int **fmpz_fprint**(FILE *fs, const fmpz_t x)

int **fmpz_print**(const fmpz_t x)

Prints the value `x` to `fs` or `stdout`, without a carriage return. The value is printed as either 0, the decimal digits of a positive integer, or a minus sign followed by the digits of a negative integer.

Returns the number of characters written to file stream.

`size_t fmpz_out_raw(FILE *fout, const fmpz_t x)`

Writes the value x to `file`. The value is written in raw binary format. The integer is written in portable format, with 4 bytes of size information, and that many bytes of limbs. Both the size and the limbs are written in decreasing significance order (i.e., in big-endian).

The output can be read with `fmpz_inp_raw`.

In case of success, return a positive number, indicating number of bytes written. In case of failure, return 0.

The output of this can also be read by `mpz_inp_raw` from GMP, since this function calls the `mpz_inp_raw` function in library gmp.

4.2.7 Basic properties and manipulation

`size_t fmpz_sizeinbase(const fmpz_t f, int b)`

Returns the size of the absolute value of f in base b , measured in numbers of digits. The base b can be between 2 and 62, inclusive.

`flint_bitcnt_t fmpz_bits(const fmpz_t f)`

Returns the number of bits required to store the absolute value of f . If f is 0 then 0 is returned.

`slong fmpz_size(const fmpz_t f)`

Returns the number of limbs required to store the absolute value of f . If f is zero then 0 is returned.

`int fmpz_sgn(const fmpz_t f)`

Returns -1 if the sign of f is negative, $+1$ if it is positive, otherwise returns 0.

`flint_bitcnt_t fmpz_val2(const fmpz_t f)`

Returns the exponent of the largest power of two dividing f , or equivalently the number of trailing zeros in the binary expansion of f . If f is zero then 0 is returned.

`void fmpz_swap(fmpz_t f, fmpz_t g)`

Efficiently swaps f and g . No data is copied.

`void fmpz_set(fmpz_t f, const fmpz_t g)`

Sets f to the same value as g .

`void fmpz_zero(fmpz_t f)`

Sets f to zero.

`void fmpz_one(fmpz_t f)`

Sets f to one.

`int fmpz_abs_fits_ui(const fmpz_t f)`

Returns whether the absolute value of f fits into an `ulong`.

`int fmpz_fits_si(const fmpz_t f)`

Returns whether the value of f fits into a `slong`.

`void fmpz_setbit(fmpz_t f, ulong i)`

Sets bit index i of f .

`int fmpz_tstbit(const fmpz_t f, ulong i)`

Test bit index i of f and return 0 or 1, accordingly.

`ulong fmpz_abs_lbound_ui_2exp(slong *exp, const fmpz_t x, int bits)`

For nonzero x , returns a mantissa m with exactly `bits` bits and sets `exp` to an exponent e , such that $|x| \geq m2^e$. The number of bits must be between 1 and `FLINT_BITS` inclusive. The mantissa is guaranteed to be correctly rounded.

ulong **fmpz_abs_ubound_ui_2exp**(*slong* *exp, const *fmpz_t* x, int bits)

For nonzero x , returns a mantissa m with exactly `bits` bits and sets `exp` to an exponent e , such that $|x| \leq m2^e$. The number of bits must be between 1 and `FLINT_BITS` inclusive. The mantissa is either correctly rounded or one unit too large (possibly meaning that the exponent is one too large, if the mantissa is a power of two).

4.2.8 Comparison

int **fmpz_cmp**(const *fmpz_t* f, const *fmpz_t* g)

int **fmpz_cmp_ui**(const *fmpz_t* f, *ulong* g)

int **fmpz_cmp_si**(const *fmpz_t* f, *slong* g)

Returns a negative value if $f < g$, positive value if $g < f$, otherwise returns 0.

int **fmpz_cmpabs**(const *fmpz_t* f, const *fmpz_t* g)

Returns a negative value if $|f| < |g|$, positive value if $|g| < |f|$, otherwise returns 0.

int **fmpz_cmp2abs**(const *fmpz_t* f, const *fmpz_t* g)

Returns a negative value if $|f| < |2g|$, positive value if $|2g| < |f|$, otherwise returns 0.

int **fmpz_equal**(const *fmpz_t* f, const *fmpz_t* g)

int **fmpz_equal_ui**(const *fmpz_t* f, *ulong* g)

int **fmpz_equal_si**(const *fmpz_t* f, *slong* g)

Returns 1 if f is equal to g , otherwise returns 0.

int **fmpz_is_zero**(const *fmpz_t* f)

Returns 1 if f is 0, otherwise returns 0.

int **fmpz_is_one**(const *fmpz_t* f)

Returns 1 if f is equal to one, otherwise returns 0.

int **fmpz_is_pm1**(const *fmpz_t* f)

Returns 1 if f is equal to one or minus one, otherwise returns 0.

int **fmpz_is_even**(const *fmpz_t* f)

Returns whether the integer f is even.

int **fmpz_is_odd**(const *fmpz_t* f)

Returns whether the integer f is odd.

4.2.9 Basic arithmetic

void **fmpz_neg**(*fmpz_t* f1, const *fmpz_t* f2)

Sets f_1 to $-f_2$.

void **fmpz_abs**(*fmpz_t* f1, const *fmpz_t* f2)

Sets f_1 to the absolute value of f_2 .

void **fmpz_add**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

void **fmpz_add_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)

void **fmpz_add_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)

Sets f to $g + h$.

void **fmpz_sub**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

void **fmpz_sub_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)

void **fmpz_sub_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)
Sets f to $g - h$.

void **fmpz_mul**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_mul_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)
void **fmpz_mul_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)
Sets f to $g \times h$.

void **fmpz_mul2_uiui**(*fmpz_t* f, const *fmpz_t* g, *ulong* x, *ulong* y)
Sets f to $g \times x \times y$ where x and y are of type *ulong*.

void **fmpz_mul_2exp**(*fmpz_t* f, const *fmpz_t* g, *ulong* e)
Sets f to $g \times 2^e$.
Note: Assumes that $e + \text{FLINT\_BITS}$ does not overflow.

void **fmpz_one_2exp**(*fmpz_t* f, *ulong* e)
Sets f to 2^e .

void **fmpz_addmul**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_addmul_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)
void **fmpz_addmul_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)
Sets f to $f + g \times h$.

void **fmpz_submul**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_submul_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)
void **fmpz_submul_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)
Sets f to $f - g \times h$.

void **fmpz_fmms**(*fmpz_t* f, const *fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_t* d)
Sets f to $a \times b + c \times d$.

void **fmpz_fmms**(*fmpz_t* f, const *fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_t* d)
Sets f to $a \times b - c \times d$.

void **fmpz_cdiv_qr**(*fmpz_t* f, *fmpz_t* s, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_fdiv_qr**(*fmpz_t* f, *fmpz_t* s, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_tdiv_qr**(*fmpz_t* f, *fmpz_t* s, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_ndiv_qr**(*fmpz_t* f, *fmpz_t* s, const *fmpz_t* g, const *fmpz_t* h)

void **fmpz_cdiv_q**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_fdiv_q**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)
void **fmpz_tdiv_q**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

void **fmpz_cdiv_q_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)
void **fmpz_fdiv_q_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)
void **fmpz_tdiv_q_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)

void **fmpz_cdiv_q_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)
void **fmpz_fdiv_q_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)
void **fmpz_tdiv_q_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)

void **fmpz_cdiv_q_2exp**(*fmpz_t* f, const *fmpz_t* g, *ulong* exp)

void **fmpz_fdiv_q_2exp**(*fmpz_t* f, const *fmpz_t* g, *ulong* exp)

void **fmpz_tdiv_q_2exp**(*fmpz_t* f, const *fmpz_t* g, *ulong* exp)

void **fmpz_fdiv_r**(*fmpz_t* s, const *fmpz_t* g, const *fmpz_t* h)

void **fmpz_cdiv_r_2exp**(*fmpz_t* s, const *fmpz_t* g, *ulong* exp)

void **fmpz_fdiv_r_2exp**(*fmpz_t* s, const *fmpz_t* g, *ulong* exp)

void **fmpz_tdiv_r_2exp**(*fmpz_t* s, const *fmpz_t* g, *ulong* exp)

Sets f to the quotient of g by h and/or s to the remainder. For the **2exp** functions, $h = 2^{\text{exp}}$. If h is 0 an exception is raised.

Rounding is made in the following way:

- **fdiv** rounds the quotient via floor rounding.
- **cdiv** rounds the quotient via ceil rounding.
- **tdiv** rounds the quotient via truncation, i.e. rounding towards zero.
- **ndiv** rounds the quotient such that the remainder has the smallest absolute value. In case of ties, it rounds the quotient towards zero.

ulong **fmpz_cdiv_ui**(const *fmpz_t* g, *ulong* h)

ulong **fmpz_fdiv_ui**(const *fmpz_t* g, *ulong* h)

ulong **fmpz_tdiv_ui**(const *fmpz_t* g, *ulong* h)

Returns the absolute value remainder of g divided by h , following the convention of rounding as seen above. If h is zero an exception is raised.

void **fmpz_divexact**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

void **fmpz_divexact_si**(*fmpz_t* f, const *fmpz_t* g, *slong* h)

void **fmpz_divexact_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)

Sets f to the quotient of g and h , assuming that the division is exact, i.e. g is a multiple of h . If h is 0 an exception is raised.

void **fmpz_divexact2_uiui**(*fmpz_t* f, const *fmpz_t* g, *ulong* x, *ulong* y)

Sets f to the quotient of g and $h = x \times y$, assuming that the division is exact, i.e. g is a multiple of h . If x or y is 0 an exception is raised.

int **fmpz_divisible**(const *fmpz_t* f, const *fmpz_t* g)

int **fmpz_divisible_si**(const *fmpz_t* f, *slong* g)

Returns 1 if there is an integer q with $f = qg$ and 0 if there is none.

int **fmpz_divides**(*fmpz_t* q, const *fmpz_t* f, const *fmpz_t* g)

Returns 1 if there is an integer q with $f = qg$ and sets q to the quotient. Otherwise returns 0 and sets q to 0.

void **fmpz_mod**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

Sets f to the remainder of g divided by h such that the remainder is non-negative. Assumes that h is not zero.

ulong **fmpz_mod_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)

Sets f to the remainder of g divided by h such that the remainder is non-negative and also returns this value. Raises an exception if h is zero.

void **fmpz_smod**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

Sets f to the signed remainder $y \equiv g \bmod h$ satisfying $-|h|/2 < y \leq |h|/2$.

void **fmpz_preinvn_init**(fmpz_preinvn_t inv, const fmpz_t f)

Compute a precomputed inverse *inv* of *f* for use in the **preinvn** functions listed below.

void **fmpz_preinvn_clear**(fmpz_preinvn_t inv)

Clean up the resources used by a precomputed inverse created with the **fmpz_preinvn_init()** function.

void **fmpz_fdiv_qr_preinvn**(fmpz_t f, fmpz_t s, const fmpz_t g, const fmpz_t h, const fmpz_preinvn_t hin)

As per **fmpz_fdiv_qr()**, but takes a precomputed inverse *hin* of *h* constructed using **fmpz_preinvn()**.

void **fmpz_pow_ui**(fmpz_t f, const fmpz_t g, ulong x)

void **fmpz_ui_pow_ui**(fmpz_t f, ulong g, ulong x)

Sets *f* to g^x . Defines $0^0 = 1$.

int **fmpz_pow_fmpz**(fmpz_t f, const fmpz_t g, const fmpz_t x)

Sets *f* to g^x . Defines $0^0 = 1$. Return 1 for success and 0 for failure. The function throws only if *x* is negative.

void **fmpz_powm_ui**(fmpz_t f, const fmpz_t g, ulong e, const fmpz_t m)

void **fmpz_powm**(fmpz_t f, const fmpz_t g, const fmpz_t e, const fmpz_t m)

Sets *f* to $g^e \bmod m$. If *e* = 0, sets *f* to 1.

Assumes that $m \neq 0$, raises an **abort** signal otherwise.

slong **fmpz_clog**(const fmpz_t x, const fmpz_t b)

slong **fmpz_clog_ui**(const fmpz_t x, ulong b)

Returns $\lfloor \log_b x \rfloor$.

Assumes that $x \geq 1$ and $b \geq 2$ and that the return value fits into a signed **slong**.

slong **fmpz_flog**(const fmpz_t x, const fmpz_t b)

slong **fmpz_flog_ui**(const fmpz_t x, ulong b)

Returns $\lfloor \log_b x \rfloor$.

Assumes that $x \geq 1$ and $b \geq 2$ and that the return value fits into a signed **slong**.

double **fmpz_dlog**(const fmpz_t x)

Returns a double precision approximation of the natural logarithm of *x*.

The accuracy depends on the implementation of the floating-point logarithm provided by the C standard library. The result can typically be expected to have a relative error no greater than 1-2 bits.

int **fmpz_sqrtmod**(fmpz_t b, const fmpz_t a, const fmpz_t p)

If *p* is prime, set *b* to a square root of *a* modulo *p* if *a* is a quadratic residue modulo *p* and return 1, otherwise return 0.

If *p* is not prime the return value is with high probability 0, indicating that *p* is not prime, or *a* is not a square modulo *p*. If *p* is not prime and the return value is 1, the value of *b* is meaningless.

void **fmpz_sqrt**(fmpz_t f, const fmpz_t g)

Sets *f* to the integer part of the square root of *g*, where *g* is assumed to be non-negative. If *g* is negative, an exception is raised.

void **fmpz_sqrtrem**(fmpz_t f, fmpz_t r, const fmpz_t g)

Sets *f* to the integer part of the square root of *g*, where *g* is assumed to be non-negative, and sets *r* to the remainder, that is, the difference $g - f^2$. If *g* is negative, an exception is raised. The behaviour is undefined if *f* and *r* are aliases.

int **fmpz_is_square**(const *fmpz_t* f)

Returns nonzero if f is a perfect square and zero otherwise.

int **fmpz_root**(*fmpz_t* r, const *fmpz_t* f, *slong* n)

Set r to the integer part of the n -th root of f . Requires that $n > 0$ and that if n is even then f be non-negative, otherwise an exception is raised. The function returns 1 if the root was exact, otherwise 0.

int **fmpz_is_perfect_power**(*fmpz_t* root, const *fmpz_t* f)

If f is a perfect power r^k set **root** to r and return k , otherwise return 0. Note that $-1, 0, 1$ are all considered perfect powers. No guarantee is made about r or k being the smallest possible value. Negative values of f are permitted.

void **fmpz_fac_ui**(*fmpz_t* f, *ulong* n)

Sets f to the factorial $n!$ where n is an *ulong*.

void **fmpz_fib_ui**(*fmpz_t* f, *ulong* n)

Sets f to the Fibonacci number F_n where n is an *ulong*.

void **fmpz_bin_uiui**(*fmpz_t* f, *ulong* n, *ulong* k)

Sets f to the binomial coefficient $\binom{n}{k}$.

void **_fmpz_rfai**(*fmpz_t* r, const *fmpz_t* x, *ulong* a, *ulong* b)

Sets r to the rising factorial $(x+a)(x+a+1)(x+a+2)\cdots(x+b-1)$. Assumes $b > a$.

void **fmpz_rfai**(*fmpz_t* r, const *fmpz_t* x, *ulong* k)

Sets r to the rising factorial $x(x+1)(x+2)\cdots(x+k-1)$.

void **fmpz_rfai_ui**(*fmpz_t* r, *ulong* x, *ulong* k)

Sets r to the rising factorial $x(x+1)(x+2)\cdots(x+k-1)$.

void **fmpz_mul_tdiv_q_2exp**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h, *ulong* exp)

Sets f to the product of g and h divided by 2^{exp} , rounding down towards zero.

void **fmpz_mul_si_tdiv_q_2exp**(*fmpz_t* f, const *fmpz_t* g, *slong* x, *ulong* exp)

Sets f to the product of g and x divided by 2^{exp} , rounding down towards zero.

4.2.10 Greatest common divisor

void **fmpz_gcd_ui**(*fmpz_t* f, const *fmpz_t* g, *ulong* h)

void **fmpz_gcd**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

Sets f to the greatest common divisor of g and h . The result is always non-negative, even if one of g and h is negative.

void **fmpz_gcd3**(*fmpz_t* f, const *fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c)

Sets f to the greatest common divisor of a , b and c . This is equivalent to calling **fmpz_gcd** twice, but may be faster.

void **fmpz_lcm**(*fmpz_t* f, const *fmpz_t* g, const *fmpz_t* h)

Sets f to the least common multiple of g and h . The result is always nonnegative, even if one of g and h is negative.

void **fmpz_gcdinv**(*fmpz_t* d, *fmpz_t* a, const *fmpz_t* f, const *fmpz_t* g)

Given integers f, g with $0 \leq f < g$, computes the greatest common divisor $d = \gcd(f, g)$ and the modular inverse $a = f^{-1} \pmod{g}$, whenever $f \neq 0$.

Assumes that d and a are not aliased.

void **fmpz_xgcd**(*fmpz_t* d, *fmpz_t* a, *fmpz_t* b, const *fmpz_t* f, const *fmpz_t* g)

Computes the extended GCD of f and g , i.e. the values a and b such that $af + bg = d$, where $d = \gcd(f, g)$. Here a will be the same as calling **fmpz_gcdinv** when $f < g$ (or vice versa for b when $g < f$).

To obtain the canonical solution to Bézout's identity, call **fmpz_xgcd_canonical_bezout** instead. This is also faster.

Assumes that there is no aliasing among the outputs.

void **fmpz_xgcd_canonical_bezout**(*fmpz_t* d, *fmpz_t* a, *fmpz_t* b, const *fmpz_t* f, const *fmpz_t* g)

Computes the extended GCD $\text{xgcd}(f, g) = (d, a, b)$ such that the solution is the canonical solution to Bézout's identity. We define the canonical solution to satisfy one of the following if one of the given conditions apply:

$$\begin{aligned}\text{xgcd}(\pm g, g) &= (|g|, 0, \text{sgn}(g)) \\ \text{xgcd}(f, 0) &= (|f|, \text{sgn}(f), 0) \\ \text{xgcd}(0, g) &= (|g|, 0, \text{sgn}(g)) \\ \text{xgcd}(f, \mp 1) &= (1, 0, \mp 1) \\ \text{xgcd}(\mp 1, g) &= (1, \mp 1, 0) \quad g \neq 0, \pm 1 \\ \text{xgcd}(\mp 2d, g) &= (d, \frac{d-|g|}{\mp 2d}, \text{sgn}(g)) \\ \text{xgcd}(f, \mp 2d) &= (d, \text{sgn}(f), \frac{d-|g|}{\mp 2d}).\end{aligned}$$

If the pair (f, g) does not satisfy any of these conditions, the solution (d, a, b) will satisfy the following:

$$|a| < \left\lfloor \frac{g}{2d} \right\rfloor, \quad |b| < \left\lfloor \frac{f}{2d} \right\rfloor.$$

Assumes that there is no aliasing among the outputs.

void **fmpz_xgcd_partial**(*fmpz_t* co2, *fmpz_t* co1, *fmpz_t* r2, *fmpz_t* r1, const *fmpz_t* L)

This function is an implementation of Lehmer extended GCD with early termination, as used in the **qfb** module. It terminates early when remainders fall below the specified bound. The initial values **r1** and **r2** are treated as successive remainders in the Euclidean algorithm and are replaced with the last two remainders computed. The values **co1** and **co2** are the last two cofactors and satisfy the identity $\text{co2} * \text{r1} - \text{co1} * \text{r2} == \pm \text{r2\_orig}$ upon termination, where **r2_orig** is the starting value of **r2** supplied, and **r1** and **r2** are the final values.

Aliasing of inputs is not allowed. Similarly aliasing of inputs and outputs is not allowed.

4.2.11 Modular arithmetic

slong **fmpz_remove**(*fmpz_t* x, const *fmpz_t* f, double finv)

Removes all factors f from x and returns the number of such.

Assumes that x is non-zero, that $f > 1$ and that **finv** is the precomputed **double** inverse of f whenever f is a small integer and 0 otherwise.

Does not support aliasing.

slong **fmpz_remove**(*fmpz_t* rop, const *fmpz_t* op, const *fmpz_t* f)

Remove all occurrences of the factor $f > 1$ from the integer **op** and sets **rop** to the resulting integer.

If **op** is zero, sets **rop** to **op** and returns 0.

Returns an **abort** signal if any of the assumptions are violated.

```
int fmpz_invmod(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to the inverse of g modulo h . The value of h may not be 0 otherwise an exception results. If the inverse exists the return value will be non-zero, otherwise the return value will be 0 and the value of f undefined. As a special case, we consider any number invertible modulo $h = \pm 1$, with inverse 0.

```
void fmpz_negmod(fmpz_t f, const fmpz_t g, const fmpz_t h)
```

Sets f to $-g \pmod{h}$, assuming g is reduced modulo h .

```
int fmpz_jacobi(const fmpz_t a, const fmpz_t n)
```

Computes the Jacobi symbol $\left(\frac{a}{n}\right)$ for any a and odd positive n .

```
int fmpz_kronecker(const fmpz_t a, const fmpz_t n)
```

Computes the Kronecker symbol $\left(\frac{a}{n}\right)$ for any a and any n .

```
void fmpz_divides_mod_list(fmpz_t xstart, fmpz_t xstride, fmpz_t xlength, const fmpz_t a, const fmpz_t b, const fmpz_t n)
```

Set $xstart$, $xstride$, and $xlength$ so that the solution set for x modulo n in $ax = b \pmod{n}$ is exactly $\{xstart + xstride i \mid 0 \leq i < xlength\}$. This function essentially gives a list of possibilities for the fraction a/b modulo n . The outputs may not be aliased, and n should be positive.

4.2.12 Bit packing and unpacking

```
int fmpz_bit_pack(ulong *arr, flint_bitcnt_t shift, flint_bitcnt_t bits, const fmpz_t coeff, int negate, int borrow)
```

Shifts the given coefficient to the left by `shift` bits and adds it to the integer in `arr` in a field of the given number of bits:

```
shift  bits  -----
X X X C C C C 0 0 0 0 0 0 0
```

An optional borrow of 1 can be subtracted from `coeff` before it is packed. If `coeff` is negative after the borrow, then a borrow will be returned by the function.

The value of `shift` is assumed to be less than `FLINT_BITS`. All but the first `shift` bits of `arr` are assumed to be zero on entry to the function.

The value of `coeff` may also be optionally (and notionally) negated before it is used, by setting the `negate` parameter to `-1`.

```
int fmpz_bit_unpack(fmpz_t coeff, ulong *arr, flint_bitcnt_t shift, flint_bitcnt_t bits, int negate, int borrow)
```

A bit field of the given number of bits is extracted from `arr`, starting after `shift` bits, and placed into `coeff`. An optional borrow of 1 may be added to the coefficient. If the result is negative, a borrow of 1 is returned. Finally, the resulting `coeff` may be negated by setting the `negate` parameter to `-1`.

The value of `shift` is expected to be less than `FLINT_BITS`.

```
void fmpz_bit_unpack_unsigned(fmpz_t coeff, const ulong *arr, flint_bitcnt_t shift, flint_bitcnt_t bits)
```

A bit field of the given number of bits is extracted from `arr`, starting after `shift` bits, and placed into `coeff`.

The value of `shift` is expected to be less than `FLINT_BITS`.

4.2.13 Logic Operations

void **fmprz_complement**(*fmprz_t* r, const *fmprz_t* f)

The variable *r* is set to the ones-complement of *f*.

void **fmprz_clrbit**(*fmprz_t* f, *ulong* i)

Sets the *i*th bit in *f* to zero.

void **fmprz_combit**(*fmprz_t* f, *ulong* i)

Complements the *i*th bit in *f*.

void **fmprz_and**(*fmprz_t* r, const *fmprz_t* a, const *fmprz_t* b)

Sets *r* to the bit-wise logical **and** of *a* and *b*.

void **fmprz_or**(*fmprz_t* r, const *fmprz_t* a, const *fmprz_t* b)

Sets *r* to the bit-wise logical (inclusive) **or** of *a* and *b*.

void **fmprz_xor**(*fmprz_t* r, const *fmprz_t* a, const *fmprz_t* b)

Sets *r* to the bit-wise logical exclusive **or** of *a* and *b*.

ulong **fmprz_popcnt**(const *fmprz_t* a)

Returns the number of ‘1’ bits in the given *Z* (aka Hamming weight or population count). The return value is undefined if the input is negative.

4.2.14 Chinese remaindering

The following functions can be used to reconstruct an integer from its residues modulo a set of prime numbers. The first two functions, *fmprz_CRT_ui()* and *fmprz_CRT()*, are easy to use and allow building the result one residue at a time, which is useful when the number of needed primes is not known in advance. The remaining functions support performing the modular reductions and reconstruction using balanced subdivision. This greatly improves efficiency for large integers but assumes that the basis of primes is known in advance. The user must precompute a **comb** structure and temporary working space with *fmprz_comb_init()* and *fmprz_comb_temp_init()*, and free this data afterwards. For simple demonstration programs showing how to use the CRT functions, see *crt.c* and *multi_crt.c* in the *examples* directory. The *fmprz_multi_CRT* class is similar to *fmprz_multi_CRT_ui* except that it performs error checking and works with arbitrary moduli.

void **fmprz_CRT_ui**(*fmprz_t* out, const *fmprz_t* r1, const *fmprz_t* m1, *ulong* r2, *ulong* m2, int sign)

Uses the Chinese Remainder Theorem to compute the unique integer $0 \leq x < M$ (if *sign* = 0) or $-M/2 < x \leq M/2$ (if *sign* = 1) congruent to *r*₁ modulo *m*₁ and *r*₂ modulo *m*₂, where $M = m_1 \times m_2$. The result *x* is stored in *out*.

It is assumed that *m*₁ and *m*₂ are positive coprime integers.

If *sign* = 0, it is assumed that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. Otherwise, it is assumed that $-m_1 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$.

void **fmprz_CRT**(*fmprz_t* out, const *fmprz_t* r1, const *fmprz_t* m1, const *fmprz_t* r2, const *fmprz_t* m2, int sign)

Use the Chinese Remainder Theorem to set *out* to the unique value $0 \leq x < M$ (if *sign* = 0) or $-M/2 < x \leq M/2$ (if *sign* = 1) congruent to *r*₁ modulo *m*₁ and *r*₂ modulo *m*₂, where $M = m_1 \times m_2$.

It is assumed that *m*₁ and *m*₂ are positive coprime integers.

If *sign* = 0, it is assumed that $0 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$. Otherwise, it is assumed that $-m_1 \leq r_1 < m_1$ and $0 \leq r_2 < m_2$.

void **fmprz_multi_mod_ui**(*ulong* *out, const *fmprz_t* in, const *fmprz_comb_t* comb, *fmprz_comb_temp_t* temp)

Reduces the multiprecision integer *in* modulo each of the primes stored in the **comb** structure. The array *out* will be filled with the residues modulo these primes. The structure

`temp` is temporary space which must be provided by `fmpz_comb_temp_init()` and cleared by `fmpz_comb_temp_clear()`.

void **fmpz_multi_CRT_ui**(*fmpz_t* output, *nn_srcptr* residues, const *fmpz_comb_t* comb, *fmpz_comb_temp_t* ctemp, int sign)

This function takes a set of residues modulo the list of primes contained in the `comb` structure and reconstructs a multiprecision integer modulo the product of the primes which has these residues modulo the corresponding primes.

If N is the product of all the primes then `out` is normalised to be in the range $[0, N)$ if `sign = 0` and the range $[-(N-1)/2, N/2]$ if `sign = 1`. The array `temp` is temporary space which must be provided by `fmpz_comb_temp_init()` and cleared by `fmpz_comb_temp_clear()`.

void **fmpz_comb_init**(*fmpz_comb_t* comb, *nn_srcptr* primes, *slong* num_primes)

Initialises a `comb` structure for multimodular reduction and recombination. The array `primes` is assumed to contain `num_primes` primes each of `FLINT_BITS - 1` bits. Modular reductions and recombinations will be done modulo this list of primes. The `primes` array must not be `free`'d until the `comb` structure is no longer required and must be cleared by the user.

void **fmpz_comb_temp_init**(*fmpz_comb_temp_t* temp, const *fmpz_comb_t* comb)

Creates temporary space to be used by multimodular and CRT functions based on an initialised `comb` structure.

void **fmpz_comb_clear**(*fmpz_comb_t* comb)

Clears the given `comb` structure, releasing any memory it uses.

void **fmpz_comb_temp_clear**(*fmpz_comb_temp_t* temp)

Clears temporary space `temp` used by multimodular and CRT functions using the given `comb` structure.

void **fmpz_multi_CRT_init**(*fmpz_multi_CRT_t* CRT)

Initialize CRT for Chinese remaindering.

int **fmpz_multi_CRT_precompute**(*fmpz_multi_CRT_t* CRT, const *fmpz* *moduli, *slong* len)

Configure CRT for repeated Chinese remaindering of `moduli`. The number of moduli, `len`, should be positive. A return of 0 indicates that the compilation failed and future calls to `fmpz_multi_CRT_precomp()` will leave the output undefined. A return of 1 indicates that the compilation was successful, which occurs if and only if either (1) `len == 1` and `modulus + 0` is nonzero, or (2) no modulus is 0, 1, -1 and all moduli are pairwise relatively prime.

void **fmpz_multi_CRT_precomp**(*fmpz_t* output, const *fmpz_multi_CRT_t* P, const *fmpz* *inputs, int sign)

Set output to an integer of smallest absolute value that is congruent to `values + i` modulo the `moduli + i` in `P`.

int **fmpz_multi_CRT**(*fmpz_t* output, const *fmpz* *moduli, const *fmpz* *values, *slong* len, int sign)

Perform the same operation as `fmpz_multi_CRT_precomp()` while internally constructing and destroying the precomputed data. All of the remarks in `fmpz_multi_CRT_precompute()` apply.

void **fmpz_multi_CRT_clear**(*fmpz_multi_CRT_t* P)

Free all space used by CRT.

4.2.15 Primality testing

int **fmpz_is_strong_probabprime**(const *fmpz_t* n, const *fmpz_t* a)

Returns 1 if n is a strong probable prime to base a , otherwise it returns 0.

int **fmpz_is_probabprime_lucas**(const *fmpz_t* n)

Performs a Lucas probable prime test with parameters chosen by Selfridge's method A as per [BaiWag1980].

Return 1 if n is a Lucas probable prime, otherwise return 0. This function declares some composites probably prime, but no primes composite.

int **fmpz_is_probabprime_BPSW**(const *fmpz_t* n)

Perform a Baillie-PSW probable prime test with parameters chosen by Selfridge's method A as per [BaiWag1980].

Return 1 if n is a Lucas probable prime, otherwise return 0.

There are no known composites passed as prime by this test, though infinitely many probably exist. The test will declare no primes composite.

int **fmpz_is_probabprime**(const *fmpz_t* p)

Performs some trial division and then some probabilistic primality tests. If p is definitely composite, the function returns 0, otherwise it is declared probably prime, i.e. prime for most practical purposes, and the function returns 1. The chance of declaring a composite prime is very small.

Subsequent calls to the same function do not increase the probability of the number being prime.

int **fmpz_is_prime_pseudosquare**(const *fmpz_t* n)

Return 0 if n is composite. If n is too large (greater than about 94 bits) the function fails silently and returns -1 , otherwise, if n is proven prime by the pseudosquares method, return 1.

Tests if n is a prime according to Theorem 2.7 in [LukPatWil1996].

We first factor N using trial division up to some limit B . In fact, the number of primes used in the trial factoring is at most `FLINT_PSEUDOSQUARES_CUTOFF`.

Next we compute N/B and find the next pseudosquare L_p above this value, using a static table as per <https://oeis.org/A002189/b002189.txt>.

As noted in the text, if p is prime then Step 3 will pass. This test rejects many composites, and so by this time we suspect that p is prime. If N is 3 or 7 modulo 8, we are done, and N is prime.

We now run a probable prime test, for which no known counterexamples are known, to reject any composites. We then proceed to prove N prime by executing Step 4. In the case that N is 1 modulo 8, if Step 4 fails, we extend the number of primes p_i at Step 3 and hope to find one which passes Step 4. We take the test one past the largest p for which we have pseudosquares L_p tabulated, as this already corresponds to the next L_p which is bigger than 2^{64} and hence larger than any prime we might be testing.

As explained in the text, Condition 4 cannot fail if N is prime.

The possibility exists that the probable prime test declares a composite prime. However in that case an error is printed, as that would be of independent interest.

int **fmpz_is_prime_pocklington**(*fmpz_t* F, *fmpz_t* R, const *fmpz_t* n, *nn_ptr* pm1, *slong* num_pm1)

Applies the Pocklington primality test. The test computes a product F of prime powers which divide $n - 1$.

The function then returns either 0 if n is definitely composite or it returns 1 if all factors of n are 1 (mod F). Also in that case, R is set to $(n - 1)/F$.

NB: a return value of 1 only proves n prime if $F \geq \sqrt{n}$.

The function does not compute which primes divide $n - 1$. Instead, these must be supplied as an array `pm1` of length `num_pm1`. It does not matter how many prime factors are supplied, but the more that are supplied, the larger F will be.

There is a balance between the amount of time spent looking for factors of $n - 1$ and the usefulness of the output (F may be as low as 2 in some cases).

A reasonable heuristic seems to be to choose `limit` to be some small multiple of $\log^3(n)/10$ (e.g. 1, 2, 5 or 10) depending on how long one is prepared to wait, then to trial factor up to the limit. (See `_fmpz_nm1_trial_factors`.)

Requires n to be odd.

```
void _fmpz_nm1_trial_factors(const fmpz_t n, nn_ptr pm1, slong *num_pm1, ulong limit)
```

Trial factors $n - 1$ up to the given limit (approximately) and stores the factors in an array `pm1` whose length is written out to `num_pm1`.

One can use $\log(n) + 2$ as a bound on the number of factors which might be produced (and hence on the length of the array that needs to be supplied).

```
int fmpz_is_prime_morrison(fmpz_t F, fmpz_t R, const fmpz_t n, nn_ptr pp1, slong num_pp1)
```

Applies the Morrison $p + 1$ primality test. The test computes a product F of primes which divide $n + 1$.

The function then returns either 0 if n is definitely composite or it returns 1 if all factors of n are $\pm 1 \pmod{F}$. Also in that case, R is set to $(n + 1)/F$.

NB: a return value of 1 only proves n prime if $F > \sqrt{n} + 1$.

The function does not compute which primes divide $n + 1$. Instead, these must be supplied as an array `pp1` of length `num_pp1`. It does not matter how many prime factors are supplied, but the more that are supplied, the larger F will be.

There is a balance between the amount of time spent looking for factors of $n + 1$ and the usefulness of the output (F may be as low as 2 in some cases).

A reasonable heuristic seems to be to choose `limit` to be some small multiple of $\log^3(n)/10$ (e.g. 1, 2, 5 or 10) depending on how long one is prepared to wait, then to trial factor up to the limit. (See `_fmpz_np1_trial_factors`.)

Requires n to be odd and non-square.

```
void _fmpz_np1_trial_factors(const fmpz_t n, nn_ptr pp1, slong *num_pp1, ulong limit)
```

Trial factors $n + 1$ up to the given limit (approximately) and stores the factors in an array `pp1` whose length is written out to `num_pp1`.

One can use $\log(n) + 2$ as a bound on the number of factors which might be produced (and hence on the length of the array that needs to be supplied).

```
int fmpz_is_prime(const fmpz_t n)
```

Attempts to prove n prime. If n is proven prime, the function returns 1. If n is definitely composite, the function returns 0.

This function calls `n_is_prime()` for n that fits in a single word. For n larger than one word, it tests divisibility by a few small primes and whether n is a perfect square to rule out trivial composites. For n up to about 81 bits, it then uses a strong probable prime test (Miller-Rabin test) with the first 13 primes as witnesses. This has been shown to prove primality [SorWeb2016].

For larger n , it does a single base-2 strong probable prime test to eliminate most composite numbers. If n passes, it does a combination of Pocklington, Morrison and Brillhart, Lehmer, Selfridge tests. If any of these tests fails to give a proof, it falls back to performing an APRCL test.

The APRCL test could theoretically fail to prove that n is prime or composite. In that case, the program aborts. This is not expected to occur in practice.

void **fmpz_lucas_chain**(*fmpz_t* Vm, *fmpz_t* Vm1, const *fmpz_t* A, const *fmpz_t* m, const *fmpz_t* n)
 Given $V_0 = 2$, $V_1 = A$ compute $V_m, V_{m+1} \pmod n$ from the recurrences $V_j = AV_{j-1} - V_{j-2} \pmod n$.

This is computed efficiently using $V_{2j} = V_j^2 - 2 \pmod n$ and $V_{2j+1} = V_j V_{j+1} - A \pmod n$.

No aliasing is permitted.

void **fmpz_lucas_chain_full**(*fmpz_t* Vm, *fmpz_t* Vm1, const *fmpz_t* A, const *fmpz_t* B, const *fmpz_t* m, const *fmpz_t* n)

Given $V_0 = 2$, $V_1 = A$ compute $V_m, V_{m+1} \pmod n$ from the recurrences $V_j = AV_{j-1} - BV_{j-2} \pmod n$.

This is computed efficiently using double and add formulas.

No aliasing is permitted.

void **fmpz_lucas_chain_double**(*fmpz_t* U2m, *fmpz_t* U2m1, const *fmpz_t* Um, const *fmpz_t* Um1, const *fmpz_t* A, const *fmpz_t* B, const *fmpz_t* n)

Given $U_m, U_{m+1} \pmod n$ compute $U_{2m}, U_{2m+1} \pmod n$.

Aliasing of U_{2m} and U_m and aliasing of U_{2m+1} and U_{m+1} is permitted. No other aliasing is allowed.

void **fmpz_lucas_chain_add**(*fmpz_t* Umn, *fmpz_t* Umn1, const *fmpz_t* Um, const *fmpz_t* Um1, const *fmpz_t* Un, const *fmpz_t* Un1, const *fmpz_t* A, const *fmpz_t* B, const *fmpz_t* n)

Given $U_m, U_{m+1} \pmod n$ and $U_n, U_{n+1} \pmod n$ compute $U_{m+n}, U_{m+n+1} \pmod n$.

Aliasing of U_{m+n} with U_m or U_n and aliasing of U_{m+n+1} with U_{m+1} or U_{n+1} is permitted. No other aliasing is allowed.

void **fmpz_lucas_chain_mul**(*fmpz_t* Ukm, *fmpz_t* Ukm1, const *fmpz_t* Um, const *fmpz_t* Um1, const *fmpz_t* A, const *fmpz_t* B, const *fmpz_t* k, const *fmpz_t* n)

Given $U_m, U_{m+1} \pmod n$ compute $U_{km}, U_{km+1} \pmod n$.

Aliasing of U_{km} and U_m and aliasing of U_{km+1} and U_{m+1} is permitted. No other aliasing is allowed.

void **fmpz_lucas_chain_VtoU**(*fmpz_t* Um, *fmpz_t* Um1, const *fmpz_t* Vm, const *fmpz_t* Vm1, const *fmpz_t* A, const *fmpz_t* B, const *fmpz_t* Dinv, const *fmpz_t* n)

Given $V_m, V_{m+1} \pmod n$ compute $U_m, U_{m+1} \pmod n$.

Aliasing of V_m and U_m and aliasing of V_{m+1} and U_{m+1} is permitted. No other aliasing is allowed.

int **fmpz_divisor_in_residue_class_lenstra**(*fmpz_t* fac, const *fmpz_t* n, const *fmpz_t* r, const *fmpz_t* s)

If there exists a proper divisor of n which is $r \pmod s$ for $0 < r < s < n$, this function returns 1 and sets **fac** to such a divisor. Otherwise the function returns 0 and the value of **fac** is undefined.

We require $\gcd(r, s) = 1$.

This is efficient if $s^3 > n$.

void **fmpz_nextprime**(*fmpz_t* res, const *fmpz_t* n, int proved)

Finds the next prime number larger than n .

If **proved** is nonzero, then the integer returned is guaranteed to actually be prime. Otherwise if n fits in `FLINT_BITS - 3` bits **n_nextprime** is called, and if not then the GMP `mpz_nextprime` function is called which uses a BPSW test.

4.2.16 Special functions

void **mpz_primorial**(*mpz_t* res, *ulong* n)

Sets **res** to **n** primorial or $n\#$, the product of all prime numbers less than or equal to n .

void **mpz_factor_euler_phi**(*mpz_t* res, const *mpz_factor_t* fac)

void **mpz_euler_phi**(*mpz_t* res, const *mpz_t* n)

Sets **res** to the Euler totient function $\phi(n)$, counting the number of positive integers less than or equal to n that are coprime to n . The factor version takes a precomputed factorisation of n .

int **mpz_factor_moebius_mu**(const *mpz_factor_t* fac)

int **mpz_moebius_mu**(const *mpz_t* n)

Computes the Moebius function $\mu(n)$, which is defined as $\mu(n) = 0$ if n has a prime factor of multiplicity greater than 1, $\mu(n) = -1$ if n has an odd number of distinct prime factors, and $\mu(n) = 1$ if n has an even number of distinct prime factors. By convention, $\mu(0) = 0$. The factor version takes a precomputed factorisation of n .

void **mpz_factor_divisor_sigma**(*mpz_t* res, *ulong* k, const *mpz_factor_t* fac)

void **mpz_divisor_sigma**(*mpz_t* res, *ulong* k, const *mpz_t* n)

Sets **res** to $\sigma_k(n)$, the sum of k th powers of all divisors of n . The factor version takes a precomputed factorisation of n .

4.3 mpz_vec.h – vectors of integers

4.3.1 Memory management

mpz *_**mpz_vec_init**(*slong* len)

Returns an initialised vector of **mpz**'s of given length.

void **_mpz_vec_clear**(*mpz* *vec, *slong* len)

Clears the entries of (**vec**, **len**) and frees the space allocated for **vec**.

4.3.2 Randomisation

void **_mpz_vec_randtest**(*mpz* *f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets the entries of a vector of the given length to random integers with up to the given number of bits per entry.

void **_mpz_vec_randtest_unsigned**(*mpz* *f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets the entries of a vector of the given length to random unsigned integers with up to the given number of bits per entry.

4.3.3 Bit sizes and norms

slong **_mpz_vec_max_bits**(const *mpz* *vec, *slong* len)

If b is the maximum number of bits of the absolute value of any coefficient of **vec**, then if any coefficient of **vec** is negative, $-b$ is returned, else b is returned.

slong **_mpz_vec_max_bits_ref**(const *mpz* *vec, *slong* len)

If b is the maximum number of bits of the absolute value of any coefficient of **vec**, then if any coefficient of **vec** is negative, $-b$ is returned, else b is returned. This is a slower reference implementation of **_mpz_vec_max_bits**.

void **_fmpz_vec_sum_max_bits**(*slong* *sumabs, *slong* *maxabs, const *fmpz* *vec, *slong* len)

Sets *sumabs* to the bit count of the sum of the absolute values of the elements of *vec*. Sets *maxabs* to the bit count of the maximum of the absolute values of the elements of *vec*.

slong **_fmpz_vec_max_limbs**(const *fmpz* *vec, *slong* len)

Returns the maximum number of limbs needed to store the absolute value of any entry in (*vec*, *len*). If all entries are zero, returns zero.

void **_fmpz_vec_height**(*fmpz_t* height, const *fmpz* *vec, *slong* len)

Computes the height of (*vec*, *len*), defined as the largest of the absolute values the coefficients. Equivalently, this gives the infinity norm of the vector. If *len* is zero, the height is 0.

slong **_fmpz_vec_height_index**(const *fmpz* *vec, *slong* len)

Returns the index of an entry of maximum absolute value in the vector. The length must be at least 1.

4.3.4 Input and output

int **_fmpz_vec_fread**(FILE *file, *fmpz* **vec, *slong* *len)

Reads a vector from the stream *file* and stores it at **vec*. The format is the same as the output format of **_fmpz_vec_fprint()**, followed by either any character or the end of the file.

The interpretation of the various input arguments depends on whether or not **vec* is NULL:

If **vec* == NULL, the value of **len* on input is ignored. Once the length has been read from *file*, **len* is set to that value and a vector of this length is allocated at **vec*. Finally, **len* coefficients are read from the input stream. In case of a file or parsing error, clears the vector and sets **vec* and **len* to NULL and 0, respectively.

Otherwise, if **vec* != NULL, it is assumed that (**vec*, **len*) is a properly initialised vector. If the length on the input stream does not match **len*, a parsing error is raised. Attempts to read the right number of coefficients from the input stream. In case of a file or parsing error, leaves the vector (**vec*, **len*) in its current state.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fmpz_vec_read**(*fmpz* **vec, *slong* *len)

Reads a vector from *stdin* and stores it at **vec*.

For further details, see **_fmpz_vec_fread()**.

int **_fmpz_vec_fprint**(FILE *file, const *fmpz* *vec, *slong* len)

Prints the vector of given length to the stream *file*. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fmpz_vec_print**(const *fmpz* *vec, *slong* len)

Prints the vector of given length to *stdout*.

For further details, see **_fmpz_vec_fprint()**.

4.3.5 Conversions

void `_fmpz_vec_get_nmod_vec`(*nn_ptr* res, const *fmpz* *poly, *slong* len, *nmod_t* mod)
 Reduce the coefficients of (poly, len) modulo the given modulus and set (res, len) to the result.

void `_fmpz_vec_set_nmod_vec`(*fmpz* *res, *nn_srcptr* poly, *slong* len, *nmod_t* mod)
 Set the coefficients of (res, len) to the symmetric modulus of the coefficients of (poly, len), i.e. convert the given coefficients modulo the given modulus n to their signed integer representatives in the range $[-n/2, n/2)$.

void `_fmpz_vec_get_fft`(*ulong* **coeffs_f, const *fmpz* *coeffs_m, *slong* l, *slong* length)
 Convert the vector of coeffs coeffs_m to an fft vector coeffs_f of the given length with l limbs per coefficient with an additional limb for overflow.

void `_fmpz_vec_set_fft`(*fmpz* *coeffs_m, *slong* length, const *nn_ptr* *coeffs_f, *slong* limbs, *slong* sign)
 Convert an fft vector coeffs_f of fully reduced Fermat numbers of the given length to a vector of fmpz's. Each is assumed to be the given number of limbs in length with an additional limb for overflow. If the output coefficients are to be signed then set sign, otherwise clear it. The resulting fmpz's will be in the range $[-n, n]$ in the signed case and in the range $[0, 2n]$ in the unsigned case where $n = 2^{(\text{FLINT\_BITS} \cdot \text{limbs} - 1)}$.

slong `_fmpz_vec_get_d_vec_2exp`(double *appv, const *fmpz* *vec, *slong* len)
 Export the array of len entries starting at the pointer vec to an array of doubles appv, each entry of which is notionally multiplied by a single returned exponent to give the original entry. The returned exponent is set to be the maximum exponent of all the original entries so that all the doubles in appv have a maximum absolute value of 1.0.

4.3.6 Assignment and basic manipulation

void `_fmpz_vec_set`(*fmpz* *vec1, const *fmpz* *vec2, *slong* len2)
 Makes a copy of (vec2, len2) into vec1.

void `_fmpz_vec_swap`(*fmpz* *vec1, *fmpz* *vec2, *slong* len2)
 Swaps the integers in (vec1, len2) and (vec2, len2).

void `_fmpz_vec_zero`(*fmpz* *vec, *slong* len)
 Zeros the entries of (vec, len).

void `_fmpz_vec_neg`(*fmpz* *vec1, const *fmpz* *vec2, *slong* len2)
 Negates (vec2, len2) and places it into vec1.

void `_fmpz_vec_scalar_abs`(*fmpz* *vec1, const *fmpz* *vec2, *slong* len2)
 Takes the absolute value of entries in (vec2, len2) and places the result into vec1.

4.3.7 Comparison

int `_fmpz_vec_equal`(const *fmpz* *vec1, const *fmpz* *vec2, *slong* len)
 Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

int `_fmpz_vec_is_zero`(const *fmpz* *vec, *slong* len)
 Returns 1 if (vec, len) is zero, and 0 otherwise.

void `_fmpz_vec_max`(*fmpz* *vec1, const *fmpz* *vec2, const *fmpz* *vec3, *slong* len)
 Sets vec1 to the pointwise maximum of vec2 and vec3.

void `_fmpz_vec_max_inplace`(*fmpz* *vec1, const *fmpz* *vec2, *slong* len)
 Sets vec1 to the pointwise maximum of vec1 and vec2.

4.3.8 Sorting

`void _fmpz_vec_sort(fmpz *vec, slong len)`
Sorts the coefficients of `vec` in ascending order.

4.3.9 Addition and subtraction

`void _fmpz_vec_add(fmpz *res, const fmpz *vec1, const fmpz *vec2, slong len2)`
Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.
`void _fmpz_vec_sub(fmpz *res, const fmpz *vec1, const fmpz *vec2, slong len2)`
Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

4.3.10 Scalar multiplication and division

`void _fmpz_vec_scalar_mul_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t x)`
Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `c`, where `c` is an `fmpz_t`.
`void _fmpz_vec_scalar_mul_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)`
Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `c`, where `c` is a `slong`.
`void _fmpz_vec_scalar_mul_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)`
Sets `(vec1, len2)` to `(vec2, len2)` multiplied by `c`, where `c` is an `ulong`.
`void _fmpz_vec_scalar_mul_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)`
Sets `(vec1, len2)` to `(vec2, len2)` multiplied by 2^{exp} .
`void _fmpz_vec_scalar_divexact_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t x)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by `x`, where the division is assumed to be exact for every entry in `vec2`.
`void _fmpz_vec_scalar_divexact_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by `x`, where the division is assumed to be exact for every entry in `vec2`.
`void _fmpz_vec_scalar_divexact_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by `x`, where the division is assumed to be exact for every entry in `vec2`.
`void _fmpz_vec_scalar_fdiv_q_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t c)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, rounding down towards minus infinity whenever the division is not exact.
`void _fmpz_vec_scalar_fdiv_q_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, rounding down towards minus infinity whenever the division is not exact.
`void _fmpz_vec_scalar_fdiv_q_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by `c`, rounding down towards minus infinity whenever the division is not exact.
`void _fmpz_vec_scalar_fdiv_q_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)`
Sets `(vec1, len2)` to `(vec2, len2)` divided by 2^{exp} , rounding down towards minus infinity whenever the division is not exact.
`void _fmpz_vec_scalar_fdiv_r_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)`
Sets `(vec1, len2)` to the remainder of `(vec2, len2)` divided by 2^{exp} , rounding down the quotient towards minus infinity whenever the division is not exact.

```
void _fmpz_vec_scalar_tdiv_q_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t c)
    Sets (vec1, len2) to (vec2, len2) divided by  $c$ , rounding towards zero whenever the division is not exact.

void _fmpz_vec_scalar_tdiv_q_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
    Sets (vec1, len2) to (vec2, len2) divided by  $c$ , rounding towards zero whenever the division is not exact.

void _fmpz_vec_scalar_tdiv_q_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)
    Sets (vec1, len2) to (vec2, len2) divided by  $c$ , rounding towards zero whenever the division is not exact.

void _fmpz_vec_scalar_tdiv_q_2exp(fmpz *vec1, const fmpz *vec2, slong len2, ulong exp)
    Sets (vec1, len2) to (vec2, len2) divided by  $2^{\text{exp}}$ , rounding down towards zero whenever the division is not exact.

void _fmpz_vec_scalar_addmul_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)

void _fmpz_vec_scalar_addmul_ui(fmpz *vec1, const fmpz *vec2, slong len2, ulong c)

void _fmpz_vec_scalar_addmul_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t c)
    Adds (vec2, len2) times  $c$  to (vec1, len2).

void _fmpz_vec_scalar_addmul_si_2exp(fmpz *vec1, const fmpz *vec2, slong len2, slong c, ulong exp)
    Adds (vec2, len2) times  $c * 2^{\text{exp}}$  to (vec1, len2), where  $c$  is a slong.

void _fmpz_vec_scalar_submul_fmpz(fmpz *vec1, const fmpz *vec2, slong len2, const fmpz_t x)
    Subtracts (vec2, len2) times  $c$  from (vec1, len2), where  $c$  is a fmpz_t.

void _fmpz_vec_scalar_submul_si(fmpz *vec1, const fmpz *vec2, slong len2, slong c)
    Subtracts (vec2, len2) times  $c$  from (vec1, len2), where  $c$  is a slong.

void _fmpz_vec_scalar_submul_si_2exp(fmpz *vec1, const fmpz *vec2, slong len2, slong c, ulong e)
    Subtracts (vec2, len2) times  $c \times 2^e$  from (vec1, len2), where  $c$  is a slong.
```

4.3.11 Sums and products

```
void _fmpz_vec_sum(fmpz_t res, const fmpz *vec, slong len)
    Sets  $\text{res}$  to the sum of the entries in (vec, len). Aliasing of  $\text{res}$  with the entries in  $\text{vec}$  is not permitted.

void _fmpz_ui_vec_prod(fmpz_t res, nn_srcptr vec, slong len)

void _fmpz_vec_prod(fmpz_t res, const fmpz *vec, slong len)
    Sets  $\text{res}$  to the product of the entries in (vec, len). Aliasing of  $\text{res}$  with the entries in  $\text{vec}$  is not permitted. Uses binary splitting.
```

4.3.12 Reduction mod p

```
void _fmpz_vec_scalar_mod_fmpz(fmpz *res, const fmpz *vec, slong len, const fmpz_t p)
    Reduces all entries in (vec, len) modulo  $p > 0$ .

void _fmpz_vec_scalar_smod_fmpz(fmpz *res, const fmpz *vec, slong len, const fmpz_t p)
    Reduces all entries in (vec, len) modulo  $p > 0$ , choosing the unique representative in  $(-p/2, p/2]$ .
```


4.3.13 Gaussian content

void **_fmpz_vec_content**(*fmpz_t* res, const *fmpz* *vec, *slong* len)

Sets **res** to the non-negative content of the entries in **vec**. The content of a zero vector, including the case when the length is zero, is defined to be zero.

void **_fmpz_vec_content_chained**(*fmpz_t* res, const *fmpz* *vec, *slong* len, const *fmpz_t* input)

Sets **res** to the non-negative content of **input** and the entries in **vec**. This is useful for calculating the common content of several vectors.

void **_fmpz_vec_lcm**(*fmpz_t* res, const *fmpz* *vec, *slong* len)

Sets **res** to the nonnegative least common multiple of the entries in **vec**. The least common multiple is zero if any entry in the vector is zero. The least common multiple of a length zero vector is defined to be one.

4.3.14 Dot product

void **_fmpz_vec_dot_general_naive**(*fmpz_t* res, const *fmpz_t* initial, int subtract, const *fmpz* *a, const *fmpz* *b, int reverse, *slong* len)

void **_fmpz_vec_dot_general**(*fmpz_t* res, const *fmpz_t* initial, int subtract, const *fmpz* *a, const *fmpz* *b, int reverse, *slong* len)

Computes the dot product of the vectors *a* and *b*, setting *res* to $s + (-1)^{\text{subtract}} \sum_{i=0}^{\text{len}-1} a_i b_i$. The initial term *s* is optional and can be omitted by passing *NULL* (equivalently, $s = 0$). The parameter *subtract* must be 0 or 1. If the *reverse* flag is 1, the second vector is reversed.

Aliasing is allowed between **res** and **initial** but not between **res** and the entries of **a** and **b**.

The *naive* version is used for testing purposes.

void **_fmpz_vec_dot**(*fmpz_t* res, const *fmpz* *vec1, const *fmpz* *vec2, *slong* len2)

Sets **res** to the dot product of (**vec1**, **len2**) and (**vec2**, **len2**).

4.4 fmpz_factor.h – integer factorisation

4.4.1 Types, macros and constants

type **fmpz_factor_struct**

type **fmpz_factor_t**

4.4.2 Factoring integers

An integer may be represented in factored form using the **fmpz_factor_t** data structure. This consists of two **fmpz** vectors representing bases and exponents, respectively. Canonically, the bases will be prime numbers sorted in ascending order and the exponents will be positive. A separate **int** field holds the sign, which may be -1 , 0 or 1 .

void **fmpz_factor_init**(*fmpz_factor_t* factor)

Initialises an **fmpz_factor_t** structure.

void **fmpz_factor_clear**(*fmpz_factor_t* factor)

Clears an **fmpz_factor_t** structure.

void **_fmpz_factor_append_ui**(*fmpz_factor_t* factor, *ulong* p, *ulong* exp)

Append a factor *p* to the given exponent to the **fmpz_factor_t** structure **factor**.

void **_fmpz_factor_append**(*fmpz_factor_t* factor, const *fmpz_t* p, *ulong* exp)

Append a factor p to the given exponent to the *fmpz_factor_t* structure *factor*.

void **fmpz_factor**(*fmpz_factor_t* factor, const *fmpz_t* n)

Factors n into prime numbers. If n is zero or negative, the sign field of the *factor* object will be set accordingly.

int **fmpz_factor_smooth**(*fmpz_factor_t* factor, const *fmpz_t* n, *ulong* bits, int proved)

Factors n into prime numbers up to approximately the given number of bits and possibly one additional cofactor, which may or may not be prime.

If the number is definitely factored fully, the return value is 1, otherwise the final factor (which may have exponent greater than 1) is composite and needs to be factored further.

If the number has a factor of around the given number of bits, there is at least a two-thirds chance of finding it. Smaller factors should be found with a much higher probability.

The amount of time spent factoring can be controlled by lowering or increasing *bits*. However, the quadratic sieve may be faster if *bits* is set to more than one third of the number of bits of n .

The function uses trial factoring up to *bits* = 15, followed by a primality test and a perfect power test to check if the factorisation is complete. If *bits* is at least 16, it proceeds to use the elliptic curve method to look for larger factors.

The behavior of primality testing is determined by the *proved* parameter:

If *proved* is set to 1 the function will prove all factors prime (other than the last factor, if the return value is 0).

If *proved* is set to 0, the function will only check that factors are probable primes.

If *proved* is set to -1, the function will not test primality after performing trial division. A perfect power test is still performed.

As an exception to the rules stated above, this function will call *n_factor* internally if n or the remainder after trial division is smaller than one word, guaranteeing a complete factorisation.

void **fmpz_factor_si**(*fmpz_factor_t* factor, *ulong* n)

Like *fmpz_factor*, but takes a machine integer n as input.

int **fmpz_factor_trial_range**(*fmpz_factor_t* factor, const *fmpz_t* n, *ulong* start, *ulong* num_primes)

Factors n into prime factors using trial division. If n is zero or negative, the sign field of the *factor* object will be set accordingly.

The algorithm starts with the given start index in the *flint_primes* table and uses at most *num_primes* primes from that point.

The function returns 1 if n is completely factored, otherwise it returns 0.

int **fmpz_factor_trial**(*fmpz_factor_t* factor, const *fmpz_t* n, *ulong* num_primes)

Factors n into prime factors using trial division. If n is zero or negative, the sign field of the *factor* object will be set accordingly.

The algorithm uses the given number of primes, which must be in the range [0, 3512]. An exception is raised if a number outside this range is passed.

The function returns 1 if n is completely factored, otherwise it returns 0.

The final entry in the factor struct is set to the cofactor after removing prime factors, if this is not 1.

void **fmpz_factor_refine**(*fmpz_factor_t* res, const *fmpz_factor_t* f)

Attempts to improve a partial factorization of an integer by “refining” the factorization *f* to a more complete factorization *res* whose bases are pairwise relatively prime.

This function does not require its input to be in canonical form, nor does it guarantee that the resulting factorization will be canonical.

void **mpz_factor_expand_iterative**(*mpz_t* n, const *mpz_factor_t* factor)

Evaluates an integer in factored form back to an *mpz_t*.

This currently exponentiates the bases separately and multiplies them together one by one, although much more efficient algorithms exist.

int **mpz_factor_pp1**(*mpz_t* factor, const *mpz_t* n, *ulong* B1, *ulong* B2_sqrt, *ulong* c)

Use Williams' $p + 1$ method to factor n , using a prime bound in stage 1 of B1 and a prime limit in stage 2 of at least the square of B2_sqrt. If a factor is found, the function returns 1 and **factor** is set to the factor that is found. Otherwise, the function returns 0.

The value c should be a random value greater than 2. Successive calls to the function with different values of c give additional chances to factor n with roughly exponentially decaying probability of finding a factor which has been missed (if $p + 1$ or $p - 1$ is not smooth for any prime factors p of n then the function will not ever succeed).

int **mpz_factor_pollard_brent_single**(*mpz_t* p_factor, *mpz_t* n_in, *mpz_t* yi, *mpz_t* ai, *ulong* max_iters)

Pollard Rho algorithm for integer factorization. Assumes that the n is not prime. **factor** is set as the factor if found. Takes as input the initial value y , to start polynomial evaluation, and a , the constant of the polynomial used. It is not assured that the factor found will be prime. Does not compute the complete factorization, just one factor. Returns the number of limbs of factor if factorization is successful (non trivial factor is found), else returns 0.

max_iters is the number of iterations tried in process of finding the cycle. If the algorithm fails to find a non trivial factor in one call, it tries again (this time with a different set of random values).

int **mpz_factor_pollard_brent**(*mpz_t* factor, *flint_rand_t* state, *mpz_t* n, *ulong* max_tries, *ulong* max_iters)

Pollard Rho algorithm for integer factorization. Assumes that the n is not prime. **factor** is set as the factor if found. It is not assured that the factor found will be prime. Does not compute the complete factorization, just one factor. Returns the number of limbs of factor if factorization is successful (non trivial factor is found), else returns 0.

max_iters is the number of iterations tried in process of finding the cycle. If the algorithm fails to find a non trivial factor in one call, it tries again (this time with a different set of random values). This process is repeated a maximum of **max_tries** times.

The algorithm used is a modification of the original Pollard Rho algorithm, suggested by Richard Brent. It can be found in the paper available at <https://maths-people.anu.edu.au/~brent/pd/rpb051i.pdf>

4.4.3 Input and output

int **mpz_factor_fprint**(FILE *fs, const *mpz_factor_t* factor)

int **mpz_factor_print**(const *mpz_factor_t* factor)

Prints the factorization **factor** into **fs** or **stdout**. If **factor** is zero, it prints 0. Else, it prints the factorization as $f_{\{1\}}^{e_{\{1\}}} * \dots * f_{\{n\}}^{e_{\{n\}}}$, where $f_{\{i\}}$ and $e_{\{i\}}$ are the i -th factor and exponent, where $e_{\{i\}}$ is omitted if $e_i = 1$. In particular, if **factor** is 1 or -1 , it prints 1 or -1 , respectively.

Returns the number of characters written to file stream.

4.4.4 Elliptic curve (ECM) method

Factoring of `mpz` integers using ECM

void **mpz_factor_ecm_init**(*ecm_t* ecm_inf, *ulong* sz)

Initializes the *ecm_t* struct. This is needed in some functions and carries data between subsequent calls.

void **mpz_factor_ecm_clear**(*ecm_t* ecm_inf)

Clears the *ecm_t* struct.

void **mpz_factor_ecm_double**(*nn_ptr* x, *nn_ptr* z, *nn_ptr* x0, *nn_ptr* z0, *nn_ptr* n, *ecm_t* ecm_inf)

Sets the point $(x : z)$ to two times $(x_0 : z_0)$ modulo n according to the formula

$$x = (x_0 + z_0)^2 \cdot (x_0 - z_0)^2 \pmod{n},$$

$$z = 4x_0z_0((x_0 - z_0)^2 + 4a_{24}x_0z_0) \pmod{n}.$$

ecm_inf is used just to use temporary *nn_ptr*'s in the structure. This group doubling is valid only for points expressed in Montgomery projective coordinates.

void **mpz_factor_ecm_add**(*nn_ptr* x, *nn_ptr* z, *nn_ptr* x1, *nn_ptr* z1, *nn_ptr* x2, *nn_ptr* z2, *nn_ptr* x0, *nn_ptr* z0, *nn_ptr* n, *ecm_t* ecm_inf)

Sets the point $(x : z)$ to the sum of $(x_1 : z_1)$ and $(x_2 : z_2)$ modulo n , given the difference $(x_0 : z_0)$ according to the formula

$$x = 4z_0(x_1x_2 - z_1z_2)^2 \pmod{n},$$

$$z = 4x_0(x_2z_1 - x_1z_2)^2 \pmod{n}.$$

ecm_inf is used just to use temporary *nn_ptr*'s in the structure. This group addition is valid only for points expressed in Montgomery projective coordinates.

void **mpz_factor_ecm_mul_montgomery_ladder**(*nn_ptr* x, *nn_ptr* z, *nn_ptr* x0, *nn_ptr* z0, *ulong* k, *nn_ptr* n, *ecm_t* ecm_inf)

Montgomery ladder algorithm for scalar multiplication of elliptic points.

Sets the point $(x : z)$ to $k(x_0 : z_0)$ modulo n .

ecm_inf is used just to use temporary *nn_ptr*'s in the structure. Valid only for points expressed in Montgomery projective coordinates.

int **mpz_factor_ecm_select_curve**(*nn_ptr* f, *nn_ptr* sigma, *nn_ptr* n, *ecm_t* ecm_inf)

Selects a random elliptic curve given a random integer *sigma*, according to Suyama's parameterization. If the factor is found while selecting the curve, the number of limbs required to store the factor is returned, otherwise 0.

It could be possible that the selected curve is unsuitable for further computations, in such a case, -1 is returned.

Also selects the initial point x_0 , and the value of $(a + 2)/4$, where a is a curve parameter. Sets z_0 as 1. All these are stored in the *ecm_t* struct.

The curve selected is of Montgomery form, the points selected satisfy the curve and are projective coordinates.

int **mpz_factor_ecm_stage_I**(*nn_ptr* f, const *ulong* *prime_array, *ulong* num, *ulong* B1, *nn_ptr* n, *ecm_t* ecm_inf)

Stage I implementation of the ECM algorithm.

f is set as the factor if found. *num* is number of prime numbers $\leq$ the bound *B1*. *prime_array* is an array of first *B1* primes. *n* is the number being factored.

If the factor is found, number of words required to store the factor is returned, otherwise 0.

int **fmpz_factor_ecm_stage_II**(*nn_ptr* f, *ulong* B1, *ulong* B2, *ulong* P, *nn_ptr* n, *ecm_t* ecm_inf)
Stage II implementation of the ECM algorithm.

f is set as the factor if found. **B1**, **B2** are the two bounds. **P** is the primorial (approximately equal to $\sqrt{B2}$). **n** is the number being factored.

If the factor is found, number of words required to store the factor is returned, otherwise 0.

int **fmpz_factor_ecm**(*fmpz_t* f, *ulong* curves, *ulong* B1, *ulong* B2, *flint_rand_t* state, const *fmpz_t* n_in)

Outer wrapper function for the ECM algorithm. In case **f** can fit in a single unsigned word, a call to **n_factor_ecm** is made.

The function calls stage I and II, and the precomputations (builds **prime_array** for stage I, **GCD_table** and **prime_table** for stage II).

f is set as the factor if found. **curves** is the number of random curves being tried. **B1**, **B2** are the two bounds or stage I and stage II. **n** is the number being factored.

If a factor is found in stage I, 1 is returned. If a factor is found in stage II, 2 is returned. If a factor is found while selecting the curve, -1 is returned. Otherwise 0 is returned.

4.5 fmpz_mat.h – matrices over the integers

The *fmpz_mat_t* data type represents dense matrices of multiprecision integers, implemented using *fmpz* vectors.

No automatic resizing is performed: in general, the user must provide matrices of correct dimensions for both input and output variables. Output variables are *not* allowed to be aliased with input variables unless otherwise noted.

Matrices are indexed from zero: an $m \times n$ matrix has rows of index $0, 1, \dots, m-1$ and columns of index $0, 1, \dots, n-1$. One or both of m and n may be zero.

Elements of a matrix can be read or written using the **fmpz_mat_entry** macro, which returns a reference to the entry at a given row and column index. This reference can be passed as an input or output *fmpz_t* variable to any function in the **fmpz** module for direct manipulation.

4.5.1 Simple example

The following example creates the 2×2 matrix A with value $2i + j$ at row i and column j , computes $B = A^2$, and prints both matrices.

```
#include "fmpz.h"
#include "fmpz_mat.h"

int main()
{
    long i, j;
    fmpz_mat_t A;
    fmpz_mat_t B;
    fmpz_mat_init(A, 2, 2);
    fmpz_mat_init(B, 2, 2);
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            fmpz_set_ui(fmpz_mat_entry(A, i, j), 2*i+j);
    fmpz_mat_mul(B, A, A);
    flint_printf("A = \n");
    fmpz_mat_print_pretty(A);
```

(continues on next page)

(continued from previous page)

```

    flint_printf("A^2 = \n");
    fmpz_mat_print_pretty(B);
    fmpz_mat_clear(A);
    fmpz_mat_clear(B);
}

```

The output is:

```

A =
[[0 1]
 [2 3]]
A^2 =
[[2 3]
 [6 11]]

```

4.5.2 Types, macros and constants

type `fmpz_mat_struct`

type `fmpz_mat_t`

4.5.3 Memory management

void `fmpz_mat_init`(*fmpz_mat_t* mat, *slong* rows, *slong* cols)

Initialises a matrix with the given number of rows and columns for use.

void `fmpz_mat_clear`(*fmpz_mat_t* mat)

Clears the given matrix.

4.5.4 Basic assignment and manipulation

void `fmpz_mat_set`(*fmpz_mat_t* mat1, const *fmpz_mat_t* mat2)

Sets `mat1` to a copy of `mat2`. The dimensions of `mat1` and `mat2` must be the same.

void `fmpz_mat_init_set`(*fmpz_mat_t* mat, const *fmpz_mat_t* src)

Initialises the matrix `mat` to the same size as `src` and sets it to a copy of `src`.

slong `fmpz_mat_nrows`(const *fmpz_mat_t* mat)

slong `fmpz_mat_ncols`(const *fmpz_mat_t* mat)

Returns respectively the number of rows and columns of the matrix.

void `fmpz_mat_swap`(*fmpz_mat_t* mat1, *fmpz_mat_t* mat2)

Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

void `fmpz_mat_swap_entrywise`(*fmpz_mat_t* mat1, *fmpz_mat_t* mat2)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

fmpz *`fmpz_mat_entry`(const *fmpz_mat_t* mat, *slong* i, *slong* j)

Returns a reference to the entry of `mat` at row *i* and column *j*. This reference can be passed as an input or output variable to any function in the `fmpz` module for direct manipulation.

Both *i* and *j* must not exceed the dimensions of the matrix.

This function is implemented as a macro.

void **fmpz_mat_zero**(*fmpz_mat_t* mat)

Sets all entries of **mat** to 0.

void **fmpz_mat_one**(*fmpz_mat_t* mat)

Sets **mat** to the unit matrix, having ones on the main diagonal and zeroes elsewhere. If **mat** is nonsquare, it is set to the truncation of a unit matrix.

void **fmpz_mat_swap_rows**(*fmpz_mat_t* mat, *slong* *perm, *slong* r, *slong* s)

Swaps rows **r** and **s** of **mat**. If **perm** is non-NULL, the permutation of the rows will also be applied to **perm**.

void **fmpz_mat_swap_cols**(*fmpz_mat_t* mat, *slong* *perm, *slong* r, *slong* s)

Swaps columns **r** and **s** of **mat**. If **perm** is non-NULL, the permutation of the columns will also be applied to **perm**.

void **fmpz_mat_invert_rows**(*fmpz_mat_t* mat, *slong* *perm)

Swaps rows **i** and **r - i** of **mat** for $0 \leq i < r/2$, where **r** is the number of rows of **mat**. If **perm** is non-NULL, the permutation of the rows will also be applied to **perm**.

void **fmpz_mat_invert_cols**(*fmpz_mat_t* mat, *slong* *perm)

Swaps columns **i** and **c - i** of **mat** for $0 \leq i < c/2$, where **c** is the number of columns of **mat**. If **perm** is non-NULL, the permutation of the columns will also be applied to **perm**.

4.5.5 Window

void **fmpz_mat_window_init**(*fmpz_mat_t* window, const *fmpz_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2)

Initializes the matrix **window** to be an **r2 - r1** by **c2 - c1** submatrix of **mat** whose (0,0) entry is the (**r1**, **c1**) entry of **mat**. The memory for the elements of **window** is shared with **mat**.

void **fmpz_mat_window_clear**(*fmpz_mat_t* window)

Clears the matrix **window** and releases any memory that it uses. Note that the memory to the underlying matrix that **window** points to is not freed.

4.5.6 Random matrix generation

void **fmpz_mat_randbits**(*fmpz_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits)

Sets the entries of **mat** to random signed integers whose absolute values have the given number of binary bits.

void **fmpz_mat_randtest**(*fmpz_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits)

Sets the entries of **mat** to random signed integers whose absolute values have a random number of bits up to the given number of bits inclusive.

void **fmpz_mat_randintrel**(*fmpz_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits)

Sets **mat** to be a random *integer relations* matrix, with signed entries up to the given number of bits.

The number of columns of **mat** must be equal to one more than the number of rows. The format of the matrix is a set of random integers in the left hand column and an identity matrix in the remaining square submatrix.

void **fmpz_mat_randsimdioph**(*fmpz_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits, *flint_bitcnt_t* bits2)

Sets **mat** to a random *simultaneous diophantine* matrix.

The matrix must be square. The top left entry is set to 2^{bits2} . The remainder of that row is then set to signed random integers of the given number of binary bits. The remainder of the first

column is zero. Running down the rest of the diagonal are the values 2^{bits} with all remaining entries zero.

void **fmpz_mat_randntrulike**(*fmpz_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits, *ulong* q)

Sets a square matrix **mat** of even dimension to a random *NTRU like* matrix.

The matrix is broken into four square submatrices. The top left submatrix is set to the identity. The bottom left submatrix is set to the zero matrix. The bottom right submatrix is set to q times the identity matrix. Finally the top right submatrix has the following format. A random vector h of length $r/2$ is created, with random signed entries of the given number of bits. Then entry (i, j) of the submatrix is set to $h[i + j \bmod r/2]$.

void **fmpz_mat_randntrulike2**(*fmpz_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits, *ulong* q)

Sets a square matrix **mat** of even dimension to a random *NTRU like* matrix.

The matrix is broken into four square submatrices. The top left submatrix is set to q times the identity matrix. The top right submatrix is set to the zero matrix. The bottom right submatrix is set to the identity matrix. Finally the bottom left submatrix has the following format. A random vector h of length $r/2$ is created, with random signed entries of the given number of bits. Then entry (i, j) of the submatrix is set to $h[i + j \bmod r/2]$.

void **fmpz_mat_randajtai**(*fmpz_mat_t* mat, *flint_rand_t* state, double alpha)

Sets a square matrix **mat** to a random *ajtai* matrix. The diagonal entries (i, i) are set to a random entry in the range $[1, 2^{b-1}]$ inclusive where $b = \lfloor (2r - i)^\alpha \rfloor$ for some double parameter α . The entries below the diagonal in column i are set to a random entry in the range $(-2^b + 1, 2^b - 1)$ whilst the entries to the right of the diagonal in row i are set to zero.

int **fmpz_mat_randpermdiag**(*fmpz_mat_t* mat, *flint_rand_t* state, const *fmpz* *diag, *slong* n)

Sets **mat** to a random permutation of the rows and columns of a given diagonal matrix. The diagonal matrix is specified in the form of an array of the n initial entries on the main diagonal.

The return value is 0 or 1 depending on whether the permutation is even or odd.

void **fmpz_mat_randrank**(*fmpz_mat_t* mat, *flint_rand_t* state, *slong* rank, *flint_bitcnt_t* bits)

Sets **mat** to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the nonzero elements being random integers of the given bit size.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling *fmpz_mat_randops()*.

void **fmpz_mat_randedet**(*fmpz_mat_t* mat, *flint_rand_t* state, const *fmpz_t* det)

Sets **mat** to a random sparse matrix with minimal number of nonzero entries such that its determinant has the given value.

Note that the matrix will be zero if **det** is zero. In order to generate a non-zero singular matrix, the function *fmpz_mat_randrank()* can be used.

The matrix can be transformed into a dense matrix with unchanged determinant by subsequently calling *fmpz_mat_randops()*.

void **fmpz_mat_randops**(*fmpz_mat_t* mat, *flint_rand_t* state, *slong* count)

Randomises **mat** by performing elementary row or column operations. More precisely, at most **count** random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, the determinant) unchanged.

4.5.7 Input and output

int **mpz_mat_fprint**(FILE *file, const *mpz_mat_t* mat)

Prints the given matrix to the stream **file**. The format is the number of rows, a space, the number of columns, two spaces, then a space separated list of coefficients, one row after the other.

In case of success, returns a positive value; otherwise, returns a non-positive value.

int **mpz_mat_fprint_pretty**(FILE *file, const *mpz_mat_t* mat)

Prints the given matrix to the stream **file**. The format is an opening square bracket, then on each line a row of the matrix, followed by a closing square bracket. Each row is written as an opening square bracket followed by a space separated list of coefficients followed by a closing square bracket.

In case of success, returns a positive value; otherwise, returns a non-positive value.

int **mpz_mat_print**(const *mpz_mat_t* mat)

Prints the given matrix to the stream **stdout**. For further details, see *mpz_mat_fprint()*.

int **mpz_mat_print_pretty**(const *mpz_mat_t* mat)

Prints the given matrix to **stdout**. For further details, see *mpz_mat_fprint_pretty()*.

int **mpz_mat_fread**(FILE *file, *mpz_mat_t* mat)

Reads a matrix from the stream **file**, storing the result in **mat**. The expected format is the number of rows, a space, the number of columns, two spaces, then a space separated list of coefficients, one row after the other.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

int **mpz_mat_read**(*mpz_mat_t* mat)

Reads a matrix from **stdin**, storing the result in **mat**.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

4.5.8 Comparison

int **mpz_mat_equal**(const *mpz_mat_t* mat1, const *mpz_mat_t* mat2)

Returns a non-zero value if **mat1** and **mat2** have the same dimensions and entries, and zero otherwise.

int **mpz_mat_is_zero**(const *mpz_mat_t* mat)

Returns a non-zero value if all entries **mat** are zero, and otherwise returns zero.

int **mpz_mat_is_one**(const *mpz_mat_t* mat)

Returns a non-zero value if **mat** is the unit matrix or the truncation of a unit matrix, and otherwise returns zero.

int **mpz_mat_is_empty**(const *mpz_mat_t* mat)

Returns a non-zero value if the number of rows or the number of columns in **mat** is zero, and otherwise returns zero.

int **mpz_mat_is_square**(const *mpz_mat_t* mat)

Returns a non-zero value if the number of rows is equal to the number of columns in **mat**, and otherwise returns zero.

int **mpz_mat_is_zero_row**(const *mpz_mat_t* mat, *slong* i)

Returns a non-zero value if row *i* of **mat** is zero.

int **mpz_mat_equal_col**(*mpz_mat_t* M, *slong* m, *slong* n)

Returns 1 if columns *m* and *n* of the matrix *M* are equal, otherwise returns 0.

int **mpz_mat_equal_row**(*mpz_mat_t* M, *slong* m, *slong* n)

Returns 1 if rows *m* and *n* of the matrix *M* are equal, otherwise returns 0.

4.5.9 Transpose

void **fmpz_mat_transpose**(*fmpz_mat_t* B, const *fmpz_mat_t* A)

Sets *B* to A^T , the transpose of *A*. Dimensions must be compatible. Aliasing is allowed for square matrices.

4.5.10 Concatenate

void **fmpz_mat_concat_vertical**(*fmpz_mat_t* res, const *fmpz_mat_t* mat1, const *fmpz_mat_t* mat2)

Sets *res* to vertical concatenation of (*mat1*, *mat2*) in that order. Matrix dimensions: *mat1*: $m \times n$, *mat2*: $k \times n$, *res*: $(m + k) \times n$.

void **fmpz_mat_concat_horizontal**(*fmpz_mat_t* res, const *fmpz_mat_t* mat1, const *fmpz_mat_t* mat2)

Sets *res* to horizontal concatenation of (*mat1*, *mat2*) in that order. Matrix dimensions: *mat1*: $m \times n$, *mat2*: $m \times k$, *res*: $m \times (n + k)$.

4.5.11 Modular reduction and reconstruction

void **fmpz_mat_get_nmod_mat**(*nmod_mat_t* Amod, const *fmpz_mat_t* A)

Sets the entries of *Amod* to the entries of *A* reduced by the modulus of *Amod*.

void **fmpz_mat_set_nmod_mat**(*fmpz_mat_t* A, const *nmod_mat_t* Amod)

Sets the entries of *Amod* to the residues in *Amod*, normalised to the interval $-m/2 \leq r < m/2$ where *m* is the modulus.

void **fmpz_mat_set_nmod_mat_unsigned**(*fmpz_mat_t* A, const *nmod_mat_t* Amod)

Sets the entries of *Amod* to the residues in *Amod*, normalised to the interval $0 \leq r < m$ where *m* is the modulus.

void **fmpz_mat_CRT_ui**(*fmpz_mat_t* res, const *fmpz_mat_t* mat1, const *fmpz_t* m1, const *nmod_mat_t* mat2, int sign)

Given *mat1* with entries modulo *m* and *mat2* with modulus *n*, sets *res* to the CRT reconstruction modulo mn with entries satisfying $-mn/2 \leq c < mn/2$ (if *sign* = 1) or $0 \leq c < mn$ (if *sign* = 0).

void **fmpz_mat_multi_mod_ui_precomp**(*nmod_mat_t* *residues, *slong* nres, const *fmpz_mat_t* mat, const *fmpz_comb_t* comb, *fmpz_comb_temp_t* temp)

Sets each of the *nres* matrices in *residues* to *mat* reduced modulo the modulus of the respective matrix, given precomputed *comb* and *comb_temp* structures.

Note: *fmpz.h* must be included **before** *fmpz_mat.h* in order for this function to be declared.

void **fmpz_mat_multi_mod_ui**(*nmod_mat_t* *residues, *slong* nres, const *fmpz_mat_t* mat)

Sets each of the *nres* matrices in *residues* to *mat* reduced modulo the modulus of the respective matrix.

This function is provided for convenience purposes. For reducing or reconstructing multiple integer matrices over the same set of moduli, it is faster to use *fmpz_mat_multi_mod_precomp*.

void **fmpz_mat_multi_CRT_ui_precomp**(*fmpz_mat_t* mat, *nmod_mat_t* *const residues, *slong* nres, const *fmpz_comb_t* comb, *fmpz_comb_temp_t* temp, int sign)

Reconstructs *mat* from its images modulo the *nres* matrices in *residues*, given precomputed *comb* and *comb_temp* structures.

Note: *fmpz.h* must be included **before** *fmpz_mat.h* in order for this function to be declared.

void **fmprz_mat_multi_CRT_ui**(*fmprz_mat_t* mat, *nmod_mat_t* *const residues, *slong* nres, int sign)
Reconstructs *mat* from its images modulo the *nres* matrices in *residues*.

This function is provided for convenience purposes. For reducing or reconstructing multiple integer matrices over the same set of moduli, it is faster to use *fmprz_mat_multi_CRT_ui_precomp()*.

4.5.12 Addition and subtraction

void **fmprz_mat_add**(*fmprz_mat_t* C, const *fmprz_mat_t* A, const *fmprz_mat_t* B)
Sets *C* to the elementwise sum $A + B$. All inputs must be of the same size. Aliasing is allowed.

void **fmprz_mat_sub**(*fmprz_mat_t* C, const *fmprz_mat_t* A, const *fmprz_mat_t* B)
Sets *C* to the elementwise difference $A - B$. All inputs must be of the same size. Aliasing is allowed.

void **fmprz_mat_neg**(*fmprz_mat_t* B, const *fmprz_mat_t* A)
Sets *B* to the elementwise negation of *A*. Both inputs must be of the same size. Aliasing is allowed.

4.5.13 Matrix-scalar arithmetic

void **fmprz_mat_scalar_mul_si**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *slong* c)
void **fmprz_mat_scalar_mul_ui**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *ulong* c)
void **fmprz_mat_scalar_mul_fmpz**(*fmprz_mat_t* B, const *fmprz_mat_t* A, const *fmpz_t* c)
Set $B = A * c$ where *A* is an *fmprz_mat_t* and *c* is a scalar respectively of type *slong*, *ulong*, or *fmpz_t*. The dimensions of *A* and *B* must be compatible.

void **fmprz_mat_scalar_addmul_si**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *slong* c)
void **fmprz_mat_scalar_addmul_ui**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *ulong* c)
void **fmprz_mat_scalar_addmul_fmpz**(*fmprz_mat_t* B, const *fmprz_mat_t* A, const *fmpz_t* c)
Set $B = B + A * c$ where *A* is an *fmprz_mat_t* and *c* is a scalar respectively of type *slong*, *ulong*, or *fmpz_t*. The dimensions of *A* and *B* must be compatible.

void **fmprz_mat_scalar_submul_si**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *slong* c)
void **fmprz_mat_scalar_submul_ui**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *ulong* c)
void **fmprz_mat_scalar_submul_fmpz**(*fmprz_mat_t* B, const *fmprz_mat_t* A, const *fmpz_t* c)
Set $B = B - A * c$ where *A* is an *fmprz_mat_t* and *c* is a scalar respectively of type *slong*, *ulong*, or *fmpz_t*. The dimensions of *A* and *B* must be compatible.

void **fmprz_mat_scalar_addmul_nmod_mat_ui**(*fmprz_mat_t* B, const *nmod_mat_t* A, *ulong* c)
void **fmprz_mat_scalar_addmul_nmod_mat_fmpz**(*fmprz_mat_t* B, const *nmod_mat_t* A, const *fmpz_t* c)
Set $B = B + A * c$ where *A* is an *nmod_mat_t* and *c* is a scalar respectively of type *ulong* or *fmpz_t*. The dimensions of *A* and *B* must be compatible.

void **fmprz_mat_scalar_divexact_si**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *slong* c)
void **fmprz_mat_scalar_divexact_ui**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *ulong* c)
void **fmprz_mat_scalar_divexact_fmpz**(*fmprz_mat_t* B, const *fmprz_mat_t* A, const *fmpz_t* c)
Set $A = B / c$, where *B* is an *fmprz_mat_t* and *c* is a scalar respectively of type *slong*, *ulong*, or *fmpz_t*, which is assumed to divide all elements of *B* exactly.

void **fmprz_mat_scalar_mul_2exp**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *ulong* exp)
Set the matrix *B* to the matrix *A*, of the same dimensions, multiplied by 2^{exp} .

void **fmprz_mat_scalar_tdiv_q_2exp**(*fmprz_mat_t* B, const *fmprz_mat_t* A, *ulong* exp)
Set the matrix *B* to the matrix *A*, of the same dimensions, divided by 2^{exp} , rounding down towards zero.

void **fmpz_mat_scalar_smod**(*fmpz_mat_t* B, const *fmpz_mat_t* A, const *fmpz_t* P)

Set the matrix B to the matrix A, of the same dimensions, with each entry reduced modulo P in the symmetric moduli system. We require $P > 0$.

4.5.14 Matrix multiplication

void **fmpz_mat_mul**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Sets C to the matrix product $C = AB$. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

This function automatically switches between classical and multimodular multiplication, based on a heuristic comparison of the dimensions and entry sizes.

void **fmpz_mat_mul_classical**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Sets C to the matrix product $C = AB$ computed using classical matrix algorithm.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

void **fmpz_mat_mul_waksman**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Sets C to the matrix product $C = AB$ computed using Waksman multiplication, which does only $n^3/2 + O(n^2)$ products, but many additions. This is good for small matrices with large entries.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

void **fmpz_mat_mul_strassen**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B. Uses Strassen multiplication (the Strassen-Winograd variant).

void **_fmpz_mat_mul_multi_mod**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B, int sign, *flint_bitcnt_t* bits)

void **fmpz_mat_mul_multi_mod**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Sets C to the matrix product $C = AB$ computed using a multimodular algorithm. C is computed modulo several small prime numbers and reconstructed using the Chinese Remainder Theorem. This generally becomes more efficient than classical multiplication for large matrices.

The absolute value of the elements of C should be $< 2^{\text{bits}}$, and **sign** should be 0 if the entries of C are known to be nonnegative and 1 otherwise. The function `fmpz_mat_mul_multi_mod()` calculates a rigorous bound automatically. If the default bound is too pessimistic, `_fmpz_mat_mul_multi_mod()` can be used with a custom bound.

The matrices must have compatible dimensions for matrix multiplication. No aliasing is allowed.

int **fmpz_mat_mul_blas**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Tries to set $C = AB$ using BLAS and returns 1 for success and 0 for failure. Dimensions must be compatible for matrix multiplication. No aliasing is allowed. This function currently will fail if the matrices are empty, their dimensions are too large, or their max bits size is over one million bits.

void **fmpz_mat_mul_fft**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Aliasing is allowed.

void **fmpz_mat_sqr**(*fmpz_mat_t* B, const *fmpz_mat_t* A)

Sets B to the square of the matrix A, which must be a square matrix. Aliasing is allowed. The function calls `fmpz_mat_mul()` for dimensions less than 12 and calls `fmpz_mat_sqr_bodrato()` for cases in which the latter is faster.

void **fmpz_mat_sqr_bodrato**(*fmpz_mat_t* B, const *fmpz_mat_t* A)

Sets B to the square of the matrix A, which must be a square matrix. Aliasing is allowed. The Bodrato algorithm is described in [Bodrato2010]. It is highly efficient for squaring matrices which satisfy both the following conditions: (a) large elements, (b) dimensions less than 150.

void **fmpz_mat_pow**(*fmpz_mat_t* B, const *fmpz_mat_t* A, *ulong* e)

Sets B to the matrix A raised to the power e, where A must be a square matrix. Aliasing is allowed.

void **_fmpz_mat_mul_small**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

This internal function sets C to the matrix product $C = AB$ computed using classical matrix algorithm assuming that all entries of A and B are small, that is, have bits $\leq FLINT\_BITS - 2$. No aliasing is allowed.

void **_fmpz_mat_mul_double_word**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

This function is only for internal use and assumes that either:

- the entries of A and B are all nonnegative and strictly less than $2^{2*FLINT\_BITS}$, or
- the entries of A and B are all strictly less than $2^{2*FLINT\_BITS-1}$ in absolute value.

void **fmpz_mat_mul_fmpz_vec**(*fmpz_t* c, const *fmpz_mat_t* A, const *fmpz_t* b, *slong* blen)

void **fmpz_mat_mul_fmpz_vec_ptr**(*fmpz_t* const *c, const *fmpz_mat_t* A, const *fmpz_t* const *b, *slong* blen)

Compute a matrix-vector product of A and (b, blen) and store the result in c. The vector (b, blen) is either truncated or zero-extended to the number of columns of A. The number of entries written to c is always equal to the number of rows of A.

void **fmpz_mat_fmpz_vec_mul**(*fmpz_t* c, const *fmpz_t* a, *slong* alen, const *fmpz_mat_t* B)

void **fmpz_mat_fmpz_vec_mul_ptr**(*fmpz_t* const *c, const *fmpz_t* const *a, *slong* alen, const *fmpz_mat_t* B)

Compute a vector-matrix product of (a, alen) and B and store the result in c. The vector (a, alen) is either truncated or zero-extended to the number of rows of B. The number of entries written to c is always equal to the number of columns of B.

4.5.15 Inverse

int **fmpz_mat_inv**(*fmpz_mat_t* Ainv, *fmpz_t* den, const *fmpz_mat_t* A)

Sets (Ainv, den) to the inverse matrix of A. Returns 1 if A is nonsingular and 0 if A is singular. Aliasing of Ainv and A is allowed.

The denominator is not guaranteed to be minimal, but is guaranteed to be a divisor of the determinant of A.

This function uses a direct formula for matrices of size two or less, and otherwise solves for the identity matrix using fraction-free LU decomposition.

4.5.16 Kronecker product

void **fmpz_mat_kronecker_product**(*fmpz_mat_t* C, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Sets C to the Kronecker product of A and B.

4.5.17 Content

void **fmpz_mat_content**(*fmpz_t* mat_gcd, const *fmpz_mat_t* A)

Sets mat_gcd as the gcd of all the elements of the matrix A. Returns 0 if the matrix is empty.

4.5.18 Trace

void **fmpz_mat_trace**(*fmpz_t* trace, const *fmpz_mat_t* mat)

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

4.5.19 Determinant

void **fmpz_mat_det**(*fmpz_t* det, const *fmpz_mat_t* A)

Sets **det** to the determinant of the square matrix A . The matrix of dimension 0×0 is defined to have determinant 1.

This function automatically chooses between *fmpz_mat_det_cofactor()*, *fmpz_mat_det_bareiss()*, *fmpz_mat_det_modular()* and *fmpz_mat_det_modular_accelerated()* (with **proved** = 1), depending on the size of the matrix and its entries.

void **fmpz_mat_det_cofactor**(*fmpz_t* det, const *fmpz_mat_t* A)

Sets **det** to the determinant of the square matrix A computed using direct cofactor expansion. This function only supports matrices up to size 4×4 .

void **fmpz_mat_det_bareiss**(*fmpz_t* det, const *fmpz_mat_t* A)

Sets **det** to the determinant of the square matrix A computed using the Bareiss algorithm. A copy of the input matrix is row reduced using fraction-free Gaussian elimination, and the determinant is read off from the last element on the main diagonal.

void **fmpz_mat_det_modular**(*fmpz_t* det, const *fmpz_mat_t* A, int proved)

Sets **det** to the determinant of the square matrix A (if **proved** = 1), or a probabilistic value for the determinant (**proved** = 0), computed using a multimodular algorithm.

The determinant is computed modulo several small primes and reconstructed using the Chinese Remainder Theorem. With **proved** = 1, sufficiently many primes are chosen to satisfy the bound computed by *fmpz_mat_det_bound*. With **proved** = 0, the determinant is considered determined if it remains unchanged modulo several consecutive primes (currently if their product exceeds 2^{100}).

void **fmpz_mat_det_modular_accelerated**(*fmpz_t* det, const *fmpz_mat_t* A, int proved)

Sets **det** to the determinant of the square matrix A (if **proved** = 1), or a probabilistic value for the determinant (**proved** = 0), computed using a multimodular algorithm.

This function uses the same basic algorithm as *fmpz_mat_det_modular*, but instead of computing $\det(A)$ directly, it generates a divisor d of $\det(A)$ and then computes $x = \det(A)/d$ modulo several small primes not dividing d . This typically accelerates the computation by requiring fewer primes for large matrices, since d with high probability will be nearly as large as the determinant. This trick is described in [AbbottBronsteinMulders1999].

void **fmpz_mat_det_modular_given_divisor**(*fmpz_t* det, const *fmpz_mat_t* A, const *fmpz_t* d, int proved)

Given a positive divisor d of $\det(A)$, sets **det** to the determinant of the square matrix A (if **proved** = 1), or a probabilistic value for the determinant (**proved** = 0), computed using a multimodular algorithm.

void **fmpz_mat_det_bound**(*fmpz_t* bound, const *fmpz_mat_t* A)

Sets **bound** to a nonnegative integer B such that $|\det(A)| \leq B$. Assumes A to be a square matrix. The bound is computed from the Hadamard inequality $|\det(A)| \leq \prod \|a_i\|_2$ where the product is taken over the rows a_i of A .

void **fmpz_mat_det_bound_nonzero**(*fmpz_t* bound, const *fmpz_mat_t* A)

As per *fmpz_mat_det_bound*() but excludes zero columns. For use with non-square matrices.

void **fmpz_mat_det_divisor**(*fmpz_t* d, const *fmpz_mat_t* A)

Sets *d* to some positive divisor of the determinant of the given square matrix *A*, if the determinant is nonzero. If $|\det(A)| = 0$, *d* will always be set to zero.

A divisor is obtained by solving $Ax = b$ for an arbitrarily chosen right-hand side *b* using Dixon's algorithm and computing the least common multiple of the denominators in *x*. This yields a divisor *d* such that $|\det(A)|/d$ is tiny with very high probability.

4.5.20 Permanent

int **fmpz_mat_permanent**(*fmpz_t* res, const *fmpz_mat_t* A)

Sets *res* to the permanent of the square matrix *A*, returning 1 on success. If the matrix is too large, returns 0.

4.5.21 Transforms

void **fmpz_mat_similarity**(*fmpz_mat_t* A, *slong* r, *fmpz_t* d)

Applies a similarity transform to the $n \times n$ matrix *M* in-place.

If *P* is the $n \times n$ identity matrix the zero entries of whose row *r* (0-indexed) have been replaced by *d*, this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

4.5.22 Characteristic polynomial

void **_fmpz_mat_charpoly_berkowitz**(*fmpz_t* *cp, const *fmpz_mat_t* mat)

Sets (cp, n+1) to the characteristic polynomial of an $n \times n$ square matrix.

void **fmpz_mat_charpoly_berkowitz**(*fmpz_poly_t* cp, const *fmpz_mat_t* mat)

Computes the characteristic polynomial of length $n + 1$ of an $n \times n$ square matrix. Uses an $O(n^4)$ algorithm based on the method of Berkowitz.

void **_fmpz_mat_charpoly_modular**(*fmpz_t* *cp, const *fmpz_mat_t* mat)

Sets (cp, n+1) to the characteristic polynomial of an $n \times n$ square matrix.

void **fmpz_mat_charpoly_modular**(*fmpz_poly_t* cp, const *fmpz_mat_t* mat)

Computes the characteristic polynomial of length $n + 1$ of an $n \times n$ square matrix. Uses a modular method based on an $O(n^3)$ method over $\mathbb{Z}/n\mathbb{Z}$.

void **_fmpz_mat_charpoly**(*fmpz_t* *cp, const *fmpz_mat_t* mat)

Sets (cp, n+1) to the characteristic polynomial of an $n \times n$ square matrix.

void **fmpz_mat_charpoly**(*fmpz_poly_t* cp, const *fmpz_mat_t* mat)

Computes the characteristic polynomial of length $n + 1$ of an $n \times n$ square matrix.

4.5.23 Minimal polynomial

slong **_fmpz_mat_minpoly_modular**(*fmpz_t* *cp, const *fmpz_mat_t* mat)

Sets (cp, n+1) to the modular polynomial of an $n \times n$ square matrix and returns its length.

void **fmpz_mat_minpoly_modular**(*fmpz_poly_t* cp, const *fmpz_mat_t* mat)

Computes the minimal polynomial of an $n \times n$ square matrix. Uses a modular method based on an average time $O(n^3)$, worst case $O(n^4)$ method over $\mathbb{Z}/n\mathbb{Z}$.

slong **fmpz_mat_minpoly**(*fmpz* *cp, const *fmpz_mat_t* mat)

Sets cp to the minimal polynomial of an $n \times n$ square matrix and returns its length.

void **fmpz_mat_minpoly**(*fmpz_poly_t* cp, const *fmpz_mat_t* mat)

Computes the minimal polynomial of an $n \times n$ square matrix.

4.5.24 Rank

slong **fmpz_mat_rank**(const *fmpz_mat_t* A)

Returns the rank, that is, the number of linearly independent columns (equivalently, rows), of A. The rank is computed by row reducing a copy of A.

4.5.25 Column partitioning

int **fmpz_mat_col_partition**(*slong* *part, *fmpz_mat_t* M, int short_circuit)

Returns the number p of distinct columns of M (or 0 if the flag `short_circuit` is set and this number is greater than the number of rows of M). The entries of array `part` are set to values in $[0, p)$ such that two entries of `part` are equal iff the corresponding columns of M are equal. This function is used in van Hoeij polynomial factoring.

4.5.26 Nonsingular solving

The following functions allow solving matrix-matrix equations $AX = B$ where the system matrix A is square and has full rank. The solving is implicitly done over the field of rational numbers: except where otherwise noted, an integer matrix $\hat{X}$ and a separate denominator d (`den`) are computed such that $A(\hat{X}/d) = b$, equivalently such that $A\hat{X} = bd$ holds over the integers. No guarantee is made that the numerators and denominator are reduced to lowest terms, but the denominator is always guaranteed to be a divisor of the determinant of A . If A is singular, `den` will be set to zero and the elements of the solution vector or matrix will have undefined values. No aliasing is allowed between arguments.

int **fmpz_mat_solve**(*fmpz_mat_t* X, *fmpz_t* den, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

This function uses Cramer's rule for small systems and fraction-free LU decomposition followed by fraction-free forward and back substitution for larger systems.

Note that for very large systems, it is faster to compute a modular solution using `fmpz_mat_solve_dixon`.

int **fmpz_mat_solve_fflu**(*fmpz_mat_t* X, *fmpz_t* den, const *fmpz_mat_t* A, const *fmpz_mat_t* B)

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

int **fmpz_mat_solve_fflu_precomp**(*fmpz_mat_t* X, const *slong* *perm, const *fmpz_mat_t* FFLU, const *fmpz_mat_t* B)

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation. If no impossible division is encountered, the function returns 1. This does not mean the system has a solution, however a return value of 0 can only occur if the system is insoluble.

If the return value is 1 and r is the rank of the matrix A whose FFLU we have, then the first r rows of $p(A)y = p(b)d$ hold, where d is the denominator of the FFLU. The remaining rows must be checked by the caller.


```
int fmpz_mat_solve_cramer(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular.

Uses Cramer's rule. Only systems of size up to 3×3 are allowed.

```
void fmpz_mat_solve_bound(fmpz_t N, fmpz_t D, const fmpz_mat_t A, const fmpz_mat_t B)
```

Assuming that A is nonsingular, computes integers N and D such that the reduced numerators and denominators n/d in $A^{-1}B$ satisfy the bounds $0 \leq |n| \leq N$ and $0 \leq d \leq D$.

```
int fmpz_mat_solve_dixon(fmpz_mat_t X, fmpz_t M, const fmpz_mat_t A, const fmpz_mat_t B)
```

Solves $AX = B$ given a nonsingular square matrix A and a matrix B of compatible dimensions, using a modular algorithm. In particular, Dixon's p-adic lifting algorithm is used (currently a non-adaptive version). This is generally the preferred method for large dimensions.

More precisely, this function computes an integer M and an integer matrix X such that $AX = B \bmod M$ and such that all the reduced numerators and denominators of the elements $x = p/q$ in the full solution satisfy $2|p|q < M$. As such, the explicit rational solution matrix can be recovered uniquely by passing the output of this function to `fmpz_mat_set_fmpz_mat_mod`.

A nonzero value is returned if A is nonsingular. If A is singular, zero is returned and the values of the output variables will be undefined.

Aliasing between input and output matrices is allowed.

```
void _fmpz_mat_solve_dixon_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B, const nmod_mat_t Ainv, ulong p, const fmpz_t N, const fmpz_t D)
```

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$ using a p-adic algorithm for the supplied prime p . The values N and D are absolute value bounds for the numerator and denominator of the solution.

Uses the Dixon lifting algorithm with early termination once the lifting stabilises.

```
int fmpz_mat_solve_dixon_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses the Dixon lifting algorithm with early termination once the lifting stabilises.

```
int fmpz_mat_solve_multi_mod_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses a Chinese remainder algorithm with early termination once the lifting stabilises.

```
int fmpz_mat_can_solve_multi_mod_den(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if the system $AX = B$ can be solved. If so it computes (X, den) such that $AX = B \times \text{den}$. The computed denominator will not generally be minimal.

Uses a Chinese remainder algorithm.

Note that the matrices A and B may have any shape as long as they have the same number of rows.

```
int fmpz_mat_can_solve_fflu(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)
```


Returns 1 if the system $AX = B$ can be solved. If so it computes $(\mathbf{X}, \mathbf{den})$ such that $AX = B \times \mathbf{den}$. The computed denominator will not generally be minimal.

Uses a fraction free LU decomposition algorithm.

Note that the matrices A and B may have any shape as long as they have the same number of rows.

int **fmpz_mat_can_solve**(fmpz_mat_t X, fmpz_t den, const fmpz_mat_t A, const fmpz_mat_t B)

Returns 1 if the system $AX = B$ can be solved. If so it computes $(\mathbf{X}, \mathbf{den})$ such that $AX = B \times \mathbf{den}$. The computed denominator will not generally be minimal.

Note that the matrices A and B may have any shape as long as they have the same number of rows.

4.5.27 Row reduction

slong **fmpz_mat_find_pivot_any**(const fmpz_mat_t mat, slong start_row, slong end_row, slong c)

Attempts to find a pivot entry for row reduction. Returns a row index r between **start_row** (inclusive) and **stop_row** (exclusive) such that column c in **mat** has a nonzero entry on row r , or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

slong **fmpz_mat_fflu**(fmpz_mat_t B, fmpz_t den, slong *perm, const fmpz_mat_t A, int rank_check)

Uses fraction-free Gaussian elimination to set $(\mathbf{B}, \mathbf{den})$ to a fraction-free LU decomposition of A and returns the rank of A . Aliasing of A and B is allowed.

Pivot elements are chosen with **fmpz_mat_find_pivot_any**. If **perm** is non-NULL, the permutation of rows in the matrix will also be applied to **perm**.

If **rank_check** is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator **den** is set to $\pm \det(S)$ where S is an appropriate submatrix of A ($S = A$ if A is square) and the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

The fraction-free LU decomposition is defined in [NakTurWil1997].

slong **fmpz_mat_rref**(fmpz_mat_t B, fmpz_t den, const fmpz_mat_t A)

Sets $(\mathbf{B}, \mathbf{den})$ to the reduced row echelon form of A and returns the rank of A . Aliasing of A and B is allowed.

The algorithm used chooses between **fmpz_mat_rref_fflu** and **fmpz_mat_rref_mul** based on the dimensions of the input matrix.

slong **fmpz_mat_rref_fflu**(fmpz_mat_t B, fmpz_t den, const fmpz_mat_t A)

Sets $(\mathbf{B}, \mathbf{den})$ to the reduced row echelon form of A and returns the rank of A . Aliasing of A and B is allowed.

The algorithm proceeds by first computing a row echelon form using **fmpz_mat_fflu**. Letting the upper part of this matrix be $(U|V)P$ where U is full rank upper triangular and P is a permutation matrix, we obtain the rref by setting V to $U^{-1}V$ using back substitution. Scaling each completed row in the back substitution to the denominator **den**, we avoid introducing new fractions. This strategy is equivalent to the fraction-free Gauss-Jordan elimination in [NakTurWil1997], but faster since only the part V corresponding to the null space has to be updated.

The denominator **den** is set to $\pm \det(S)$ where S is an appropriate submatrix of A ($S = A$ if A is square). Note that the determinant is not generally the minimal denominator.

strong **fmpz_mat_rref_mul**(*fmpz_mat_t* B, *fmpz_t* den, const *fmpz_mat_t* A)

Sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.

The algorithm works by computing the reduced row echelon form of A modulo a prime p using **nmod_mat_rref**. The pivot columns and rows of this matrix will then define a non-singular submatrix of A, nonsingular solving and matrix multiplication can then be used to determine the reduced row echelon form of the whole of A. This procedure is described in [Stein2007].

int **fmpz_mat_is_in_rref_with_rank**(const *fmpz_mat_t* A, const *fmpz_t* den, *strong* rank)

Checks that the matrix A/den is in reduced row echelon form of rank **rank**, returns 1 if so and 0 otherwise.

4.5.28 Strong echelon form and Howell form

void **fmpz_mat_strong_echelon_form_mod**(*fmpz_mat_t* A, const *fmpz_t* mod)

Transforms A such that A modulo **mod** is the strong echelon form of the input matrix modulo **mod**. The Howell form and the strong echelon form are equal up to permutation of the rows, see [FieHof2014] for a definition of the strong echelon form and the algorithm used here.

A must have at least as many rows as columns.

strong **fmpz_mat_howell_form_mod**(*fmpz_mat_t* A, const *fmpz_t* mod)

Transforms A such that A modulo **mod** is the Howell form of the input matrix modulo **mod**. For a definition of the Howell form see [StoMul1998]. The Howell form is computed by first putting A into strong echelon form and then ordering the rows.

A must have at least as many rows as columns.

4.5.29 Nullspace

strong **fmpz_mat_nullspace**(*fmpz_mat_t* B, const *fmpz_mat_t* A)

Computes a basis for the right rational nullspace of A and returns the dimension of the nullspace (or nullity). B is set to a matrix with linearly independent columns and maximal rank such that $AB = 0$ (i.e. $Ab = 0$ for each column b in B), and the rank of B is returned.

In general, the entries in B will not be minimal: in particular, the pivot entries in B will generally differ from unity. B must be allocated with sufficient space to represent the result (at most $n \times n$ where n is the number of columns of A).

4.5.30 Echelon form

strong **fmpz_mat_rref_fraction_free**(*strong* *perm, *fmpz_mat_t* B, *fmpz_t* den, const *fmpz_mat_t* A)

Computes an integer matrix B and an integer **den** such that B / den is the unique row reduced echelon form (RREF) of A and returns the rank, i.e. the number of nonzero rows in B.

Aliasing of B and A is allowed, with an in-place computation being more efficient. The size of B must be the same as that of A.

The permutation order will be written to **perm** unless this argument is NULL. That is, row **i** of the output matrix will correspond to row **perm[i]** of the input matrix.

The denominator will always be a divisor of the determinant of (some submatrix of) A, but is not guaranteed to be minimal or canonical in any other sense.

4.5.31 Hermite normal form

void **fmprz_mat_hnf**(*fmprz_mat_t* H, const *fmprz_mat_t* A)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of A. The algorithm used is selected from the implementations in FLINT to be the one most likely to be optimal, based on the characteristics of the input matrix.

Aliasing of H and A is allowed. The size of H must be the same as that of A.

void **fmprz_mat_hnf_transform**(*fmprz_mat_t* H, *fmprz_mat_t* U, const *fmprz_mat_t* A)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of A along with the transformation matrix U such that $UA = H$. The algorithm used is selected from the implementations in FLINT as per **fmprz_mat_hnf**.

Aliasing of H and A is allowed. The size of H must be the same as that of A and U must be square of compatible dimension (having the same number of rows as A).

void **fmprz_mat_hnf_classical**(*fmprz_mat_t* H, const *fmprz_mat_t* A)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of A. The algorithm used is straightforward and is described, for example, in [Algorithm 2.4.4] [Coh1996].

Aliasing of H and A is allowed. The size of H must be the same as that of A.

void **fmprz_mat_hnf_xgcd**(*fmprz_mat_t* H, const *fmprz_mat_t* A)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of A. The algorithm used is an improvement on the basic algorithm and uses extended gcds to speed up computation, this method is described, for example, in [Algorithm 2.4.5] [Coh1996].

Aliasing of H and A is allowed. The size of H must be the same as that of A.

void **fmprz_mat_hnf_modular**(*fmprz_mat_t* H, const *fmprz_mat_t* A, const *fmprz_t* D)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of the $m \times n$ matrix A, where A is assumed to be of rank n and D is known to be a positive multiple of the determinant of the non-zero rows of H. The algorithm used here is due to Domich, Kannan and Trotter [DomKanTro1987] and is also described in [Algorithm 2.4.8] [Coh1996].

Aliasing of H and A is allowed. The size of H must be the same as that of A.

void **fmprz_mat_hnf_modular_eldiv**(*fmprz_mat_t* A, const *fmprz_t* D)

Transforms the $m \times n$ matrix A into Hermite normal form, where A is assumed to be of rank n and D is known to be a positive multiple of the largest elementary divisor of A. The algorithm used here is described in [FieHof2014].

void **fmprz_mat_hnf_minors**(*fmprz_mat_t* H, const *fmprz_mat_t* A)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of the $m \times n$ matrix A, where A is assumed to be of rank n . The algorithm used here is due to Kannan and Bachem [KanBac1979] and takes the principal minors to Hermite normal form in turn.

Aliasing of H and A is allowed. The size of H must be the same as that of A.

void **fmprz_mat_hnf_pernet_stein**(*fmprz_mat_t* H, const *fmprz_mat_t* A, *flint_rand_t* state)

Computes an integer matrix H such that H is the unique (row) Hermite normal form of the $m \times n$ matrix A. The algorithm used here is due to Pernet and Stein [PernetStein2010].

Aliasing of H and A is allowed. The size of H must be the same as that of A.

int **fmprz_mat_is_in_hnf**(const *fmprz_mat_t* A)

Checks that the given matrix is in Hermite normal form, returns 1 if so and 0 otherwise.

4.5.32 Smith normal form

void **fmpz_mat_snf**(*fmpz_mat_t* S, const *fmpz_mat_t* A)

Computes an integer matrix **S** such that **S** is the unique Smith normal form of **A**. The algorithm used is selected from the implementations in FLINT to be the one most likely to be optimal, based on the characteristics of the input matrix.

Aliasing of **S** and **A** is allowed. The size of **S** must be the same as that of **A**.

void **fmpz_mat_snf_diagonal**(*fmpz_mat_t* S, const *fmpz_mat_t* A)

Computes an integer matrix **S** such that **S** is the unique Smith normal form of the diagonal matrix **A**. The algorithm used simply takes gcds of pairs on the diagonal in turn until the Smith form is obtained.

Aliasing of **S** and **A** is allowed. The size of **S** must be the same as that of **A**.

void **fmpz_mat_snf_kannan_bachem**(*fmpz_mat_t* S, const *fmpz_mat_t* A)

Computes an integer matrix **S** such that **S** is the unique Smith normal form of the matrix **A**. The algorithm used here is due to Kannan and Bachem [KanBac1979]

Aliasing of **S** and **A** is allowed. The size of **S** must be the same as that of **A**.

void **fmpz_mat_snf_iliopoulos**(*fmpz_mat_t* S, const *fmpz_mat_t* A, const *fmpz_t* mod)

Computes an integer matrix **S** such that **S** is the unique Smith normal form of the nonsingular $n \times n$ matrix **A**. The algorithm used is due to Iliopoulos [Iliopoulos1989].

Aliasing of **S** and **A** is allowed. The size of **S** must be the same as that of **A**.

int **fmpz_mat_is_in_snf**(const *fmpz_mat_t* A)

Checks that the given matrix is in Smith normal form, returns 1 if so and 0 otherwise.

4.5.33 Special matrices

void **fmpz_mat_gram**(*fmpz_mat_t* B, const *fmpz_mat_t* A)

Sets **B** to the Gram matrix of the m -dimensional lattice **L** in n -dimensional Euclidean space R^n spanned by the rows of the $m \times n$ matrix **A**. Dimensions must be compatible. **A** and **B** are allowed to be the same object if **A** is a square matrix.

int **fmpz_mat_is_hadamard**(const *fmpz_mat_t* H)

Returns nonzero iff **H** is a Hadamard matrix, meaning that it is a square matrix, only has entries that are ± 1 , and satisfies $H^T = nH^{-1}$ where n is the matrix size.

int **fmpz_mat_hadamard**(*fmpz_mat_t* H)

Attempts to set the matrix **H** to a Hadamard matrix, returning 1 if successful and 0 if unsuccessful.

A Hadamard matrix of size n can only exist if n is 1, 2, or a multiple of 4. It is not known whether a Hadamard matrix exists for every size that is a multiple of 4. This function uses the Paley construction, which succeeds for all n of the form $n = 2^e$ or $n = 2^e(q + 1)$ where q is an odd prime power. Orders n for which Hadamard matrices are known to exist but for which this construction fails are 92, 116, 156, ... (OEIS A046116).

4.5.34 Conversions

int `fmpz_mat_get_d_mat`(d_mat_t B, const *fmpz_mat_t* A)

int `fmpz_mat_get_d_mat_transpose`(d_mat_t B, const *fmpz_mat_t* A)

Sets the entries of B as doubles corresponding to the entries of A and the transpose of A , respectively, rounding down towards zero if the latter cannot be represented exactly. The return value is -1 if any entry of A is too large to fit in the normal range of a double, and 0 otherwise.

Note: Requires `d_mat.h` to be included before `fmpz_mat.h` in order to declare these functions.

4.5.35 Cholesky Decomposition

void `fmpz_mat_chol_d`(d_mat_t R, const *fmpz_mat_t* A)

Computes R , the Cholesky factor of a symmetric, positive definite matrix A using the Cholesky decomposition process. (Sets R such that $A = RR^T$ where R is a lower triangular matrix.)

Note: Requires `d_mat.h` to be included before `fmpz_mat.h` in order to declare this function.

void `fmpz_mat_is_spd`(const *fmpz_mat_t* A)

Returns true iff A is symmetric and positive definite (in particular square).

We first attempt a numerical LDL^T decomposition using `arb_mat_ldl()`. If we cannot guarantee that A is positive definite, we use an exact method instead, computing the characteristic polynomial of A and applying Descartes' rule of signs.

4.5.36 LLL

int `fmpz_mat_is_reduced`(const *fmpz_mat_t* A, double delta, double eta)

int `fmpz_mat_is_reduced_gram`(const *fmpz_mat_t* A, double delta, double eta)

Returns a non-zero value if the basis A is LLL-reduced with factor `(delta, eta)`, and otherwise returns zero. The second version assumes A is the Gram matrix of the basis.

int `fmpz_mat_is_reduced_with_removal`(const *fmpz_mat_t* A, double delta, double eta, const *fmpz_t* gs_B, int newd)

int `fmpz_mat_is_reduced_gram_with_removal`(const *fmpz_mat_t* A, double delta, double eta, const *fmpz_t* gs_B, int newd)

Returns a non-zero value if the basis A is LLL-reduced with factor `(delta, eta)` for each of the first `newd` vectors and the squared Gram-Schmidt length of each of the remaining i -th vectors (where $i \geq \text{newd}$) is greater than `gs_B`, and otherwise returns zero. The second version assumes A is the Gram matrix of the basis.

4.5.37 Classical LLL

void `fmpz_mat_lll_original`(*fmpz_mat_t* A, const *fmpz_t* delta, const *fmpz_t* eta)

Takes a basis $x_1, x_2, \dots, x_m$ of the lattice $L \subset R^n$ (as the rows of a $m \times n$ matrix A). The output is a `(delta, eta)`-reduced basis $y_1, y_2, \dots, y_m$ of the lattice L (as the rows of the same $m \times n$ matrix A).

4.5.38 Modified LLL

void **fmpz_mat_lla_storjohann**(*fmpz_mat_t* A, const *fmpz_t* delta, const *fmpz_t* eta)

Takes a basis $x_1, x_2, \dots, x_m$ of the lattice $L \subset R^n$ (as the rows of a $m \times n$ matrix A). The output is an (*delta*, *eta*)-reduced basis $y_1, y_2, \dots, y_m$ of the lattice L (as the rows of the same $m \times n$ matrix A). Uses a modified version of LLL, which has better complexity in terms of the lattice dimension, introduced by Storjohann.

See “Faster Algorithms for Integer Lattice Basis Reduction.” Technical Report 249. Zurich, Switzerland: Department Informatik, ETH. July 30, 1996.

4.6 fmpz_lll.h – LLL reduction

4.6.1 Parameter manipulation

These functions are used to initialise LLL context objects which are of the type *fmpz_lll_t*. These objects contain all information about the options governing the reduction using this module’s functions including the LLL parameters *delta* and *eta*, the representation type of the input matrix (whether it is a lattice basis or a Gram matrix), and the type of Gram matrix to be used during L^2 (approximate or exact).

void **fmpz_lll_context_init_default**(*fmpz_lll_t* fl)

Sets *fl->delta*, *fl->eta*, *fl->rt* and *fl->gt* to their default values, 0.99, 0.51, *Z_BASIS* and *APPROX* respectively.

void **fmpz_lll_context_init**(*fmpz_lll_t* fl, double delta, double eta, rep_type rt, gram_type gt)

Sets *fl->delta*, *fl->eta*, *fl->rt* and *fl->gt* to *delta*, *eta*, *rt* and *gt* (given as input) respectively. *delta* and *eta* are the L^2 parameters. *delta* and *eta* must lie in the intervals (0.25, 1) and $(0.5, \sqrt{\text{delta}})$ respectively. The representation type is input using *rt* and can have the values *Z_BASIS* for a lattice basis and *GRAM* for a Gram matrix. The Gram type to be used during computation can be specified using *gt* which can assume the values *APPROX* and *EXACT*. Note that *gt* has meaning only when *rt* is *Z_BASIS*.

4.6.2 Random parameter generation

void **fmpz_lll_randtest**(*fmpz_lll_t* fl, *flint_rand_t* state)

Sets *fl->delta* and *fl->eta* to random values in the interval (0.25, 1) and $(0.5, \sqrt{\text{delta}})$ respectively. *fl->rt* is set to *GRAM* or *Z_BASIS* and *fl->gt* is set to *APPROX* or *EXACT* in a pseudo random way.

4.6.3 Heuristic dot product

double **fmpz_lll_heuristic_dot**(const double *vec1, const double *vec2, *slong* len2, const *fmpz_mat_t* B, *slong* k, *slong* j, *slong* exp_adj)

Computes the dot product of two vectors of doubles *vec1* and *vec2*, which are respectively *double* approximations (up to scaling by a power of 2) to rows *k* and *j* in the exact integer matrix B. If massive cancellation is detected an exact computation is made.

The exact computation is scaled by $2^{-\text{exp\_adj}}$, where $\text{exp\_adj} = r_2 + r_1$ where r_2 is the exponent for row *j* and r_1 is the exponent for row *k* (i.e. row *j* is notionally thought of as being multiplied by 2^{r_2} , etc.).

The final dot product computed by this function is then notionally the return value times $2^{\text{exp\_adj}}$.

4.6.4 The various Babai's

```
int fmpz_lll_check_babai(int kappa, fmpz_mat_t B, fmpz_mat_t U, d_mat_t mu, d_mat_t r,
                        double *s, d_mat_t appB, int *expo, fmpz_gram_t A, int a, int zeros, int
                        kappamax, int n, const fmpz_lll_t fl)
```

Performs floating point size reductions of the kappa -th row of B by all of the previous rows, uses d_mats mu and r for storing the GSO data. U is used to capture the unimodular transformations if it is not *NULL*. The *double* array s will contain the size of the kappa -th row if it were moved into position i . The d_mat appB is an approximation of B with each row receiving an exponent stored in expo which gets populated only when needed. The d_mat $\text{A} \rightarrow \text{appSP}$ is an approximation of the Gram matrix whose entries are scalar products of the rows of B and is used when $\text{fl} \rightarrow \text{gt} == \text{APPROX}$. When $\text{fl} \rightarrow \text{gt} == \text{EXACT}$ the fmpz_mat $\text{A} \rightarrow \text{exactSP}$ (the exact Gram matrix) is used. The index a is the smallest row index which will be reduced from the kappa -th row. Index zeros is the number of zero rows in the matrix. kappamax is the highest index which has been size-reduced so far, and n is the number of columns you want to consider. fl is an LLL (L^2) context object. The output is the value -1 if the process fails (usually due to insufficient precision) or 0 if everything was successful. These descriptions will be true for the future Babai procedures as well.

```
int fmpz_lll_check_babai_heuristic_d(int kappa, fmpz_mat_t B, fmpz_mat_t U, d_mat_t mu,
                                     d_mat_t r, double *s, d_mat_t appB, int *expo,
                                     fmpz_gram_t A, int a, int zeros, int kappamax, int n, const
                                     fmpz_lll_t fl)
```

Same as *fmpz_lll_check_babai()* but using the heuristic inner product rather than a purely floating point inner product. The heuristic will compute at full precision when there is cancellation.

```
int fmpz_lll_check_babai_heuristic(int kappa, fmpz_mat_t B, fmpz_mat_t U, gr_mat_t mu,
                                   gr_mat_t r, gr_ptr s, gr_mat_t appB, fmpz_gram_t A, int
                                   a, int zeros, int kappamax, int n, gr_ptr tmp, gr_ptr rtmp,
                                   gr_ctx_t ctx, const fmpz_lll_t fl)
```

This function is like *fmpz_lll_check_babai_heuristic_d()* but works with a generic floating-point type defined by the given context object ctx . Temporary scratch variables tmp and rtmp must be given as input.

```
int fmpz_lll_advance_check_babai(int cur_kappa, int kappa, fmpz_mat_t B, fmpz_mat_t U,
                                 d_mat_t mu, d_mat_t r, double *s, d_mat_t appB, int *expo,
                                 fmpz_gram_t A, int a, int zeros, int kappamax, int n, const
                                 fmpz_lll_t fl)
```

This is a Babai procedure which is used when size reducing a vector beyond an index which LLL has reached. cur_kappa is the index behind which we can assume B is LLL reduced, while kappa is the vector to be reduced. This procedure only size reduces the kappa -th row by vectors up to cur_kappa , not $\text{kappa} - 1$.

```
int fmpz_lll_advance_check_babai_heuristic_d(int cur_kappa, int kappa, fmpz_mat_t B,
                                              fmpz_mat_t U, d_mat_t mu, d_mat_t r, double
                                              *s, d_mat_t appB, int *expo, fmpz_gram_t A,
                                              int a, int zeros, int kappamax, int n, const
                                              fmpz_lll_t fl)
```

Same as *fmpz_lll_advance_check_babai()* but using the heuristic inner product rather than a purely floating point inner product. The heuristic will compute at full precision when there is cancellation.

4.6.5 Shift

int **fmprz_lll_shift**(const *fmprz_mat_t* B)

Computes the largest number of non-zero entries after the diagonal in B.

4.6.6 Varieties of LLL

These programs implement ideas from the book chapter [Stehle2010]. The list of function here that are heuristic in nature and may return with *B* unreduced - that is to say, not do their job - includes (but is not necessarily limited to):

- *fmprz_lll_d()*
- *fmprz_lll_d_heuristic()*
- *fmprz_lll_d_heuristic_with_removal()*
- *fmprz_lll_d_with_removal()*
- *fmprz_lll_d_with_removal_knapsack()*

int **fmprz_lll_d**(*fmprz_mat_t* B, *fmprz_mat_t* U, const *fmprz_lll_t* fl)

This is a mildly greedy version of floating point LLL using doubles only. It tries the fast version of the Babai algorithm (*fmprz_lll_check_babai()*). If that fails, then it switches to the heuristic version (*fmprz_lll_check_babai_heuristic_d()*) for only one loop and switches right back to the fast version. It reduces B in place. U is the matrix used to capture the unimodular transformations if it is not *NULL*. An exception is raised if *U* != *NULL* and *U*->*r* != *d*, where *d* is the lattice dimension. fl is the context object containing information containing the LLL parameters delta and eta. The function can perform reduction on both the lattice basis as well as its Gram matrix. The type of lattice representation can be specified via the parameter fl->rt. The type of Gram matrix to be used in computation (approximate or exact) can also be specified through the variable fl->gt (applies only if fl->rt == *Z_BASIS*).

int **fmprz_lll_d_heuristic**(*fmprz_mat_t* B, *fmprz_mat_t* U, const *fmprz_lll_t* fl)

This LLL reduces B in place using doubles only. It is similar to *fmprz_lll_d()* but only uses the heuristic inner products which attempt to detect cancellations.

int **fmprz_lll_mpf2**(*fmprz_mat_t* B, *fmprz_mat_t* U, *flint_bitcnt_t* prec, const *fmprz_lll_t* fl)

This is LLL using mpf with the given precision, prec for the underlying GSO. It reduces B in place like the other LLL functions. The original implementation of this function used GMP's mpf for arithmetic; it currently uses nfloat if prec is small enough and otherwise falls back to arf.

int **fmprz_lll_mpf**(*fmprz_mat_t* B, *fmprz_mat_t* U, const *fmprz_lll_t* fl)

A wrapper of *fmprz_lll_mpf2()*. This currently begins with *prec* == 64, then for the first 20 loops, increases the precision one limb at a time. After 20 loops, it doubles the precision each time. There is a proof that this will eventually work. The return value of this function is 0 if the LLL is successful or -1 if the precision maxes out before B is LLL-reduced.

int **fmprz_lll_wrapper**(*fmprz_mat_t* B, *fmprz_mat_t* U, const *fmprz_lll_t* fl)

A wrapper of the above procedures. It begins with the greediest version (*fmprz_lll_d()*), then adapts to the version using heuristic inner products only (*fmprz_lll_d_heuristic()*) if fl->rt == *Z_BASIS* and fl->gt == *APPROX*, and finally to the mpf version (*fmprz_lll_mpf()*) if needed.

U is the matrix used to capture the unimodular transformations if it is not *NULL*. An exception is raised if *U* != *NULL* and *U*->*r* != *d*, where *d* is the lattice dimension. fl is the context object containing information containing the LLL parameters delta and eta. The function can perform reduction on both the lattice basis as well as its Gram matrix. The type of lattice representation can be specified via the parameter fl->rt. The type of Gram matrix to be used in computation (approximate or exact) can also be specified through the variable fl->gt (applies only if fl->rt == *Z_BASIS*).


```
int fmpz_lll_d_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```

Same as `fmpz_lll_d()` but with a removal bound, `gs_B`. The return value is the new dimension of `B` if removals are desired.

```
int fmpz_lll_d_heuristic_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```

Same as `fmpz_lll_d_heuristic()` but with a removal bound, `gs_B`. The return value is the new dimension of `B` if removals are desired.

```
int fmpz_lll_mpf2_with_removal(fmpz_mat_t B, fmpz_mat_t U, flint_bitcnt_t prec, const fmpz_t gs_B, const fmpz_lll_t fl)
```

Same as `fmpz_lll_mpf2()` but with a removal bound, `gs_B`. The return value is the new dimension of `B` if removals are desired.

```
int fmpz_lll_mpf_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```

A wrapper of `fmpz_lll_mpf2_with_removal()`. This currently begins with `prec == D_BITS`, then for the first 20 loops, increases the precision one limb at a time. After 20 loops, it doubles the precision each time. There is a proof that this will eventually work. The return value of this function is the new dimension of `B` if removals are desired or -1 if the precision maxes out before `B` is LLL-reduced.

```
int fmpz_lll_wrapper_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```

A wrapper of the procedures implementing the base case LLL with the addition of the removal boundary. It begins with the greediest version (`fmpz_lll_d_with_removal()`), then adapts to the version using heuristic inner products only (`fmpz_lll_d_heuristic_with_removal()`) if `fl->rt == Z_BASIS` and `fl->gt == APPROX`, and finally to the mpf version (`fmpz_lll_mpf_with_removal()`) if needed.

```
int fmpz_lll_d_with_removal_knapsack(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```

This is floating point LLL specialized to knapsack-type lattices. It performs early size reductions occasionally which makes things faster in the knapsack case. Otherwise, it is similar to `fmpz_lll_d_with_removal`.

```
int fmpz_lll_wrapper_with_removal_knapsack(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)
```

A wrapper of the procedures implementing the LLL specialized to knapsack-type lattices. It begins with the greediest version and the engine of this version, (`fmpz_lll_d_with_removal_knapsack()`), then adapts to the version using heuristic inner products only (`fmpz_lll_d_heuristic_with_removal()`) if `fl->rt == Z_BASIS` and `fl->gt == APPROX`, and finally to the mpf version (`fmpz_lll_mpf_with_removal()`) if needed.

4.6.7 ULLL

```
int fmpz_lll_with_removal_ulll(fmpz_mat_t FM, fmpz_mat_t UM, slong new_size, const fmpz_t gs_B, const fmpz_lll_t fl)
```

ULLL is a new style of LLL which adjoins an identity matrix to the input lattice `FM`, then scales the lattice down to `new_size` bits and reduces this augmented lattice. This tends to be more stable numerically than traditional LLL which means higher dimensions can be attacked using doubles. In each iteration a new identity matrix is adjoined to the truncated lattice. `UM` is used to capture the unimodular transformations, while `gs_B` and `fl` have the same role as in the previous routines. The function is optimised for factoring polynomials.

4.6.8 LLL-reducedness

These programs implement ideas from the paper [Villard2007]. See <https://arxiv.org/abs/cs/0701183> for the algorithm of Villard.

```

int fmpz_lll_is_reduced_d(const fmpz_mat_t B, const fmpz_lll_t fl)
int fmpz_lll_is_reduced_mpfr(const fmpz_mat_t B, const fmpz_lll_t fl, flint_bitcnt_t prec)
int fmpz_lll_is_reduced_d_with_removal(const fmpz_mat_t B, const fmpz_lll_t fl, const fmpz_t
                                     gs_B, int newd)
int fmpz_lll_is_reduced_mpfr_with_removal(const fmpz_mat_t B, const fmpz_lll_t fl, const
                                     fmpz_t gs_B, int newd, flint_bitcnt_t prec)

```

A non-zero return indicates the matrix is definitely reduced, that is, that `*fmpz_mat_is_reduced()` or `fmpz_mat_is_reduced_gram()` (for the first two) `*fmpz_mat_is_reduced_with_removal()` or `fmpz_mat_is_reduced_gram_with_removal()` (for the last two) return non-zero. A zero return value is inconclusive. The *d* variants are performed in machine precision, while the *mpfr* uses a precision of *prec* bits. The current *mpfr* implementations actually use the *nfloating* type instead of MPFR for arithmetic.

```

int fmpz_lll_is_reduced(const fmpz_mat_t B, const fmpz_lll_t fl, flint_bitcnt_t prec)
int fmpz_lll_is_reduced_with_removal(const fmpz_mat_t B, const fmpz_lll_t fl, const fmpz_t
                                     gs_B, int newd, flint_bitcnt_t prec)

```

The return from these functions is always conclusive: the functions `*fmpz_mat_is_reduced()` or `fmpz_mat_is_reduced_gram()` `*fmpz_mat_is_reduced_with_removal()` or `fmpz_mat_is_reduced_gram_with_removal()` are optimized by calling the above heuristics first and returning right away if they give a conclusive answer.

4.6.9 Modified ULLL

```

void fmpz_lll_storjohann_ulll(fmpz_mat_t FM, slong new_size, const fmpz_lll_t fl)

```

Performs ULLL using `fmpz_mat_lll_storjohann()` as the LLL function.

Note: This function is currently not tested. Use at your own risk.

4.6.10 Main LLL functions

```

void fmpz_lll(fmpz_mat_t B, fmpz_mat_t U, const fmpz_lll_t fl)

```

Reduces B in place according to the parameters specified by the LLL context object *fl*.

This is the main LLL function which should be called by the user. It currently calls the ULLL algorithm (without removals). The ULLL function in turn calls a LLL wrapper which tries to choose an optimal LLL algorithm, starting with a version using just doubles (ULLL tries to maximise usage of this), then a heuristic LLL followed by a full precision floating point LLL if required.

U is the matrix used to capture the unimodular transformations if it is not *NULL*. An exception is raised if *U* != *NULL* and *U->r* != *d*, where *d* is the lattice dimension. *fl* is the context object containing information containing the LLL parameters delta and eta. The function can perform reduction on both the lattice basis as well as its Gram matrix. The type of lattice representation can be specified via the parameter *fl->rt*. The type of Gram matrix to be used in computation (approximate or exact) can also be specified through the variable *fl->gt* (applies only if *fl->rt* == *Z_BASIS*).

```

int fmpz_lll_with_removal(fmpz_mat_t B, fmpz_mat_t U, const fmpz_t gs_B, const fmpz_lll_t fl)

```

Reduces B in place according to the parameters specified by the LLL context object *fl* and removes vectors whose squared Gram-Schmidt length is greater than the bound *gs_B*. The return value is the new dimension of B to be considered for further computation.

This is the main LLL with removals function which should be called by the user. Like `fmpz_lll` it calls ULLL, but it also sets the Gram-Schmidt bound to that supplied and does removals.

4.7 fmpz_poly.h – univariate polynomials over the integers

4.7.1 Introduction

The `fmpz_poly_t` data type represents elements of $\mathbb{Z}[x]$. The `fmpz_poly` module provides routines for memory management, basic arithmetic, and conversions from or to other types.

Each coefficient of an `fmpz_poly_t` is an integer of the FLINT `fmpz_t` type. There are two advantages of this model. Firstly, the `fmpz_t` type is memory managed, so the user can manipulate individual coefficients of a polynomial without having to deal with tedious memory management. Secondly, a coefficient of an `fmpz_poly_t` can be changed without changing the size of any of the other coefficients.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

4.7.2 Simple example

The following example computes the square of the polynomial $5x^3 - 1$.

```
#include "fmpz_poly.h"

int main()
{
    fmpz_poly_t x, y;
    fmpz_poly_init(x);
    fmpz_poly_init(y);
    fmpz_poly_set_coeff_ui(x, 3, 5);
    fmpz_poly_set_coeff_si(x, 0, -1);
    fmpz_poly_mul(y, x, x);
    fmpz_poly_print(x); flint_printf("\n");
    fmpz_poly_print(y); flint_printf("\n");
    fmpz_poly_clear(x);
    fmpz_poly_clear(y);
}
```

The output is:

```
4  -1 0 0 5
7  1 0 0 -10 0 0 25
```

4.7.3 Definition of the fmpz_poly_t type

The `fmpz_poly_t` type is a typedef for an array of length 1 of `fmpz_poly_struct`'s. This permits passing parameters of type `fmpz_poly_t` by reference in a manner similar to the way GMP integers of type `mpz_t` can be passed by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `fmpz_poly_t`. For simplicity we will think of an `fmpz_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g. to access the `length` field of an `fmpz_poly_t` called `poly1` one writes `poly1->length`.

An `fmpz_poly_t` is said to be *normalised* if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `fmpz_poly` functions expect their inputs to be normalised, and unless otherwise specified they produce output that is normalised.

It is recommended that users do not access the fields of an `fmpz_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `fmpz_poly` do all the memory management for the user. One does not need to specify the maximum length or number of limbs per coefficient in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required. Each coefficient is also managed separately, being resized as needed, independently of the other coefficients.

4.7.4 Types, macros and constants

type `fmpz_poly_struct`

type `fmpz_poly_t`

4.7.5 Memory management

void `fmpz_poly_init(fmpz_poly_t poly)`

Initialises `poly` for use, setting its length to zero. A corresponding call to `fmpz_poly_clear()` must be made after finishing with the `fmpz_poly_t` to free the memory used by the polynomial.

void `fmpz_poly_init2(fmpz_poly_t poly, slong alloc)`

Initialises `poly` with space for at least `alloc` coefficients and sets the length to zero. The allocated coefficients are all set to zero.

void `fmpz_poly_realloc(fmpz_poly_t poly, slong alloc)`

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` the polynomial is first truncated to length `alloc`.

void `fmpz_poly_fit_length(fmpz_poly_t poly, slong len)`

If `len` is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least `len` coefficients. No data is lost when calling this function.

The function efficiently deals with the case where `fit_length` is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void `fmpz_poly_clear(fmpz_poly_t poly)`

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void `_fmpz_poly_normalise(fmpz_poly_t poly)`

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void `_fmpz_poly_set_length(fmpz_poly_t poly, slong newlen)`

Demotes the coefficients of `poly` beyond `newlen` and sets the length of `poly` to `newlen`.

void `fmpz_poly_attach_truncate(fmpz_poly_t trunc, const fmpz_poly_t poly, slong n)`

This function sets the uninitialised polynomial `trunc` to the low `n` coefficients of `poly`, or to `poly` if the latter doesn't have `n` coefficients. The polynomial `trunc` not be cleared or used as the output of any FLINT functions.

void **fmpz_poly_attach_shift**(*fmpz_poly_t* trunc, const *fmpz_poly_t* poly, *slong* n)

This function sets the uninitialised polynomial **trunc** to the high coefficients of **poly**, i.e. the coefficients not among the low n coefficients of **poly**. If the latter doesn't have n coefficients **trunc** is set to the zero polynomial. The polynomial **trunc** not be cleared or used as the output of any FLINT functions.

4.7.6 Polynomial parameters

slong **fmpz_poly_length**(const *fmpz_poly_t* poly)

Returns the length of **poly**. The zero polynomial has length zero.

slong **fmpz_poly_degree**(const *fmpz_poly_t* poly)

Returns the degree of **poly**, which is one less than its length.

4.7.7 Assignment and basic manipulation

void **fmpz_poly_set**(*fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets **poly1** to equal **poly2**.

void **fmpz_poly_set_si**(*fmpz_poly_t* poly, *slong* c)

Sets **poly** to the signed integer **c**.

void **fmpz_poly_set_ui**(*fmpz_poly_t* poly, *ulong* c)

Sets **poly** to the unsigned integer **c**.

void **fmpz_poly_set_fmpz**(*fmpz_poly_t* poly, const *fmpz_t* c)

Sets **poly** to the integer **c**.

int **_fmpz_poly_set_str**(*fmpz_t* *poly, const char *str)

Sets **poly** to the polynomial encoded in the null-terminated string **str**. Assumes that **poly** is allocated as a sufficiently large array suitable for the number of coefficients present in **str**.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of **poly** is undefined. If **str** is not null-terminated, calling this method might result in a segmentation fault.

int **fmpz_poly_set_str**(*fmpz_poly_t* poly, const char *str)

Imports a polynomial from a null-terminated string. If the string **str** represents a valid polynomial returns 0, otherwise returns 1.

Returns 0 if no error occurred. Otherwise, returns a non-zero value, in which case the resulting value of **poly** is undefined. If **str** is not null-terminated, calling this method might result in a segmentation fault.

char *_**fmpz_poly_get_str**(const *fmpz_t* *poly, *slong* len)

Returns the plain FLINT string representation of the polynomial (**poly**, **len**).

char *_**fmpz_poly_get_str**(const *fmpz_poly_t* poly)

Returns the plain FLINT string representation of the polynomial **poly**.

char *_**fmpz_poly_get_str_pretty**(const *fmpz_t* *poly, *slong* len, const char *x)

Returns a pretty representation of the polynomial (**poly**, **len**) using the null-terminated string **x** as the variable name.

char *_**fmpz_poly_get_str_pretty**(const *fmpz_poly_t* poly, const char *x)

Returns a pretty representation of the polynomial **poly** using the null-terminated string **x** as the variable name.

void **mpz_poly_zero**(*mpz_poly_t* poly)

Sets poly to the zero polynomial.

void **mpz_poly_one**(*mpz_poly_t* poly)

Sets poly to the constant polynomial one.

void **mpz_poly_zero_coeffs**(*mpz_poly_t* poly, *slong* i, *slong* j)

Sets the coefficients of $x^i, \dots, x^{j-1}$ to zero.

void **mpz_poly_swap**(*mpz_poly_t* poly1, *mpz_poly_t* poly2)

Swaps poly1 and poly2. This is done efficiently without copying data by swapping pointers, etc.

void **_mpz_poly_reverse**(*mpz_t* res, const *mpz_t* poly, *slong* len, *slong* n)

Sets (res, n) to the reverse of (poly, n), where poly is in fact an array of length len. Assumes that $0 < \text{len} \leq n$. Supports aliasing of res and poly, but the behaviour is undefined in case of partial overlap.

void **mpz_poly_reverse**(*mpz_poly_t* res, const *mpz_poly_t* poly, *slong* n)

This function considers the polynomial poly to be of length n, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result res may be of length less than n.

void **mpz_poly_truncate**(*mpz_poly_t* poly, *slong* newlen)

If the current length of poly is greater than newlen, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

void **mpz_poly_set_trunc**(*mpz_poly_t* res, const *mpz_poly_t* poly, *slong* n)

Sets res to a copy of poly, truncated to length n.

4.7.8 Randomisation

void **mpz_poly_randtest**(*mpz_poly_t* f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets f to a random polynomial with up to the given length and where each coefficient has up to the given number of bits. The coefficients are signed randomly.

void **mpz_poly_randtest_unsigned**(*mpz_poly_t* f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets f to a random polynomial with up to the given length and where each coefficient has up to the given number of bits.

void **mpz_poly_randtest_not_zero**(*mpz_poly_t* f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

As for *mpz_poly_randtest*() except that len and bits may not be zero and the polynomial generated is guaranteed not to be the zero polynomial.

void **mpz_poly_randtest_no_real_root**(*mpz_poly_t* p, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets p to a random polynomial without any real root, whose length is up to len and where each coefficient has up to the given number of bits.

void **mpz_poly_randtest_irreducible1**(*mpz_poly_t* pol, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

void **mpz_poly_randtest_irreducible2**(*mpz_poly_t* pol, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

void **mpz_poly_randtest_irreducible**(*mpz_poly_t* pol, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets p to a random irreducible polynomial, whose length is up to len and where each coefficient has up to the given number of bits. There are two algorithms: *irreducible1* generates an irreducible polynomial modulo a random prime number and lifts it to the integers; *irreducible2* generates a random integer polynomial, factors it, and returns a random factor. The default function chooses randomly between these methods.

4.7.9 Getting and setting coefficients

void **fmpz_poly_get_coeff_fmpz**(*fmpz_t* x, const *fmpz_poly_t* poly, *slong* n)

Sets *x* to the *n*-th coefficient of **poly**. Coefficient numbering is from zero and if *n* is set to a value beyond the end of the polynomial, zero is returned.

slong **fmpz_poly_get_coeff_si**(const *fmpz_poly_t* poly, *slong* n)

Returns coefficient *n* of **poly** as a *slong*. The result is undefined if the value does not fit into a *slong*. Coefficient numbering is from zero and if *n* is set to a value beyond the end of the polynomial, zero is returned.

ulong **fmpz_poly_get_coeff_ui**(const *fmpz_poly_t* poly, *slong* n)

Returns coefficient *n* of **poly** as a *ulong*. The result is undefined if the value does not fit into a *ulong*. Coefficient numbering is from zero and if *n* is set to a value beyond the end of the polynomial, zero is returned.

fmpz ***fmpz_poly_get_coeff_ptr**(const *fmpz_poly_t* poly, *slong* n)

Returns a reference to the coefficient of x^n in the polynomial, as an **fmpz** *. This function is provided so that individual coefficients can be accessed and operated on by functions in the **fmpz** module. This function does not make a copy of the data, but returns a reference to the actual coefficient.

Returns NULL when *n* exceeds the degree of the polynomial.

This function is implemented as a macro.

fmpz ***fmpz_poly_lead**(const *fmpz_poly_t* poly)

Returns a reference to the leading coefficient of the polynomial, as an **fmpz** *. This function is provided so that the leading coefficient can be easily accessed and operated on by functions in the **fmpz** module. This function does not make a copy of the data, but returns a reference to the actual coefficient.

Returns NULL when the polynomial is zero.

This function is implemented as a macro.

void **fmpz_poly_set_coeff_fmpz**(*fmpz_poly_t* poly, *slong* n, const *fmpz_t* x)

Sets coefficient *n* of **poly** to the **fmpz** value *x*. Coefficient numbering starts from zero and if *n* is beyond the current length of **poly** then the polynomial is extended and zero coefficients inserted if necessary.

void **fmpz_poly_set_coeff_si**(*fmpz_poly_t* poly, *slong* n, *slong* x)

Sets coefficient *n* of **poly** to the *slong* value *x*. Coefficient numbering starts from zero and if *n* is beyond the current length of **poly** then the polynomial is extended and zero coefficients inserted if necessary.

void **fmpz_poly_set_coeff_ui**(*fmpz_poly_t* poly, *slong* n, *ulong* x)

Sets coefficient *n* of **poly** to the *ulong* value *x*. Coefficient numbering starts from zero and if *n* is beyond the current length of **poly** then the polynomial is extended and zero coefficients inserted if necessary.

4.7.10 Comparison

int **fmpz_poly_equal**(const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Returns 1 if poly1 is equal to poly2, otherwise returns 0. The polynomials are assumed to be normalised.

int **fmpz_poly_equal_trunc**(const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2, *slong* n)

Return 1 if poly1 and poly2, notionally truncated to length n are equal, otherwise return 0.

int **fmpz_poly_is_zero**(const *fmpz_poly_t* poly)

Returns 1 if the polynomial is zero and 0 otherwise.

This function is implemented as a macro.

int **fmpz_poly_is_one**(const *fmpz_poly_t* poly)

Returns 1 if the polynomial is one and 0 otherwise.

int **fmpz_poly_is_unit**(const *fmpz_poly_t* poly)

Returns 1 if the polynomial is the constant polynomial ± 1 , and 0 otherwise.

int **fmpz_poly_is_gen**(const *fmpz_poly_t* poly)

Returns 1 if the polynomial is the degree 1 polynomial x , and 0 otherwise.

4.7.11 Addition and subtraction

void **_fmpz_poly_add**(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2)

Sets **res** to the sum of (poly1, len1) and (poly2, len2). It is assumed that **res** has sufficient space for the longer of the two polynomials.

void **fmpz_poly_add**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets **res** to the sum of poly1 and poly2.

void **fmpz_poly_add_series**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2, *slong* n)

Notionally truncate poly1 and poly2 to length n and then set **res** to the sum.

void **_fmpz_poly_sub**(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2)

Sets **res** to (poly1, len1) minus (poly2, len2). It is assumed that **res** has sufficient space for the longer of the two polynomials.

void **fmpz_poly_sub**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets **res** to poly1 minus poly2.

void **fmpz_poly_sub_series**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2, *slong* n)

Notionally truncate poly1 and poly2 to length n and then set **res** to the sum.

void **fmpz_poly_neg**(*fmpz_poly_t* res, const *fmpz_poly_t* poly)

Sets **res** to $-\text{poly}$.

4.7.12 Scalar absolute value, multiplication and division

```
void fmpz_poly_scalar_abs(fmpz_poly_t res, const fmpz_poly_t poly)
    Sets poly1 to the polynomial whose coefficients are the absolute value of those of poly2.
```

```
void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
    Sets poly1 to poly2 times x.
```

```
void fmpz_poly_scalar_mul_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
    Sets poly1 to poly2 times the signed slong x.
```

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
    Sets poly1 to poly2 times the ulong x.
```

```
void fmpz_poly_scalar_mul_2exp(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong exp)
    Sets poly1 to poly2 times  $2^{\text{exp}}$ .
```

```
void fmpz_poly_scalar_addmul_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
void fmpz_poly_scalar_addmul_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
void fmpz_poly_scalar_addmul_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
    Sets poly1 to  $\text{poly1} + x * \text{poly2}$ .
```

```
void fmpz_poly_scalar_submul_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
    Sets poly1 to  $\text{poly1} - x * \text{poly2}$ .
```

```
void fmpz_poly_scalar_fdiv_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
    Sets poly1 to poly2 divided by the fmpz_t x, rounding coefficients down toward  $-\infty$ .
```

```
void fmpz_poly_scalar_fdiv_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
    Sets poly1 to poly2 divided by the slong x, rounding coefficients down toward  $-\infty$ .
```

```
void fmpz_poly_scalar_fdiv_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
    Sets poly1 to poly2 divided by the ulong x, rounding coefficients down toward  $-\infty$ .
```

```
void fmpz_poly_scalar_fdiv_2exp(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
    Sets poly1 to poly2 divided by  $2^x$ , rounding coefficients down toward  $-\infty$ .
```

```
void fmpz_poly_scalar_tdiv_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t x)
    Sets poly1 to poly2 divided by the fmpz_t x, rounding coefficients toward 0.
```

```
void fmpz_poly_scalar_tdiv_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
    Sets poly1 to poly2 divided by the slong x, rounding coefficients toward 0.
```

```
void fmpz_poly_scalar_tdiv_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
    Sets poly1 to poly2 divided by the ulong x, rounding coefficients toward 0.
```

```
void fmpz_poly_scalar_tdiv_2exp(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
    Sets poly1 to poly2 divided by  $2^x$ , rounding coefficients toward 0.
```

```
void fmpz_poly_scalar_divexact_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t
    x)
    Sets poly1 to poly2 divided by the fmpz_t x, assuming the division is exact for every coefficient.
```

```
void fmpz_poly_scalar_divexact_si(fmpz_poly_t poly1, const fmpz_poly_t poly2, slong x)
    Sets poly1 to poly2 divided by the slong x, assuming the coefficient is exact for every coefficient.
```

```
void fmpz_poly_scalar_divexact_ui(fmpz_poly_t poly1, const fmpz_poly_t poly2, ulong x)
    Sets poly1 to poly2 divided by the ulong x, assuming the coefficient is exact for every coefficient.
```

```
void fmpz_poly_scalar_mod_fmpz(fmpz_poly_t poly1, const fmpz_poly_t poly2, const fmpz_t p)
    Sets poly1 to poly2, reducing each coefficient modulo  $p > 0$ .
```

void **fmpz_poly_scalar_smod_fmpz**(*fmpz_poly_t* poly1, const *fmpz_poly_t* poly2, const *fmpz_t* p)
 Sets poly1 to poly2, symmetrically reducing each coefficient modulo $p > 0$, that is, choosing the unique representative in the interval $(-p/2, p/2]$.

slong **fmpz_poly_remove_content_2exp**(*fmpz_t* *pol, *slong* len)
 Remove the 2-content of pol and return the number k that is the maximal non-negative integer so that 2^k divides all coefficients of the polynomial. For the zero polynomial, 0 is returned.

void **fmpz_poly_scale_2exp**(*fmpz_t* *pol, *slong* len, *slong* k)
 Scale (pol, len) to $p(2^k X)$ in-place and divide by the 2-content (so that the gcd of coefficients is odd). If k is negative the polynomial is multiplied by 2^{kd} .

4.7.13 Bit packing

void **fmpz_poly_bit_pack**(*nn_ptr* arr, const *fmpz_t* *poly, *slong* len, *flint_bitcnt_t* bit_size, int negate)

Packs the coefficients of poly into bitfields of the given bit_size, negating the coefficients before packing if negate is set to -1.

int **fmpz_poly_bit_unpack**(*fmpz_t* *poly, *slong* nlo, *slong* nhi, *nn_srcptr* arr, *flint_bitcnt_t* bit_size, int negate)

Unpacks the polynomial of given length nhi from the array as packed into fields of the given bit_size, finally negating the coefficients if negate is set to -1. Returns borrow, which is nonzero if a leading term with coefficient ± 1 should be added at position nhi of poly.

If nlo is zero, all coefficients are unpacked; otherwise the coefficients in the range $[nlo, nhi)$ are unpacked and written to indices $[0, nhi - nlo)$ in poly.

void **fmpz_poly_bit_unpack_unsigned**(*fmpz_t* *poly, *slong* nlo, *slong* nhi, *nn_srcptr* arr, *flint_bitcnt_t* bit_size)

Unpacks the polynomial of given length from the array as packed into fields of the given bit_size. The coefficients are assumed to be unsigned.

void **fmpz_poly_bit_pack**(*fmpz_t* f, const *fmpz_poly_t* poly, *flint_bitcnt_t* bit_size)

Packs poly into bitfields of size bit_size, writing the result to f. The sign of f will be the same as that of the leading coefficient of poly.

void **fmpz_poly_bit_unpack**(*fmpz_poly_t* poly, const *fmpz_t* f, *flint_bitcnt_t* bit_size)

Unpacks the polynomial with signed coefficients packed into fields of size bit_size as represented by the integer f.

void **fmpz_poly_bit_unpack_unsigned**(*fmpz_poly_t* poly, const *fmpz_t* f, *flint_bitcnt_t* bit_size)

Unpacks the polynomial with unsigned coefficients packed into fields of size bit_size as represented by the integer f. It is required that f is nonnegative.

4.7.14 Multiplication

void **fmpz_poly_mul_classical**(*fmpz_t* *res, const *fmpz_t* *poly1, *slong* len1, const *fmpz_t* *poly2, *slong* len2)

Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2).

Assumes len1 and len2 are positive. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

void **fmpz_poly_mul_classical**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets res to the product of poly1 and poly2, computed using the classical or schoolbook method.

```
void _fmpz_poly_mulow_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2,
                               slong len2, slong n)
```

Sets (res, n) to the first n coefficients of (poly1, len1) multiplied by (poly2, len2).

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Assumes neither len1 nor len2 is zero.

```
void fmpz_poly_mulow_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                              poly2, slong n)
```

Sets res to the first n coefficients of poly1 * poly2.

```
void _fmpz_poly_mulhigh_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2,
                                 slong len2, slong start)
```

Sets the first start coefficients of res to zero and the remainder to the corresponding coefficients of (poly1, len1) * (poly2, len2).

Assumes start $\leq \text{len1} + \text{len2} - 1$. Assumes neither len1 nor len2 is zero.

```
void fmpz_poly_mulhigh_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                                poly2, slong start)
```

Sets the first start coefficients of res to zero and the remainder to the corresponding coefficients of the product of poly1 and poly2.

```
void _fmpz_poly_mulmid_classical(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2,
                                slong len2, slong nlo, slong nhi)
```

```
void _fmpz_poly_mulmid_KS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
                          slong nlo, slong nhi)
```

```
void _fmpz_poly_mulmid_SS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
                          slong nlo, slong nhi)
```

```
void _fmpz_poly_mulmid(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
                      slong nlo, slong nhi)
```

Sets (res, nhi - nlo) to the coefficients at indices $[nlo, nhi]$ in the full product of (poly1, len1) and (poly2, len2). Assumes that len1 and len2 are positive and that $0 \leq nlo < nhi \leq \text{len1} + \text{len2} - 1$.

```
void fmpz_poly_mulmid_classical(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                                poly2)
```

```
void fmpz_poly_mulmid_KS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

```
void fmpz_poly_mulmid_SS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

```
void fmpz_poly_mulmid(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets res to the polynomial formed by the coefficients at indices $[nlo, nhi]$ in the product of poly1 and poly2. Equivalently, compute $[(poly1 \cdot poly2) \bmod x^{nhi}] / x^{nlo}$.

```
void _fmpz_poly_mul_karatsuba(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong
                              len2)
```

Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2). Assumes len1 $\geq$ len2 $>$ 0. Allows zero-padding of the two input polynomials. No aliasing of inputs with outputs is allowed.

```
void fmpz_poly_mul_karatsuba(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                             poly2)
```

Sets res to the product of poly1 and poly2.

```
void _fmpz_poly_mulow_karatsuba_n(fmpz *res, const fmpz *poly1, const fmpz *poly2, slong n)
```

Sets res to the product of poly1 and poly2 and truncates to the given length. It is assumed that poly1 and poly2 are precisely the given length, possibly zero padded. Assumes n is not zero.

```
void fmpz_poly_mulow_karatsuba_n(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                                 poly2, slong n)
```

Sets res to the product of poly1 and poly2 and truncates to the given length.

`void _fmpz_poly_mulhigh_karatsuba_n(fmpz *res, const fmpz *poly1, const fmpz *poly2, slong len)`
Sets `res` to the product of `poly1` and `poly2` and truncates at the top to the given length. The first `len - 1` coefficients are set to zero. It is assumed that `poly1` and `poly2` are precisely the given length, possibly zero padded. Assumes `len` is not zero.

`void fmpz_poly_mulhigh_karatsuba_n(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong len)`
Sets the first `len - 1` coefficients of the result to zero and the remaining coefficients to the corresponding coefficients of the product of `poly1` and `poly2`. Assumes `poly1` and `poly2` are at most of the given length.

`void _fmpz_poly_mul_KS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)`
Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`.
Places no assumptions on `len1` and `len2`. Allows zero-padding of the two input polynomials. Supports aliasing of inputs and outputs.

`void fmpz_poly_mul_KS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)`
Sets `res` to the product of `poly1` and `poly2`.

`void _fmpz_poly_mullow_KS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)`
Sets `(res, n)` to the lowest `n` coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.
Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes `n` is positive. Supports aliasing between `res`, `poly1` and `poly2`.

`void fmpz_poly_mullow_KS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)`
Sets `res` to the lowest `n` coefficients of the product of `poly1` and `poly2`.

`void _fmpz_poly_mul_SS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)`
Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`.
We must have `len1 > 1` and `len2 > 1`. Allows zero-padding of the two input polynomials. Supports aliasing of inputs and outputs.

`void fmpz_poly_mul_SS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)`
Sets `res` to the product of `poly1` and `poly2`. Uses the Schönhage-Strassen algorithm.

`void _fmpz_poly_mullow_SS(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)`
Sets `(res, n)` to the lowest `n` coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.
Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. We must have `len1 > 1` and `len2 > 1`. Assumes `n` is positive. Supports aliasing between `res`, `poly1` and `poly2`.

`void fmpz_poly_mullow_SS(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)`
Sets `res` to the lowest `n` coefficients of the product of `poly1` and `poly2`.

`void _fmpz_poly_mul(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)`
Sets `(res, len1 + len2 - 1)` to the product of `(poly1, len1)` and `(poly2, len2)`. Assumes `len1 >= len2 > 0`. Allows zero-padding of the two input polynomials. Does not support aliasing between the inputs and the output.

`void fmpz_poly_mul(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)`
Sets `res` to the product of `poly1` and `poly2`. Chooses an optimal algorithm from the choices above.

```
void _fmpz_poly_mul(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
                    slong n)
```

Sets `(res, n)` to the lowest n coefficients of the product of `(poly1, len1)` and `(poly2, len2)`.

Assumes $\text{len1} \geq \text{len2} > 0$ and $0 < n \leq \text{len1} + \text{len2} - 1$. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fmpz_poly_mul(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong
n)
```

Sets `res` to the lowest n coefficients of the product of `poly1` and `poly2`.

```
void fmpz_poly_mulhigh_n(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2,
                        slong n)
```

Sets the high n coefficients of `res` to the high n coefficients of the product of `poly1` and `poly2`, assuming the latter are precisely n coefficients in length, zero padded if necessary. The remaining $n - 1$ coefficients may be arbitrary.

```
void _fmpz_poly_mulhigh(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
                        slong start)
```

Sets all but the low n coefficients of `res` to the corresponding coefficients of the product of `poly1` of length len1 and `poly2` of length len2 , the remaining coefficients being arbitrary. It is assumed that $\text{len1} \geq \text{len2} > 0$ and that $0 < n < \text{len1} + \text{len2} - 1$. Aliasing of inputs is not permitted.

4.7.15 FFT precached multiplication

```
void fmpz_poly_mul_SS_precache_init(fmpz_poly_mul_precache_t pre, slong len1, slong bits1,
                                    const fmpz_poly_t poly2)
```

Precompute the FFT of `poly2` to enable repeated multiplication of `poly2` by polynomials whose length does not exceed len1 and whose number of bits per coefficient does not exceed bits1 .

The value bits1 may be negative, i.e. it may be the result of calling `fmpz_poly_max_bits`. The function only considers the absolute value of bits1 .

Suppose len2 is the length of `poly2` and $\text{len} = \text{len1} + \text{len2} - 1$ is the maximum output length of a polynomial multiplication using `pre`. Then internally len is rounded up to a power of two, 2^n say. The truncated FFT algorithm is used to smooth performance but note that it can only do this in the range $(2^{n-1}, 2^n]$. Therefore, it may be more efficient to recompute `pre` for cases where the output length will fall below $2^{n-1} + 1$. Otherwise the implementation will zero pad them up to that length.

Note that the Schoenhage-Strassen algorithm is only efficient for polynomials with relatively large coefficients relative to the length of the polynomials.

Also note that there are no restrictions on the polynomials. In particular the polynomial whose FFT is being precached does not have to be either longer or shorter than the polynomials it is to be multiplied by.

```
void fmpz_poly_mul_precache_clear(fmpz_poly_mul_precache_t pre)
```

Clear the space allocated by `fmpz_poly_mul_SS_precache_init`.

```
void _fmpz_poly_mullo SS_precache(fmpz *output, const fmpz *input1, slong len1,
                                   fmpz_poly_mul_precache_t pre, slong trunc)
```

Write into `output` the first `trunc` coefficients of the polynomial `(input1, len1)` by the polynomial whose FFT was precached by `fmpz_poly_mul_SS_precache_init` and stored in `pre`.

For performance reasons it is recommended that all polynomials be truncated to at most `trunc` coefficients if possible.

```
void fmpz_poly_mullo SS_precache(fmpz_poly_t res, const fmpz_poly_t poly1,
                                  fmpz_poly_mul_precache_t pre, slong n)
```

Set `res` to the product of `poly1` by the polynomial whose FFT was precached by `fmprz_poly_mul_SS_precache_init` (and stored in `pre`). The result is truncated to n coefficients (and normalised).

There are no restrictions on the length of `poly1` other than those given in the call to `fmprz_poly_mul_SS_precache_init`.

```
void fmprz_poly_mul_SS_precache(fmprz_poly_t res, const fmprz_poly_t poly1,  
                                fmprz_poly_mul_precache_t pre)
```

Set `res` to the product of `poly1` by the polynomial whose FFT was precached by `fmprz_poly_mul_SS_precache_init` (and stored in `pre`).

There are no restrictions on the length of `poly1` other than those given in the call to `fmprz_poly_mul_SS_precache_init`.

4.7.16 Squaring

```
void _fmprz_poly_sqr_KS(fmprz_t *rop, const fmprz_t *op, slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmprz_poly_sqr_KS(fmprz_poly_t rop, const fmprz_poly_t op)
```

Sets `rop` to the square of the polynomial `op` using Kronecker segmentation.

```
void _fmprz_poly_sqr_karatsuba(fmprz_t *rop, const fmprz_t *op, slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmprz_poly_sqr_karatsuba(fmprz_poly_t rop, const fmprz_poly_t op)
```

Sets `rop` to the square of the polynomial `op` using the Karatsuba multiplication algorithm.

```
void _fmprz_poly_sqr_classical(fmprz_t *rop, const fmprz_t *op, slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmprz_poly_sqr_classical(fmprz_poly_t rop, const fmprz_poly_t op)
```

Sets `rop` to the square of the polynomial `op` using the classical or schoolbook method.

```
void _fmprz_poly_sqr(fmprz_t *rop, const fmprz_t *op, slong len)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `len > 0`.

Supports zero-padding in `(op, len)`. Does not support aliasing.

```
void fmprz_poly_sqr(fmprz_poly_t rop, const fmprz_poly_t op)
```

Sets `rop` to the square of the polynomial `op`.

```
void _fmprz_poly_sqr_low_KS(fmprz_t *res, const fmprz_t *poly, slong len, slong n)
```

Sets `(res, n)` to the lowest n coefficients of the square of `(poly, len)`.

Assumes that `len` is positive, but does allow for the polynomial to be zero-padded. The polynomial may be zero, too. Assumes n is positive. Supports aliasing between `res` and `poly`.

```
void fmprz_poly_sqr_low_KS(fmprz_poly_t res, const fmprz_poly_t poly, slong n)
```

Sets `res` to the lowest n coefficients of the square of `poly`.

```
void _fmprz_poly_sqr_low_karatsuba_n(fmprz_t *res, const fmprz_t *poly, slong n)
```

Sets `(res, n)` to the square of `(poly, n)` truncated to length n , which is assumed to be positive. Allows for `poly` to be zero-padded.

void **fmpz_poly_sqrlo_karatsuba_n**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)
 Sets *res* to the square of *poly* and truncates to the given length.

void **_fmpz_poly_sqrlo_classical**(*fmpz_t* *res, const *fmpz_t* *poly, *slong* len, *slong* n)
 Sets (*res*, *n*) to the first *n* coefficients of the square of (*poly*, *len*).
 Assumes that $0 < n \leq 2 * len - 1$.

void **fmpz_poly_sqrlo_classical**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)
 Sets *res* to the first *n* coefficients of the square of *poly*.

void **_fmpz_poly_sqrlo**(*fmpz_t* *res, const *fmpz_t* *poly, *slong* len, *slong* n)
 Sets (*res*, *n*) to the lowest *n* coefficients of the square of (*poly*, *len*).
 Assumes $len1 \geq len2 > 0$ and $0 < n \leq 2 * len - 1$. Allows for zero-padding in the input.
 Does not support aliasing between the input and the output.

void **fmpz_poly_sqrlo**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)
 Sets *res* to the lowest *n* coefficients of the square of *poly*.

4.7.17 Powering

void **_fmpz_poly_pow_multinomial**(*fmpz_t* *res, const *fmpz_t* *poly, *slong* len, *ulong* e)
 Computes $res = poly^e$. This uses the J.C.P. Miller pure recurrence as follows:
 If ℓ is the index of the lowest non-zero coefficient in *poly*, as a first step this method zeros out the lowest $e\ell$ coefficients of *res*. The recurrence above is then used to compute the remaining coefficients.
 Assumes $len > 0$, $e > 0$. Does not support aliasing.

void **fmpz_poly_pow_multinomial**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *ulong* e)
 Computes $res = poly^e$ using a generalisation of binomial expansion called the J.C.P. Miller pure recurrence [1], [2]. If *e* is zero, returns one, so that in particular $0^0 = 1$.
 The formal statement of the recurrence is as follows. Write the input polynomial as $P(x) = p_0 + p_1x + \dots + p_mx^m$ with $p_0 \neq 0$ and let

$$P(x)^n = a(n, 0) + a(n, 1)x + \dots + a(n, mn)x^{mn}.$$

Then $a(n, 0) = p_0^n$ and, for all $1 \leq k \leq mn$,

$$a(n, k) = (kp_0)^{-1} \sum_{i=1}^m p_i((n+1)i - k)a(n, k - i).$$

[1] D. Knuth, The Art of Computer Programming Vol. 2, Seminumerical Algorithms, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997)

[2] D. Zeilberger, The J.C.P. Miller Recurrence for Exponentiating a Polynomial, and its q-Analog, Journal of Difference Equations and Applications, 1995, Vol. 1, pp. 57–60

void **_fmpz_poly_pow_binomial**(*fmpz_t* *res, const *fmpz_t* *poly, *ulong* e)
 Computes $res = poly^e$ when *poly* is of length 2, using binomial expansion.
 Assumes $e > 0$. Does not support aliasing.

void **fmpz_poly_pow_binomial**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *ulong* e)
 Computes $res = poly^e$ when *poly* is of length 2, using binomial expansion.
 If the length of *poly* is not 2, raises an exception and aborts.

void **_fmpz_poly_pow_addchains**(fmpz *res, const fmpz *poly, slong len, const int *a, int n)

Given a star chain $1 = a_0 < a_1 < \dots < a_n = e$ computes $\text{res} = \text{poly}^e$.

A star chain is an addition chain $1 = a_0 < a_1 < \dots < a_n$ such that, for all $i > 0$, $a_i = a_{i-1} + a_j$ for some $j < i$.

Assumes that $e > 2$, or equivalently $n > 1$, and $\text{len} > 0$. Does not support aliasing.

void **fmpz_poly_pow_addchains**(fmpz_poly_t res, const fmpz_poly_t poly, ulong e)

Computes $\text{res} = \text{poly}^e$ using addition chains whenever $0 \leq e \leq 148$.

If $e > 148$, raises an exception and aborts.

void **_fmpz_poly_pow_binexp**(fmpz *res, const fmpz *poly, slong len, ulong e)

Sets $\text{res} = \text{poly}^e$ using left-to-right binary exponentiation as described on p. 461 of [Knu1997].

Assumes that $\text{len} > 0$, $e > 1$. Assumes that res is an array of length at least $e * (\text{len} - 1) + 1$. Does not support aliasing.

void **fmpz_poly_pow_binexp**(fmpz_poly_t res, const fmpz_poly_t poly, ulong e)

Computes $\text{res} = \text{poly}^e$ using the binary exponentiation algorithm. If e is zero, returns one, so that in particular $0^0 = 1$.

void **_fmpz_poly_pow_small**(fmpz *res, const fmpz *poly, slong len, ulong e)

Sets $\text{res} = \text{poly}^e$ whenever $0 \leq e \leq 4$.

Assumes that $\text{len} > 0$ and that res is an array of length at least $e * (\text{len} - 1) + 1$. Does not support aliasing.

void **_fmpz_poly_pow**(fmpz *res, const fmpz *poly, slong len, ulong e)

Sets $\text{res} = \text{poly}^e$, assuming that e , $\text{len} > 0$ and that res has space for $e * (\text{len} - 1) + 1$ coefficients. Does not support aliasing.

void **fmpz_poly_pow**(fmpz_poly_t res, const fmpz_poly_t poly, ulong e)

Computes $\text{res} = \text{poly}^e$. If e is zero, returns one, so that in particular $0^0 = 1$.

void **_fmpz_poly_pow_trunc**(fmpz *res, const fmpz *poly, ulong e, slong n)

Sets (res, n) to (poly, n) raised to the power e and truncated to length n .

Assumes that $e, n > 0$. Allows zero-padding of (poly, n) . Does not support aliasing of any inputs and outputs.

void **fmpz_poly_pow_trunc**(fmpz_poly_t res, const fmpz_poly_t poly, ulong e, slong n)

Notationally raises poly to the power e , truncates the result to length n and writes the result in res . This is computed much more efficiently than simply powering the polynomial and truncating.

Thus, if $n = 0$ the result is zero. Otherwise, whenever $e = 0$ the result will be the constant polynomial equal to 1.

This function can be used to raise power series to a power in an efficient way.

4.7.18 Shifting

void **_fmpz_poly_shift_left**(fmpz *res, const fmpz *poly, slong len, slong n)

Sets $(\text{res}, \text{len} + n)$ to $(\text{poly}, \text{len})$ shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that len and n are positive, and that res fits $\text{len} + n$ elements. Supports aliasing between res and poly .

void **fmpz_poly_shift_left**(fmpz_poly_t res, const fmpz_poly_t poly, slong n)

Sets res to poly shifted left by n coeffs. Zero coefficients are inserted.

void **_fmpz_poly_shift_right**(*fmpz* *res, const *fmpz* *poly, *slong* len, *slong* n)

Sets (res, len - n) to (poly, len) shifted right by n coefficients.

Assumes that len and n are positive, that $\text{len} > n$, and that res fits $\text{len} - n$ elements. Supports aliasing between res and poly, although in this case the top coefficients of poly are not set to zero.

void **fmpz_poly_shift_right**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)

Sets res to poly shifted right by n coefficients. If n is equal to or greater than the current length of poly, res is set to the zero polynomial.

4.7.19 Bit sizes and norms

ulong **fmpz_poly_max_limbs**(const *fmpz_poly_t* poly)

Returns the maximum number of limbs required to store the absolute value of coefficients of poly. If poly is zero, returns 0.

slong **fmpz_poly_max_bits**(const *fmpz_poly_t* poly)

Computes the maximum number of bits b required to store the absolute value of coefficients of poly. If all the coefficients of poly are non-negative, b is returned, otherwise $-b$ is returned.

void **fmpz_poly_height**(*fmpz_t* height, const *fmpz_poly_t* poly)

Computes the height of poly, defined as the largest of the absolute values of the coefficients of poly. Equivalently, this gives the infinity norm of the coefficients. If poly is zero, the height is 0.

void **_fmpz_poly_2norm**(*fmpz_t* res, const *fmpz* *poly, *slong* len)

Sets res to the Euclidean norm of (poly, len), that is, the integer square root of the sum of the squares of the coefficients of poly.

void **fmpz_poly_2norm**(*fmpz_t* res, const *fmpz_poly_t* poly)

Sets res to the Euclidean norm of poly, that is, the integer square root of the sum of the squares of the coefficients of poly.

ulong **_fmpz_poly_2norm_normalised_bits**(const *fmpz* *poly, *slong* len)

Returns an upper bound on the number of bits of the normalised Euclidean norm of (poly, len), i.e. the number of bits of the Euclidean norm divided by the absolute value of the leading coefficient. The returned value will be no more than 1 bit too large.

This is used in the computation of the Landau-Mignotte bound.

It is assumed that $\text{len} > 0$. The result only makes sense if the leading coefficient is nonzero.

4.7.20 Greatest common divisor

void **_fmpz_poly_gcd_subresultant**(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2)

Computes the greatest common divisor (res, len2) of (poly1, len1) and (poly2, len2), assuming $\text{len1} \geq \text{len2} > 0$. The result is normalised to have positive leading coefficient. Aliasing between res, poly1 and poly2 is supported.

void **fmpz_poly_gcd_subresultant**(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Computes the greatest common divisor res of poly1 and poly2, normalised to have non-negative leading coefficient.

This function uses the subresultant algorithm as described in Algorithm 3.3.1 of [Coh1996].

```
int _fmpz_poly_gcd_heuristic(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
```

Computes the greatest common divisor (**res**, **len2**) of (**poly1**, **len1**) and (**poly2**, **len2**), assuming **len1** $\geq$ **len2** $>$ 0. The result is normalised to have positive leading coefficient. Aliasing between **res**, **poly1** and **poly2** is not supported. The function may not always succeed in finding the GCD. If it fails, the function returns 0, otherwise it returns 1.

```
int fmpz_poly_gcd_heuristic(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor **res** of **poly1** and **poly2**, normalised to have non-negative leading coefficient.

The function may not always succeed in finding the GCD. If it fails, the function returns 0, otherwise it returns 1.

This function uses the heuristic GCD algorithm (GCDHEU). The basic strategy is to remove the content of the polynomials, pack them using Kronecker segmentation (given a bound on the size of the coefficients of the GCD) and take the integer GCD. Unpack the result and test divisibility.

```
void _fmpz_poly_gcd_modular(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
```

Computes the greatest common divisor (**res**, **len2**) of (**poly1**, **len1**) and (**poly2**, **len2**), assuming **len1** $\geq$ **len2** $>$ 0. The result is normalised to have positive leading coefficient. Aliasing between **res**, **poly1** and **poly2** is not supported.

```
void fmpz_poly_gcd_modular(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor **res** of **poly1** and **poly2**, normalised to have non-negative leading coefficient.

This function uses the modular GCD algorithm. The basic strategy is to remove the content of the polynomials, reduce them modulo sufficiently many primes and do CRT reconstruction until some bound is reached (or we can prove with trial division that we have the GCD).

```
void _fmpz_poly_gcd(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
```

Computes the greatest common divisor **res** of (**poly1**, **len1**) and (**poly2**, **len2**), assuming **len1** $\geq$ **len2** $>$ 0. The result is normalised to have positive leading coefficient.

Assumes that **res** has space for **len2** coefficients. Aliasing between **res**, **poly1** and **poly2** is not supported.

```
void fmpz_poly_gcd(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the greatest common divisor **res** of **poly1** and **poly2**, normalised to have non-negative leading coefficient.

```
void _fmpz_poly_xgcd_modular(fmpz_t r, fmpz *s, fmpz *t, const fmpz *f, slong len1, const fmpz *g, slong len2)
```

Set r to the resultant of $(f, len1)$ and $(g, len2)$. If the resultant is zero, the function returns immediately. Otherwise it finds polynomials s and t such that $s*f + t*g = r$. The length of s will be no greater than **len2** and the length of t will be no greater than **len1** (both are zero padded if necessary).

It is assumed that **len1** $\geq$ **len2** $>$ 0. No aliasing of inputs and outputs is permitted.

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

Uses a multimodular algorithm. The resultant is first computed and extended GCDs modulo various primes p are computed and combined using CRT. When the CRT stabilises the resulting polynomials are simply reduced modulo further primes until a proven bound is reached.

```
void fmpz_poly_xgcd_modular(fmpz_t r, fmpz_poly_t s, fmpz_poly_t t, const fmpz_poly_t f, const fmpz_poly_t g)
```

Set r to the resultant of f and g . If the resultant is zero, the function then returns immediately, otherwise s and t are found such that $s*f + t*g = r$.

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

Uses the multimodular algorithm.

```
void _fmpz_poly_xgcd(fmpz_t r, fmpz *s, fmpz *t, const fmpz *f, slong len1, const fmpz *g, slong len2)
```

Set r to the resultant of $(f, \text{len1})$ and $(g, \text{len2})$. If the resultant is zero, the function returns immediately. Otherwise it finds polynomials s and t such that $s*f + t*g = r$. The length of s will be no greater than len2 and the length of t will be no greater than len1 (both are zero padded if necessary).

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

It is assumed that $\text{len1} \geq \text{len2} > 0$. No aliasing of inputs and outputs is permitted.

```
void fmpz_poly_xgcd(fmpz_t r, fmpz_poly_t s, fmpz_poly_t t, const fmpz_poly_t f, const fmpz_poly_t g)
```

Set r to the resultant of f and g . If the resultant is zero, the function then returns immediately, otherwise s and t are found such that $s*f + t*g = r$.

The function assumes that f and g are primitive (have Gaussian content equal to 1). The result is undefined otherwise.

```
void _fmpz_poly_lcm(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
```

Sets $(\text{res}, \text{len1} + \text{len2} - 1)$ to the least common multiple of the two polynomials $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$, normalised to have non-negative leading coefficient.

Assumes that $\text{len1} \geq \text{len2} > 0$.

Does not support aliasing.

```
void fmpz_poly_lcm(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Sets res to the least common multiple of the two polynomials poly1 and poly2 , normalised to have non-negative leading coefficient.

If either of the two polynomials is zero, sets res to zero.

This ensures that the equality

$$fg = \gcd(f, g) \text{lcm}(f, g)$$

holds up to sign.

```
void _fmpz_poly_resultant_modular(fmpz_t res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
```

Sets res to the resultant of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$, assuming that $\text{len1} \geq \text{len2} > 0$.

```
void fmpz_poly_resultant_modular(fmpz_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the resultant of poly1 and poly2 .

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

This function uses the modular algorithm described in [Col1971].

```
void fmpz_poly_resultant_modular_div(fmpz_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                                   poly2, const fmpz_t div, slong nbits)
```

Computes the resultant of `poly1` and `poly2` divided by `div` using a slight modification of the above function. It is assumed that the resultant is exactly divisible by `div` and the result `res` has at most `nbits` bits. This bypasses the computation of general bounds.

```
void _fmpz_poly_resultant_euclidean(fmpz_t res, const fmpz *poly1, slong len1, const fmpz
                                   *poly2, slong len2)
```

Sets `res` to the resultant of `(poly1, len1)` and `(poly2, len2)`, assuming that `len1 >= len2 > 0`.

```
void fmpz_poly_resultant_euclidean(fmpz_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                                   poly2)
```

Computes the resultant of `poly1` and `poly2`.

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

This function uses the algorithm described in Algorithm 3.3.7 of [Coh1996].

```
void _fmpz_poly_resultant(fmpz_t res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2)
```

Sets `res` to the resultant of `(poly1, len1)` and `(poly2, len2)`, assuming that `len1 >= len2 > 0`.

```
void fmpz_poly_resultant(fmpz_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Computes the resultant of `poly1` and `poly2`.

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

4.7.21 Discriminant

```
void _fmpz_poly_discriminant(fmpz_t res, const fmpz *poly, slong len)
```

Set `res` to the discriminant of `(poly, len)`. Assumes `len > 1`.

```
void fmpz_poly_discriminant(fmpz_t res, const fmpz_poly_t poly)
```

Set `res` to the discriminant of `poly`. We normalise the discriminant so that $\text{disc}(f) = (-1)^{n(n-1)/2} \text{res}(f, f') / \text{lc}(f)$, thus $\text{disc}(f) = \text{lc}(f)^{(2n-2)} \prod_{i < j} (r_i - r_j)^2$, where $\text{lc}(f)$ is the leading coefficient of f , n is the degree of f and r_i are the roots of f .

4.7.22 Gaussian content

```
void _fmpz_poly_content(fmpz_t res, const fmpz *poly, slong len)
```

Sets `res` to the non-negative content of `(poly, len)`. Aliasing between `res` and the coefficients of `poly` is not supported.

```
void fmpz_poly_content(fmpz_t res, const fmpz_poly_t poly)
```

Sets `res` to the non-negative content of `poly`. The content of the zero polynomial is defined to be zero. Supports aliasing, that is, `res` is allowed to be one of the coefficients of `poly`.

void **_fmpz_poly_primitive_part**(*fmpz* *res, const *fmpz* *poly, *slong* len)

Sets (res, len) to (poly, len) divided by the content of (poly, len), and normalises the result to have non-negative leading coefficient.

Assumes that (poly, len) is non-zero. Supports aliasing of res and poly.

void **fmpz_poly_primitive_part**(*fmpz_poly_t* res, const *fmpz_poly_t* poly)

Sets res to poly divided by the content of poly, and normalises the result to have non-negative leading coefficient. If poly is zero, sets res to zero.

4.7.23 Square-free

int **_fmpz_poly_is_squarefree**(const *fmpz* *poly, *slong* len)

Returns whether the polynomial (poly, len) is square-free.

int **fmpz_poly_is_squarefree**(const *fmpz_poly_t* poly)

Returns whether the polynomial poly is square-free. A non-zero polynomial is defined to be square-free if it has no non-unit square factors. We also define the zero polynomial to be square-free.

Returns 1 if the length of poly is at most 2. Returns whether the discriminant is zero for quadratic polynomials. Otherwise, returns whether the greatest common divisor of poly and its derivative has length 1.

4.7.24 Euclidean division

int **_fmpz_poly_divrem_basecase**(*fmpz* *Q, *fmpz* *R, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB, int exact)

Computes (Q, lenA - lenB + 1), (R, lenA) such that $A = BQ + R$ and each coefficient of R beyond lenB is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbb{Q}$.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in (A, lenA). R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

If the flag exact is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B. This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag exact to 0. In this case, no checks or early aborts occur and the function always returns 1.

void **fmpz_poly_divrem_basecase**(*fmpz_poly_t* Q, *fmpz_poly_t* R, const *fmpz_poly_t* A, const *fmpz_poly_t* B)

Computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B. If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbb{Q}$. An exception is raised if B is zero.

int **_fmpz_poly_divrem_divconquer_recursive**(*fmpz* *Q, *fmpz* *BQ, *fmpz* *W, const *fmpz* *A, const *fmpz* *B, *slong* lenB, int exact)

Computes (Q, lenB), (BQ, 2 lenB - 1) such that $BQ = B \times Q$ and $A = BQ + R$ where each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B. We assume that $\text{len}(A) = 2 \text{len}(B) - 1$. If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$.

Assumes $\text{len}(B) > 0$. Allows zero-padding in (A, lenA). Requires a temporary array (W, 2 lenB - 1). No aliasing of input and output operands is allowed.

This function does not read the bottom $\text{len}(B) - 1$ coefficients from A , which means that they might not even need to exist in allocated memory.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```
int _fmpz_poly_divrem_divconquer(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B,
                                slong lenB, int exact)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (R, lenA) such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . No aliasing of input and output operands is allowed.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```
void fmpz_poly_divrem_divconquer(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const
                                fmpz_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

```
int _fmpz_poly_divrem(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB, int
                    exact)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (R, lenA) such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbb{Q}$.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . No aliasing of input and output operands is allowed.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```
void fmpz_poly_divrem(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

```
int _fmpz_poly_div_basecase(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong
                          lenB, int exact)
```

Computes the quotient $(Q, \text{lenA} - \text{lenB} + 1)$ of (A, lenA) divided by (B, lenB) .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B .

If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$.

Assumes $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in $(A, \text{len}A)$. Requires a temporary array R of size at least the (actual) length of A . For convenience, R may be `NULL`. R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```
void fmpz_poly_div_basecase(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes the quotient Q of A divided by B .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B .

If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

```
int _fmpz_poly_divrem_low_divconquer_recursive(fmpz *Q, fmpz *BQ, const fmpz *A, const fmpz
                                              *B, slong lenB, int exact)
```

Divide and conquer division of $(A, 2 \text{len}B - 1)$ by $(B, \text{len}B)$, computing only the bottom $\text{len}(B) - 1$ coefficients of BQ .

Assumes $\text{len}(B) > 0$. Requires BQ to have length at least $2 \text{len}(B) - 1$, although only the bottom $\text{len}(B) - 1$ coefficients will carry meaningful output. Does not support any aliasing. Allows zero-padding in A , but not in B .

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```
int _fmpz_poly_div_divconquer_recursive(fmpz *Q, fmpz *temp, const fmpz *A, const fmpz *B,
                                       slong lenB, int exact)
```

Recursive short division in the balanced case.

Computes the quotient $(Q, \text{len}B)$ of $(A, 2 \text{len}B - 1)$ upon division by $(B, \text{len}B)$. Requires $\text{len}(B) > 0$. Needs a temporary array `temp` of length $2 \text{len}(B) - 1$. Does not support any aliasing.

For further details, see [Mul2000].

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

```
int _fmpz_poly_div_divconquer(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, int
                             exact)
```

Computes the quotient $(Q, \text{lenA} - \text{lenB} + 1)$ of (A, lenA) upon division by (B, lenB) . Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Does not support aliasing.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

void `fmpz_poly_div_divconquer`(*fmpz_poly_t* Q, const *fmpz_poly_t* A, const *fmpz_poly_t* B)

Computes the quotient Q of A divided by B .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B .

If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

int `_fmpz_poly_div`(*fmpz* *Q, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB, int exact)

Computes the quotient $(Q, \text{lenA} - \text{lenB} + 1)$ of (A, lenA) divided by (B, lenB) .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . Aliasing of input and output operands is not allowed.

If the flag `exact` is 1, the function stops if an inexact division is encountered, upon which the function will return 0. If no inexact division is encountered, the function returns 1. Note that this does not guarantee the remainder of the polynomial division is zero, merely that its length is less than that of B . This feature is useful for series division and for divisibility testing (upon testing the remainder).

For ordinary use set the flag `exact` to 0. In this case, no checks or early aborts occur and the function always returns 1.

void `fmpz_poly_div`(*fmpz_poly_t* Q, const *fmpz_poly_t* A, const *fmpz_poly_t* B)

Computes the quotient Q of A divided by B .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

void `_fmpz_poly_rem_basecase`(*fmpz* *R, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB)

Computes the remainder (R, lenA) of (A, lenA) upon division by (B, lenB) .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbb{Q}$.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

void `fmpz_poly_rem_basecase`(*fmpz_poly_t* R, const *fmpz_poly_t* A, const *fmpz_poly_t* B)

Computes the remainder R of A upon division by B .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

void **_fmpz_poly_rem**(fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB)

Computes the remainder (R, lenA) of (A, lenA) upon division by (B, lenB) .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same thing as division over $\mathbb{Q}$.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . Aliasing of input and output operands is not allowed.

void **fmpz_poly_rem**(fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B)

Computes the remainder R of A upon division by B .

Notationally, computes Q, R such that $A = BQ + R$ and each coefficient of R beyond $\text{len}(B) - 1$ is reduced modulo the leading coefficient of B . If the leading coefficient of B is ± 1 or the division is exact, this is the same as division over $\mathbb{Q}$. An exception is raised if B is zero.

void **_fmpz_poly_div_root_fmpz**(fmpz *Q, const fmpz *A, slong len, const fmpz_t c)

Computes the quotient $(Q, \text{len}-1)$ of (A, len) upon division by $x - c$.

Supports aliasing of Q and A , but the result is undefined in case of partial overlap.

void **fmpz_poly_div_root_fmpz**(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_t c)

Computes the quotient $(Q, \text{len}-1)$ of (A, len) upon division by $x - c$.

void **_fmpz_poly_divexact**(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB)

void **fmpz_poly_divexact**(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B)

Like *fmpz_poly_div()*, but assumes that the division is exact.

void **_fmpz_poly_divexact_root_fmpzq**(fmpz *Q, const fmpz *A, slong len, const fmpz_t c)

Computes the quotient $(Q, \text{len}-1)$ of (A, len) upon division by $qx - p$ where $c = p/q$. Assumes the division is exact.

Supports aliasing of Q and A , but the result is undefined in case of partial overlap.

void **fmpz_poly_divexact_root_fmpzq**(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_t c)

Computes the quotient $(Q, \text{len}-1)$ of (A, len) upon division by $qx - p$ where $c = p/q$. Assumes the division is exact.

4.7.25 Division with precomputed inverse

void **_fmpz_poly_preinvert**(fmpz *B_inv, const fmpz *B, slong n)

Given a monic polynomial B of length n , compute a precomputed inverse B_inv of length n for use in the functions below. No aliasing of B and B_inv is permitted. We assume n is not zero.

void **fmpz_poly_preinvert**(fmpz_poly_t B_inv, const fmpz_poly_t B)

Given a monic polynomial B , compute a precomputed inverse B_inv for use in the functions below. An exception is raised if B is zero.

void **_fmpz_poly_div_preinv**(fmpz *Q, const fmpz *A, slong len1, const fmpz *B, const fmpz *B_inv, slong len2)

Given a precomputed inverse B_inv of the polynomial B of length len2 , compute the quotient Q of A by B . We assume the length len1 of A is at least len2 . The polynomial Q must have space for $\text{len1} - \text{len2} + 1$ coefficients. No aliasing of operands is permitted.

void **fmpz_poly_div_preinv**(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B, const fmpz_poly_t B_inv)

Given a precomputed inverse B_inv of the polynomial B , compute the quotient Q of A by B . Aliasing of B and B_inv is not permitted.

```
void _fmpz_poly_divrem_preinv(fmpz *Q, fmpz *A, slong len1, const fmpz *B, const fmpz *B_inv,
                             slong len2)
```

Given a precomputed inverse B_inv of the polynomial B of length $len2$, compute the quotient Q of A by B . The remainder is then placed in A . We assume the length $len1$ of A is at least $len2$. The polynomial Q must have space for $len1 - len2 + 1$ coefficients. No aliasing of operands is permitted.

```
void fmpz_poly_divrem_preinv(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const
                             fmpz_poly_t B, const fmpz_poly_t B_inv)
```

Given a precomputed inverse B_inv of the polynomial B , compute the quotient Q of A by B and the remainder R . Aliasing of B and B_inv is not permitted.

```
fmpz **_fmpz_poly_powers_precompute(const fmpz *B, slong len)
```

Computes $2*len - 1$ powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

```
void fmpz_poly_powers_precompute(fmpz_poly_powers_precomp_t pinv, fmpz_poly_t poly)
```

Computes $2*len - 1$ powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

```
void _fmpz_poly_powers_clear(fmpz **powers, slong len)
```

Clean up resources used by precomputed powers which have been computed by `_fmpz_poly_powers_precompute`.

```
void fmpz_poly_powers_clear(fmpz_poly_powers_precomp_t pinv)
```

Clean up resources used by precomputed powers which have been computed by `fmpz_poly_powers_precompute`.

```
void _fmpz_poly_rem_powers_precomp(fmpz *A, slong m, const fmpz *B, slong n, fmpz **const
                                   powers)
```

Set A to the remainder of A divide B given precomputed powers mod B provided by `_fmpz_poly_powers_precompute`. No aliasing is allowed.

```
void fmpz_poly_rem_powers_precomp(fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B,
                                  const fmpz_poly_powers_precomp_t B_inv)
```

Set R to the remainder of A divide B given precomputed powers mod B provided by `fmpz_poly_powers_precompute`.

4.7.26 Divisibility testing

```
int _fmpz_poly_divides(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB)
```

Returns 1 if $(B, lenB)$ divides $(A, lenA)$ exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $len(A) \geq len(B) > 0$ and that Q has space for $len(A) - len(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fmpz_poly_divides(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

```
slong fmpz_poly_remove(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2)
```

Set res to $poly1$ divided by the highest power of $poly2$ that divides it and return the power. The divisor $poly2$ must not be zero or ± 1 , otherwise an exception is raised.

4.7.27 Division mod p

```
void fmpz_poly_divlow_smodp(fmpz *res, const fmpz_poly_t f, const fmpz_poly_t g, const fmpz_t p,
                           slong n)
```

Compute the n lowest coefficients of f divided by g , assuming the division is exact modulo p . The computed coefficients are reduced modulo p using the symmetric remainder system. We require f to be at least n in length. The function can handle trailing zeroes, but the low nonzero coefficient of g must be coprime to p . This is a bespoke function used by factoring.

```
void fmpz_poly_divhigh_smodp(fmpz *res, const fmpz_poly_t f, const fmpz_poly_t g, const fmpz_t
                             p, slong n)
```

Compute the n highest coefficients of f divided by g , assuming the division is exact modulo p . The computed coefficients are reduced modulo p using the symmetric remainder system. We require f to be as output by `fmpz_poly_mulhigh_n` given polynomials g and a polynomial of length n as inputs. The leading coefficient of g must be coprime to p . This is a bespoke function used by factoring.

4.7.28 Power series division

```
void _fmpz_poly_inv_series_basecase(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)
```

Computes the first n terms of the inverse power series of $(Q, \text{len}Q)$ using a recurrence.

Assumes that $n \geq 1$ and that Q has constant term ± 1 . Does not support aliasing.

```
void fmpz_poly_inv_series_basecase(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)
```

Computes the first n terms of the inverse power series of Q using a recurrence, assuming that Q has constant term ± 1 and $n \geq 1$.

```
void _fmpz_poly_inv_series_newton(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)
```

Computes the first n terms of the inverse power series of $(Q, \text{len}Q)$ using Newton iteration.

Assumes that $n \geq 1$ and that Q has constant term ± 1 . Does not support aliasing.

```
void fmpz_poly_inv_series_newton(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)
```

Computes the first n terms of the inverse power series of Q using Newton iteration, assuming Q has constant term ± 1 and $n \geq 1$.

```
void _fmpz_poly_inv_series(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)
```

Computes the first n terms of the inverse power series of $(Q, \text{len}Q)$.

Assumes that $n \geq 1$ and that Q has constant term ± 1 . Does not support aliasing.

```
void fmpz_poly_inv_series(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)
```

Computes the first n terms of the inverse power series of Q , assuming Q has constant term ± 1 and $n \geq 1$.

```
void _fmpz_poly_div_series_basecase(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong
                                   Blen, slong n)
```

```
void _fmpz_poly_div_series_divconquer(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong
                                      Blen, slong n)
```

```
void _fmpz_poly_div_series(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong Blen, slong
                           n)
```

Divides (A, Alen) by (B, Blen) as power series over $\mathbb{Z}$, assuming B has constant term ± 1 and $n \geq 1$. Aliasing is not supported.

```
void fmpz_poly_div_series_basecase(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B,
                                   slong n)
```

```
void fmpz_poly_div_series_divconquer(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t
                                   B, slong n)
```

```
void fmpz_poly_div_series(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B, slong n)
```

Performs power series division in $\mathbb{Z}[[x]]/(x^n)$. The function considers the polynomials A and B as power series of length n starting with the constant terms. The function assumes that B has constant term ± 1 and $n \geq 1$.

4.7.29 Pseudo division

```
void _fmpz_poly_pseudo_divrem_basecase(fmpz *Q, fmpz *R, ulong *d, const fmpz *A, slong lenA,
                                       const fmpz *B, slong lenB, const fmpz_preinvn_t inv)
```

If ℓ is the leading coefficient of B , then computes Q, R such that $\ell^d A = QB + R$. This function is used for simulating division over $\mathbb{Q}$.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that Q can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that R can fit $\text{len}(A)$ coefficients. Supports aliasing of (R, lenA) and (A, lenA) . But other than this, no aliasing of the inputs and outputs is supported.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

```
void fmpz_poly_pseudo_divrem_basecase(fmpz_poly_t Q, fmpz_poly_t R, ulong *d, const
                                       fmpz_poly_t A, const fmpz_poly_t B)
```

If ℓ is the leading coefficient of B , then computes Q, R such that $\ell^d A = QB + R$. This function is used for simulating division over $\mathbb{Q}$.

```
void _fmpz_poly_pseudo_divrem_divconquer(fmpz *Q, fmpz *R, ulong *d, const fmpz *A, slong
                                         lenA, const fmpz *B, slong lenB, const
                                         fmpz_preinvn_t inv)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1), (R, \text{lenA})$ such that $\ell^d A = BQ + R$, only setting the bottom $\text{len}(B) - 1$ coefficients of R to their correct values. The remaining top coefficients of (R, lenA) may be arbitrary.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . No aliasing of input and output operands is allowed.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

```
void fmpz_poly_pseudo_divrem_divconquer(fmpz_poly_t Q, fmpz_poly_t R, ulong *d, const
                                       fmpz_poly_t A, const fmpz_poly_t B)
```

Computes Q, R , and d such that $\ell^d A = BQ + R$, where R has length less than the length of B and ℓ is the leading coefficient of B . An exception is raised if B is zero.

```
void _fmpz_poly_pseudo_divrem_cohen(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz
                                    *B, slong lenB)
```

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that Q can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that R can fit $\text{len}(A)$ coefficients. Supports aliasing of (R, lenA) and (A, lenA) . But other than this, no aliasing of the inputs and outputs is supported.

```
void fmpz_poly_pseudo_divrem_cohen(fmpz_poly_t Q, fmpz_poly_t R, const fmpz_poly_t A, const
                                   fmpz_poly_t B)
```

This is a variant of `fmpz_poly_pseudo_divrem` which computes polynomials Q and R such that $\ell^d A = BQ + R$. However, the value of d is fixed at $\max\{0, \text{len}(A) - \text{len}(B) + 1\}$.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be close to zero. Note that this function is not asymptotically fast. It is efficient only for short polynomials, e.g. when $\text{len}(B) < 32$.

void `fmpz_poly_pseudo_rem_cohen`(*fmpz* *R, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB)
Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that R can fit $\text{len}(A)$ coefficients. Supports aliasing of (R, lenA) and (A, lenA) . But other than this, no aliasing of the inputs and outputs is supported.

void `fmpz_poly_pseudo_rem_cohen`(*fmpz_poly_t* R, const *fmpz_poly_t* A, const *fmpz_poly_t* B)
This is a variant of `fmpz_poly_pseudo_rem()` which computes polynomials Q and R such that $\ell^d A = BQ + R$, but only returns R . However, the value of d is fixed at $\max\{0, \text{len}(A) - \text{len}(B) + 1\}$.

This function is faster when the remainder is not well behaved, i.e. where it is not expected to be close to zero. Note that this function is not asymptotically fast. It is efficient only for short polynomials, e.g. when $\text{len}(B) < 32$.

This function uses the algorithm described in Algorithm 3.1.2 of [Coh1996].

void `_fmpz_poly_pseudo_divrem`(*fmpz* *Q, *fmpz* *R, *ulong* *d, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB, const *fmpz_preinvn_t* inv)

If ℓ is the leading coefficient of B , then computes $(Q, \text{lenA} - \text{lenB} + 1)$, $(R, \text{lenB} - 1)$ and d such that $\ell^d A = BQ + R$. This function is used for simulating division over $\mathbb{Q}$.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$. Assumes that Q can fit $\text{len}(A) - \text{len}(B) + 1$ coefficients, and that R can fit $\text{len}(A)$ coefficients, although on exit only the bottom $\text{len}(B)$ coefficients will carry meaningful data.

Supports aliasing of (R, lenA) and (A, lenA) . But other than this, no aliasing of the inputs and outputs is supported.

An optional precomputed inverse of the leading coefficient of B from `fmpz_preinvn_init` can be supplied. Otherwise `inv` should be `NULL`.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

void `fmpz_poly_pseudo_divrem`(*fmpz_poly_t* Q, *fmpz_poly_t* R, *ulong* *d, const *fmpz_poly_t* A, const *fmpz_poly_t* B)

Computes Q , R , and d such that $\ell^d A = BQ + R$.

void `_fmpz_poly_pseudo_div`(*fmpz* *Q, *ulong* *d, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB, const *fmpz_preinvn_t* inv)

Pseudo-division, only returning the quotient.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

void `fmpz_poly_pseudo_div`(*fmpz_poly_t* Q, *ulong* *d, const *fmpz_poly_t* A, const *fmpz_poly_t* B)
Pseudo-division, only returning the quotient.

void `_fmpz_poly_pseudo_rem`(*fmpz* *R, *ulong* *d, const *fmpz* *A, *slong* lenA, const *fmpz* *B, *slong* lenB, const *fmpz_preinvn_t* inv)

Pseudo-division, only returning the remainder.

Note: `fmpz.h` has to be included before `fmpz_poly.h` in order for `fmpz_poly.h` to declare this function.

void `fmpz_poly_pseudo_rem`(*fmpz_poly_t* R, *ulong* *d, const *fmpz_poly_t* A, const *fmpz_poly_t* B)
Pseudo-division, only returning the remainder.

4.7.30 Derivative

void **_fmpz_poly_derivative**(*fmpz* *rpoly, const *fmpz* *poly, *slong* len)
Sets (rpoly, len - 1) to the derivative of (poly, len). Also handles the cases where len is 0 or 1 correctly. Supports aliasing of rpoly and poly.

void **fmpz_poly_derivative**(*fmpz_poly_t* res, const *fmpz_poly_t* poly)
Sets res to the derivative of poly.

void **_fmpz_poly_nth_derivative**(*fmpz* *rpoly, const *fmpz* *poly, *ulong* n, *slong* len)
Sets (rpoly, len - n) to the nth derivative of (poly, len). Also handles the cases where len <= n correctly. Supports aliasing of rpoly and poly.

void **fmpz_poly_nth_derivative**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *ulong* n)
Sets res to the nth derivative of poly.

4.7.31 Evaluation

void **_fmpz_poly_evaluate_divconquer_fmpz**(*fmpz_t* res, const *fmpz* *poly, *slong* len, const *fmpz_t* a)
Evaluates the polynomial (poly, len) at the integer *a* using a divide and conquer approach. Assumes that the length of the polynomial is at least one. Allows zero padding. Does not allow aliasing between res and x.

void **fmpz_poly_evaluate_divconquer_fmpz**(*fmpz_t* res, const *fmpz_poly_t* poly, const *fmpz_t* a)
Evaluates the polynomial poly at the integer *a* using a divide and conquer approach.
Aliasing between res and a is supported, however, res may not be part of poly.

void **_fmpz_poly_evaluate_horner_fmpz**(*fmpz_t* res, const *fmpz* *f, *slong* len, const *fmpz_t* a)
Evaluates the polynomial (f, len) at the integer *a* using Horner's rule, and sets res to the result. Aliasing between res and a or any of the coefficients of f is not supported.

void **fmpz_poly_evaluate_horner_fmpz**(*fmpz_t* res, const *fmpz_poly_t* f, const *fmpz_t* a)
Evaluates the polynomial f at the integer *a* using Horner's rule, and sets res to the result.
As expected, aliasing between res and a is supported. However, res may not be aliased with a coefficient of f.

void **_fmpz_poly_evaluate_fmpz**(*fmpz_t* res, const *fmpz* *f, *slong* len, const *fmpz_t* a)
Evaluates the polynomial (f, len) at the integer *a* and sets res to the result. Aliasing between res and a or any of the coefficients of f is not supported.

void **fmpz_poly_evaluate_fmpz**(*fmpz_t* res, const *fmpz_poly_t* f, const *fmpz_t* a)
Evaluates the polynomial f at the integer *a* and sets res to the result.
As expected, aliasing between res and a is supported. However, res may not be aliased with a coefficient of f.

void **_fmpz_poly_evaluate_divconquer_fmpq**(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz* *f, *slong* len, const *fmpz_t* anum, const *fmpz_t* aden)
Evaluates the polynomial (f, len) at the rational (anum, aden) using a divide and conquer approach, and sets (rnum, rden) to the result in lowest terms. Assumes that the length of the polynomial is at least one.
Aliasing between (rnum, rden) and (anum, aden) or any of the coefficients of f is not supported.

void **fmpz_poly_evaluate_divconquer_fmpq**(*fmpq_t* res, const *fmpz_poly_t* f, const *fmpq_t* a)
Evaluates the polynomial f at the rational *a* using a divide and conquer approach, and sets res to the result.

void `_fmpz_poly_evaluate_horner_fmpz`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz* *f, *slong* len, const *fmpz_t* anum, const *fmpz_t* aden)

Evaluates the polynomial (f, len) at the rational (anum, aden) using Horner's rule, and sets (rnum, rden) to the result in lowest terms.

Aliasing between (rnum, rden) and (anum, aden) or any of the coefficients of f is not supported.

void `fmpz_poly_evaluate_horner_fmpz`(*fmpz_t* res, const *fmpz_poly_t* f, const *fmpz_t* a)

Evaluates the polynomial f at the rational a using Horner's rule, and sets res to the result.

void `_fmpz_poly_evaluate_fmpz`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz* *f, *slong* len, const *fmpz_t* anum, const *fmpz_t* aden)

Evaluates the polynomial (f, len) at the rational (anum, aden) and sets (rnum, rden) to the result in lowest terms.

Aliasing between (rnum, rden) and (anum, aden) or any of the coefficients of f is not supported.

void `fmpz_poly_evaluate_fmpz`(*fmpz_t* res, const *fmpz_poly_t* f, const *fmpz_t* a)

Evaluates the polynomial f at the rational a, and sets res to the result.

ulong `_fmpz_poly_evaluate_mod`(const *fmpz* *poly, *slong* len, *ulong* a, *ulong* n, *ulong* ninv)

Evaluates (poly, len) at the value a modulo n and returns the result. The last argument ninv must be set to the precomputed inverse of n, which can be obtained using the function `n_preinvert_limb()`.

ulong `fmpz_poly_evaluate_mod`(const *fmpz_poly_t* poly, *ulong* a, *ulong* n)

Evaluates poly at the value a modulo n and returns the result.

void `fmpz_poly_evaluate_fmpz_vec`(*fmpz* *res, const *fmpz_poly_t* f, const *fmpz* *a, *slong* n)

Evaluates f at the n values given in the vector a, writing the results to res.

double `_fmpz_poly_evaluate_horner_d`(const *fmpz* *poly, *slong* n, double d)

Evaluate (poly, n) at the double d. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in FLINT for quick and dirty evaluations of polynomials with all coefficients positive.

double `fmpz_poly_evaluate_horner_d`(const *fmpz_poly_t* poly, double d)

Evaluate poly at the double d. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in FLINT for quick and dirty evaluations of polynomials with all coefficients positive.

double `_fmpz_poly_evaluate_horner_d_2exp`(*slong* *exp, const *fmpz* *poly, *slong* n, double d)

Evaluate (poly, n) at the double d. Return the result as a double and an exponent exp combination. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in FLINT for quick and dirty evaluations of polynomials with all coefficients positive.

double `fmpz_poly_evaluate_horner_d_2exp`(*slong* *exp, const *fmpz_poly_t* poly, double d)

Evaluate poly at the double d. Return the result as a double and an exponent exp combination. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in FLINT for quick and dirty evaluations of polynomials with all coefficients positive.

double `_fmpz_poly_evaluate_horner_d_2exp2`(*slong* *exp, const *fmpz* *poly, *slong* n, double d, *slong* dexp)

Evaluate poly at $d \cdot 2^{\text{dexp}}$. Return the result as a double and an exponent exp combination. No attempt is made to do this efficiently or in a numerically stable way. It is currently only used in FLINT for quick and dirty evaluations of polynomials with all coefficients positive.

4.7.32 Newton basis

void `_fmpz_poly_monomial_to_newton`(*fmpz* *poly, const *fmpz* *roots, *slong* n)

Converts (poly, n) in-place from its coefficients given in the standard monomial basis to the Newton basis for the roots $r_0, r_1, \dots, r_{n-2}$. In other words, this determines output coefficients c_i such that $c_0 + c_1(x - r_0) + c_2(x - r_0)(x - r_1) + \dots + c_{n-1}(x - r_0)(x - r_1) \cdots (x - r_{n-2})$ is equal to the input polynomial. Uses repeated polynomial division.

void `_fmpz_poly_newton_to_monomial`(*fmpz* *poly, const *fmpz* *roots, *slong* n)

Converts (poly, n) in-place from its coefficients given in the Newton basis for the roots $r_0, r_1, \dots, r_{n-2}$ to the standard monomial basis. In other words, this evaluates $c_0 + c_1(x - r_0) + c_2(x - r_0)(x - r_1) + \dots + c_{n-1}(x - r_0)(x - r_1) \cdots (x - r_{n-2})$ where c_i are the input coefficients for poly. Uses Horner's rule.

4.7.33 Interpolation

int `_fmpz_poly_interpolate_newton`(*fmpz* *poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

int `fmpz_poly_interpolate_newton`(*fmpz_poly_t* poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

int `_fmpz_poly_interpolate_multi_mod`(*fmpz* *poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

int `fmpz_poly_interpolate_multi_mod`(*fmpz_poly_t* poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

int `_fmpz_poly_interpolate`(*fmpz* *poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

int `fmpz_poly_interpolate`(*fmpz_poly_t* poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

Sets poly to the unique interpolating polynomial of degree at most $n - 1$ satisfying $f(x_i) = y_i$ for every pair x_i, y_i in xs and ys, assuming that this polynomial has integer coefficients and that the evaluation points x_i are distinct.

If the x_i are not distinct or if the interpolating polynomial does not have integer coefficients, returns 0 indicating failure. Otherwise returns 1 indicating success.

The *newton* algorithm first interpolates in the Newton basis and then converts to the monomial basis. The *multi_mod* algorithm interpolates modulo one or more prime numbers and combines the results using CRT, using adaptive termination. The default implementation chooses automatically between the *newton* and *multi_mod* algorithms.

void `_fmpz_poly_interpolate_exact_newton`(*fmpz* *poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

void `fmpz_poly_interpolate_exact_newton`(*fmpz_poly_t* poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

void `_fmpz_poly_interpolate_exact`(*fmpz* *poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

void `fmpz_poly_interpolate_exact`(*fmpz_poly_t* poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

Like `fmpz_poly_interpolate()` etc., but optimized by omitting error handling. If the x_i are not distinct or if the interpolating polynomial does not have integer coefficients, the behavior is undefined. The default implementation chooses automatically between the *exact_newton* and *multi_mod* algorithms.

void `fmpz_poly_interpolate_fmpz_vec`(*fmpz_poly_t* poly, const *fmpz* *xs, const *fmpz* *ys, *slong* n)

Like `fmpz_poly_interpolate()`, but if the x_i values are not distinct or if the interpolating polynomial does not have integer coefficients, a `FLINT_INEXACT` exception is thrown.

4.7.34 Composition

void `_fmpz_poly_compose_horner`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2)

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for $(len1-1)*(len2-1) + 1$ coefficients. Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

void `fmpz_poly_compose_horner`(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets `res` to the composition of `poly1` and `poly2`. To be more precise, denoting `res`, `poly1`, and `poly2` by f , g , and h , sets $f(t) = g(h(t))$.

This implementation uses Horner's method.

void `_fmpz_poly_compose_divconquer`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2)

Computes the composition of `(poly1, len1)` and `(poly2, len2)` using a divide and conquer approach and places the result into `res`, assuming `res` can hold the output of length $(len1 - 1) * (len2 - 1) + 1$.

Assumes `len1, len2 > 0`. Does not support aliasing between `res` and any of `(poly1, len1)` and `(poly2, len2)`.

void `fmpz_poly_compose_divconquer`(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets `res` to the composition of `poly1` and `poly2`. To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f , g , and h , respectively, sets $f(t) = g(h(t))$.

void `_fmpz_poly_compose`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2)

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for $(len1-1)*(len2-1) + 1$ coefficients. Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

void `fmpz_poly_compose`(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets `res` to the composition of `poly1` and `poly2`. To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f , g , and h , respectively, sets $f(t) = g(h(t))$.

4.7.35 Inflation and deflation

void `fmpz_poly_inflate`(*fmpz_poly_t* result, const *fmpz_poly_t* input, *ulong* inflation)

Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `inflation`.

void `fmpz_poly_deflate`(*fmpz_poly_t* result, const *fmpz_poly_t* input, *ulong* deflation)

Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

ulong `fmpz_poly_deflation`(const *fmpz_poly_t* input)

Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 if `input` is a constant polynomial.

4.7.36 Taylor shift

`void _fmpz_poly_taylor_shift_horner(fmpz *poly, const fmpz_t c, slong n)`
Performs the Taylor shift composing *poly* by $x + c$ in-place. Uses an efficient version Horner's rule.

`void fmpz_poly_taylor_shift_horner(fmpz_poly_t g, const fmpz_poly_t f, const fmpz_t c)`
Performs the Taylor shift composing *f* by $x + c$.

`void _fmpz_poly_taylor_shift_divconquer(fmpz *poly, const fmpz_t c, slong n)`
Performs the Taylor shift composing *poly* by $x + c$ in-place. Uses the divide-and-conquer polynomial composition algorithm.

`void fmpz_poly_taylor_shift_divconquer(fmpz_poly_t g, const fmpz_poly_t f, const fmpz_t c)`
Performs the Taylor shift composing *f* by $x + c$. Uses the divide-and-conquer polynomial composition algorithm.

`void _fmpz_poly_taylor_shift_multi_mod(fmpz *poly, const fmpz_t c, slong n)`
Performs the Taylor shift composing *poly* by $x + c$ in-place. Uses a multimodular algorithm, distributing the computation across `flint_get_num_threads()` threads.

`void fmpz_poly_taylor_shift_multi_mod(fmpz_poly_t g, const fmpz_poly_t f, const fmpz_t c)`
Performs the Taylor shift composing *f* by $x + c$. Uses a multimodular algorithm, distributing the computation across `flint_get_num_threads()` threads.

`void _fmpz_poly_taylor_shift(fmpz *poly, const fmpz_t c, slong n)`
Performs the Taylor shift composing *poly* by $x + c$ in-place.

`void fmpz_poly_taylor_shift(fmpz_poly_t g, const fmpz_poly_t f, const fmpz_t c)`
Performs the Taylor shift composing *f* by $x + c$.

4.7.37 Power series composition

`void _fmpz_poly_compose_series_horner(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)`
Sets *res* to the composition of *poly1* and *poly2* modulo x^n , where the constant term of *poly2* is required to be zero.
Assumes that *len1*, *len2*, $n > 0$, that *len1*, *len2* $\leq n$, and that $(len1-1) * (len2-1) + 1 \leq n$, and that *res* has space for *n* coefficients. Does not support aliasing between any of the inputs and the output.
This implementation uses the Horner scheme.

`void fmpz_poly_compose_series_horner(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t poly2, slong n)`
Sets *res* to the composition of *poly1* and *poly2* modulo x^n , where the constant term of *poly2* is required to be zero.
This implementation uses the Horner scheme.

`void _fmpz_poly_compose_series_brent_kung(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2, slong n)`
Sets *res* to the composition of *poly1* and *poly2* modulo x^n , where the constant term of *poly2* is required to be zero.
Assumes that *len1*, *len2*, $n > 0$, that *len1*, *len2* $\leq n$, and that $(len1-1) * (len2-1) + 1 \leq n$, and that *res* has space for *n* coefficients. Does not support aliasing between any of the inputs and the output.
This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978].

```
void fmpz_poly_compose_series_brent_kung(fmpz_poly_t res, const fmpz_poly_t poly1, const
                                         fmpz_poly_t poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n , where the constant term of `poly2` is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978].

```
void _fmpz_poly_compose_series(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong
                               len2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n , where the constant term of `poly2` is required to be zero.

Assumes that `len1`, `len2`, `n` > 0, that `len1`, `len2` ≤ `n`, and that $(len1-1) * (len2-1) + 1 \leq n$, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

```
void fmpz_poly_compose_series(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_poly_t
                               poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n , where the constant term of `poly2` is required to be zero.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs.

4.7.38 Power series reversion

```
void _fmpz_poly_revert_series(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n)
```

```
void fmpz_poly_revert_series(fmpz_poly_t Qinv, const fmpz_poly_t Q, slong n)
```

Sets `Qinv` to the compositional inverse or reversion of `Q` as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$. It is required that $Q_0 = 0$ and $Q_1 = \pm 1$.

Wraps `_gr_poly_revert_series()` which chooses automatically between various algorithms.

4.7.39 Square root

```
int _fmpz_poly_sqrtrem_classical(fmpz *res, fmpz *r, const fmpz *poly, slong len)
```

Returns 1 if $(poly, len)$ can be written in the form $A^2 + R$ where $\deg(R) < \deg(poly)$, otherwise returns 0. If it can be so written, $(res, m - 1)$ is set to A and (res, m) is set to R , where $m = \deg(poly)/2 + 1$.

For efficiency reasons, `r` must have room for `len` coefficients, and may alias `poly`.

```
int fmpz_poly_sqrtrem_classical(fmpz_poly_t b, fmpz_poly_t r, const fmpz_poly_t a)
```

If a can be written as $b^2 + r$ with $\deg(r) < \deg(a)/2$, return 1 and set `b` and `r` appropriately. Otherwise return 0.

```
int _fmpz_poly_sqrtrem_divconquer(fmpz *res, fmpz *r, const fmpz *poly, slong len, fmpz *temp)
```

Returns 1 if $(poly, len)$ can be written in the form $A^2 + R$ where $\deg(R) < \deg(poly)$, otherwise returns 0. If it can be so written, $(res, m - 1)$ is set to A and (res, m) is set to R , where $m = \deg(poly)/2 + 1$.

For efficiency reasons, `r` must have room for `len` coefficients, and may alias `poly`. Temporary space of `len` coefficients is required.

```
int fmpz_poly_sqrtrem_divconquer(fmpz_poly_t b, fmpz_poly_t r, const fmpz_poly_t a)
```

If a can be written as $b^2 + r$ with $\deg(r) < \deg(a)/2$, return 1 and set `b` and `r` appropriately. Otherwise return 0.

int `_fmpz_poly_sqrt_classical`(*fmpz* *res, const *fmpz* *poly, *slong* len, int exact)

If `exact` is 1 and `(poly, len)` is a perfect square, sets `(res, len / 2 + 1)` to the square root of `poly` with positive leading coefficient and returns 1. Otherwise returns 0.

If `exact` is 0, allows a remainder after the square root, which is not computed.

This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom, requiring $O(n^2)$ coefficient operations.

int `fmpz_poly_sqrt_classical`(*fmpz_poly_t* b, const *fmpz_poly_t* a)

If `a` is a perfect square, sets `b` to the square root of `a` with positive leading coefficient and returns 1. Otherwise returns 0.

int `_fmpz_poly_sqrt_KS`(*fmpz* *res, const *fmpz* *poly, *slong* len)

Heuristic square root. If the return value is `-1`, the function failed, otherwise it succeeded and the following applies.

If `(poly, len)` is a perfect square, sets `(res, len / 2 + 1)` to the square root of `poly` with positive leading coefficient and returns 1. Otherwise returns 0.

This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom.

int `fmpz_poly_sqrt_KS`(*fmpz_poly_t* b, const *fmpz_poly_t* a)

Heuristic square root. If the return value is `-1`, the function failed, otherwise it succeeded and the following applies.

If `a` is a perfect square, sets `b` to the square root of `a` with positive leading coefficient and returns 1. Otherwise returns 0.

int `_fmpz_poly_sqrt_divconquer`(*fmpz* *res, const *fmpz* *poly, *slong* len, int exact)

If `exact` is 1 and `(poly, len)` is a perfect square, sets `(res, len / 2 + 1)` to the square root of `poly` with positive leading coefficient and returns 1. Otherwise returns 0.

If `exact` is 0, allows a remainder after the square root, which is not computed.

This function first uses various tests to detect nonsquares quickly. Then, it computes the square root iteratively from top to bottom.

int `fmpz_poly_sqrt_divconquer`(*fmpz_poly_t* b, const *fmpz_poly_t* a)

If `a` is a perfect square, sets `b` to the square root of `a` with positive leading coefficient and returns 1. Otherwise returns 0.

int `_fmpz_poly_sqrt`(*fmpz* *res, const *fmpz* *poly, *slong* len)

If `(poly, len)` is a perfect square, sets `(res, len / 2 + 1)` to the square root of `poly` with positive leading coefficient and returns 1. Otherwise returns 0.

int `fmpz_poly_sqrt`(*fmpz_poly_t* b, const *fmpz_poly_t* a)

If `a` is a perfect square, sets `b` to the square root of `a` with positive leading coefficient and returns 1. Otherwise returns 0.

int `_fmpz_poly_sqrt_series`(*fmpz* *res, const *fmpz* *poly, *slong* len, *slong* n)

Set `(res, n)` to the square root of the series `(poly, n)`, if it exists, and return 1, otherwise, return 0.

If the valuation of `poly` is not zero, `res` is zero padded to make up for the fact that the square root may not be known to precision `n`.

int `fmpz_poly_sqrt_series`(*fmpz_poly_t* b, const *fmpz_poly_t* a, *slong* n)

Set `b` to the square root of the series `a`, where the latter is taken to be a series of precision `n`. If such a square root exists, return 1, otherwise, return 0.

Note that if the valuation of `a` is not zero, `b` will not have precision `n`. It is given only to the precision to which the square root can be computed.

4.7.40 Power sums

void `_fmpz_poly_power_sums_naive`(*fmpz* *res, const *fmpz* *poly, *slong* len, *slong* n)
 Compute the (truncated) power sums series of the monic polynomial (poly, len) up to length n using Newton identities.

void `fmpz_poly_power_sums_naive`(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)
 Compute the (truncated) power sum series of the monic polynomial poly up to length n using Newton identities.

void `fmpz_poly_power_sums`(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)
 Compute the (truncated) power sums series of the monic polynomial poly up to length n . That is the power series whose coefficient of degree i is the sum of the i -th power of all (complex) roots of the polynomial poly.

void `_fmpz_poly_power_sums_to_poly`(*fmpz* *res, const *fmpz* *poly, *slong* len)
 Compute the (monic) polynomial given by its power sums series (poly, len).

void `fmpz_poly_power_sums_to_poly`(*fmpz_poly_t* res, const *fmpz_poly_t* Q)
 Compute the (monic) polynomial given its power sums series (Q).

4.7.41 Signature

void `_fmpz_poly_signature`(*slong* *r1, *slong* *r2, const *fmpz* *poly, *slong* len)
 Computes the signature (r_1, r_2) of the polynomial (poly, len). Assumes that the polynomial is squarefree over $\mathbb{Q}$.

void `fmpz_poly_signature`(*slong* *r1, *slong* *r2, const *fmpz_poly_t* poly)
 Computes the signature (r_1, r_2) of the polynomial poly, which is assumed to be square-free over $\mathbb{Q}$. The values of r_1 and $2r_2$ are the number of real and complex roots of the polynomial, respectively. For convenience, the zero polynomial is allowed, in which case the output is (0,0).

If the polynomial is not square-free, the behaviour is undefined and an exception may be raised.

This function uses the algorithm described in Algorithm 4.1.11 of [Coh1996].

4.7.42 Hensel lifting

void `fmpz_poly_hensel_build_tree`(*slong* *link, *fmpz_poly_t* *v, *fmpz_poly_t* *w, const *nmod_poly_factor_t* fac)
 Initialises and builds a Hensel tree consisting of two arrays v , w of polynomials and an array of links, called link.

The caller supplies a set of r local factors (in the factor structure fac) of some polynomial F over $\mathbb{Z}$. They also supply two arrays of initialised polynomials v and w , each of length $2r - 2$ and an array link, also of length $2r - 2$.

We will have five arrays: a v of *fmpz_poly_t*'s and a V of *nmod_poly_t*'s and also a w and a W and link. Here's the idea: we sort each leaf and node of a factor tree by degree, in fact choosing to multiply the two smallest factors, then the next two smallest (factors or products) etc. until a tree is made. The tree will be stored in the v 's. The first two elements of v will be the smallest modular factors, the last two elements of v will multiply to form F itself. Since v will be rearranging the original factors we will need to be able to recover the original order. For this we use the array link which has nonnegative even numbers and negative numbers. It is an array of *slongs* which aligns with V and v if link has a negative number in spot j that means V_j is an original modular factor which has been lifted, if link[j] is a nonnegative even number then V_j stores a product of the two entries at $V[\text{link}[j]]$ and $V[\text{link}[j]+1]$. W and w play the role of the extended GCD, at V_0, V_2, V_4 , etc. we have a new product, W_0, W_2, W_4 , etc. are the XGCD cofactors of the V 's.

For example, $V_0W_0 + V_1W_1 \equiv 1 \pmod{p^\ell}$ for some ℓ . These will be lifted along with the entries in V . It is not enough to just lift each factor, we have to lift the entire tree and the tree of XGCD cofactors.

```
void fmpz_poly_hensel_lift(fmpz_poly_t G, fmpz_poly_t H, fmpz_poly_t A, fmpz_poly_t B, const
    fmpz_poly_t f, const fmpz_poly_t g, const fmpz_poly_t h, const
    fmpz_poly_t a, const fmpz_poly_t b, const fmpz_t p, const fmpz_t p1)
```

This is the main Hensel lifting routine, which performs a Hensel step from polynomials mod p to polynomials mod $P = pp_1$. One starts with polynomials f, g, h such that $f = gh \pmod{p}$. The polynomials a, b satisfy $ag + bh = 1 \pmod{p}$.

The lifting formulae are

$$\begin{aligned} G &= \left(\left(\frac{f - gh}{p} \right) b \bmod g \right) p + g \\ H &= \left(\left(\frac{f - gh}{p} \right) a \bmod h \right) p + h \\ B &= \left(\left(\frac{1 - aG - bH}{p} \right) b \bmod g \right) p + b \\ A &= \left(\left(\frac{1 - aG - bH}{p} \right) a \bmod h \right) p + a \end{aligned}$$

Upon return we have $AG + BH = 1 \pmod{P}$ and $f = GH \pmod{P}$, where $G = g \pmod{p}$ etc.

We require that $1 < p_1 \leq p$ and that the input polynomials f, g, h have degree at least 1 and that the input polynomials a and b are non-zero.

The output arguments G, H, A, B may only be aliased with the input arguments g, h, a, b , respectively.

```
void fmpz_poly_hensel_lift_without_inverse(fmpz_poly_t Gout, fmpz_poly_t Hout, const
    fmpz_poly_t f, const fmpz_poly_t g, const
    fmpz_poly_t h, const fmpz_poly_t a, const
    fmpz_poly_t b, const fmpz_t p, const fmpz_t p1)
```

Given polynomials such that $f = gh \pmod{p}$ and $ag + bh = 1 \pmod{p}$, lifts only the factors g and h modulo $P = pp_1$.

See `fmpz_poly_hensel_lift()`.

```
void fmpz_poly_hensel_lift_only_inverse(fmpz_poly_t Aout, fmpz_poly_t Bout, const
    fmpz_poly_t G, const fmpz_poly_t H, const
    fmpz_poly_t a, const fmpz_poly_t b, const fmpz_t p,
    const fmpz_t p1)
```

Given polynomials such that $f = gh \pmod{p}$ and $ag + bh = 1 \pmod{p}$, lifts only the cofactors a and b modulo $P = pp_1$.

See `fmpz_poly_hensel_lift()`.

```
void fmpz_poly_hensel_lift_tree_recursive(slong *link, fmpz_poly_t *v, fmpz_poly_t *w,
    fmpz_poly_t f, slong j, slong inv, const fmpz_t p0,
    const fmpz_t p1)
```

Takes a current Hensel tree (`link`, `v`, `w`) and a pair $(j, j + 1)$ of entries in the tree and lifts the tree from mod p_0 to mod $P = p_0p_1$, where $1 < p_1 \leq p_0$.

Set `inv` to -1 if restarting Hensel lifting, 0 if stopping and 1 otherwise.

Here $f = gh$ is the polynomial whose factors we are trying to lift. We will have that `v[j]` is the product of `v[link[j]]` and `v[link[j] + 1]` as described above.

Does support aliasing of f with one of the polynomials in the lists v and w . But the polynomials in these two lists are not allowed to be aliases of each other.


```
void fmpz_poly_hensel_lift_tree(slong *link, fmpz_poly_t *v, fmpz_poly_t *w, fmpz_poly_t f,
                               slong r, const fmpz_t p, slong e0, slong e1, slong inv)
```

Computes $p_0 = p^{e_0}$ and $p_1 = p^{e_1 - e_0}$ for a small prime p and $P = p^{e_1}$.

If we aim to lift to p^b then f is the polynomial whose factors we wish to lift, made monic mod p^b . As usual, (link, v, w) is an initialised tree.

This starts the recursion on lifting the *product tree* for lifting from p^{e_0} to p^{e_1} . The value of `inv` corresponds to that given for the function `fmpz_poly_hensel_lift_tree_recursive()`. We set r to the number of local factors of f .

In terms of the notation, above $P = p^{e_1}$, $p_0 = p^{e_0}$ and $p_1 = p^{e_1 - e_0}$.

Assumes that f is monic.

Assumes that $1 < p_1 \leq p_0$, that is, $0 < e_1 \leq e_0$.

```
slong _fmpz_poly_hensel_start_lift(fmpz_poly_factor_t lifted_fac, slong *link, fmpz_poly_t *v,
                                  fmpz_poly_t *w, const fmpz_poly_t f, const
                                  nmod_poly_factor_t local_fac, slong N)
```

This function takes the local factors in `local_fac` and Hensel lifts them until they are known mod p^N , where $N \geq 1$.

These lifted factors will be stored (in the same ordering) in `lifted_fac`. It is assumed that `link`, `v`, and `w` are initialized arrays of `fmpz_poly_t`'s with at least $2 * r - 2$ entries and that $r \geq 2$. This is done outside of this function so that you can keep them for restarting Hensel lifting later. The product of local factors must be squarefree.

The return value is an exponent which must be passed to the function `_fmpz_poly_hensel_continue_lift()` as `prev_exp` if the Hensel lifting is to be resumed.

Currently, supports the case when $N = 1$ for convenience, although it is preferable in this case to simply iterate over the local factors and convert them to polynomials over $\mathbf{Z}$.

```
slong _fmpz_poly_hensel_continue_lift(fmpz_poly_factor_t lifted_fac, slong *link, fmpz_poly_t
                                     *v, fmpz_poly_t *w, const fmpz_poly_t f, slong prev,
                                     slong curr, slong N, const fmpz_t p)
```

This function restarts a stopped Hensel lift.

It lifts from `curr` to N . It also requires `prev` (to lift the cofactors) given as the return value of the function `_fmpz_poly_hensel_start_lift()` or the function `_fmpz_poly_hensel_continue_lift()`. The current lifted factors are supplied in `lifted_fac` and upon return are updated there. As usual `link`, `v`, and `w` describe the current Hensel tree, r is the number of local factors and p is the small prime modulo whose power we are lifting to. It is required that `curr` be at least 1 and that $N > \text{curr}$.

Currently, supports the case when `prev` and `curr` are equal.

```
void fmpz_poly_hensel_lift_once(fmpz_poly_factor_t lifted_fac, const fmpz_poly_t f, const
                               nmod_poly_factor_t local_fac, slong N)
```

This function does a Hensel lift.

It lifts local factors stored in `local_fac` of f to p^N , where $N \geq 2$. The lifted factors will be stored in `lifted_fac`. This lift cannot be restarted. This function is a convenience function intended for end users. The product of local factors must be squarefree.

4.7.43 Input and output

The functions in this section are not intended to be particularly fast. They are intended mainly as a debugging aid.

For the string output functions there are two variants. The first uses a simple string representation of polynomials which prints only the length of the polynomial and the integer coefficients, whilst the latter variant, appended with `_pretty`, uses a more traditional string representation of polynomials which prints a variable name as part of the representation.

The first string representation is given by a sequence of integers, in decimal notation, separated by white space. The first integer gives the length of the polynomial; the remaining integers are the coefficients. For example $5x^3 - x + 1$ is represented by the string "4 1 -1 0 5", and the zero polynomial is represented by "0". The coefficients may be signed and arbitrary precision.

The string representation of the functions appended by `_pretty` includes only the non-zero terms of the polynomial, starting with the one of highest degree. Each term starts with a coefficient, prepended with a sign, followed by the character `*`, followed by a variable name, which must be passed as a string parameter to the function, followed by a caret `^` followed by a non-negative exponent.

If the sign of the leading coefficient is positive, it is omitted. Also the exponents of the degree 1 and 0 terms are omitted, as is the variable and the `*` character in the case of the degree 0 coefficient. If the coefficient is plus or minus one, the coefficient is omitted, except for the sign.

Some examples of the `_pretty` representation are:

```
5*x^3+7*x-4
x^2+3
-x^4+2*x-1
x+1
5
```

int `_fmpz_poly_print`(const *fmpz* *poly, *slong* len)

Prints the polynomial (poly, len) to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int `fmpz_poly_print`(const *fmpz_poly_t* poly)

Prints the polynomial to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int `_fmpz_poly_print_pretty`(const *fmpz* *poly, *slong* len, const char *x)

Prints the pretty representation of (poly, len) to `stdout`, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int `fmpz_poly_print_pretty`(const *fmpz_poly_t* poly, const char *x)

Prints the pretty representation of poly to `stdout`, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int `_fmpz_poly_fprint`(FILE *file, const *fmpz* *poly, *slong* len)

Prints the polynomial (poly, len) to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int `fmpz_poly_fprint`(FILE *file, const *fmpz_poly_t* poly)

Prints the polynomial to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **mpz_poly_fprint_pretty**(FILE *file, const *mpz* *poly, *slong* len, const char *x)

Prints the pretty representation of (poly, len) to the stream file, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **mpz_poly_fprint_pretty**(FILE *file, const *mpz_poly_t* poly, const char *x)

Prints the pretty representation of poly to the stream file, using the string x to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **mpz_poly_read**(*mpz_poly_t* poly)

Reads a polynomial from stdin, storing the result in poly.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

int **mpz_poly_read_pretty**(*mpz_poly_t* poly, char **x)

Reads a polynomial in pretty format from stdin.

For further details, see the documentation for the function *mpz_poly_fread_pretty*.

int **mpz_poly_fread**(FILE *file, *mpz_poly_t* poly)

Reads a polynomial from the stream file, storing the result in poly.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

int **mpz_poly_fread_pretty**(FILE *file, *mpz_poly_t* poly, char **x)

Reads a polynomial from the file file and sets poly to this polynomial. The string *x is set to the variable name that is used in the input.

Returns a positive value, equal to the number of characters read from the file, in case of success.

Returns a non-positive value in case of failure, which could either be a read error or the indicator of a malformed input.

4.7.44 Modular reduction and reconstruction

void **mpz_poly_get_nmod_poly**(*nmod_poly_t* Amod, const *mpz_poly_t* A)

Sets the coefficients of Amod to the coefficients in A, reduced by the modulus of Amod.

void **mpz_poly_set_nmod_poly**(*mpz_poly_t* A, const *nmod_poly_t* Amod)

Sets the coefficients of A to the residues in Amod, normalised to the interval $-m/2 \leq r < m/2$ where m is the modulus.

void **mpz_poly_set_nmod_poly_unsigned**(*mpz_poly_t* A, const *nmod_poly_t* Amod)

Sets the coefficients of A to the residues in Amod, normalised to the interval $0 \leq r < m$ where m is the modulus.

void **mpz_poly_CRT_ui_precomp**(*mpz* *res, const *mpz* *poly1, *slong* len1, const *mpz_t* m1, *nn_srcptr* poly2, *slong* len2, *ulong* m2, *ulong* m2inv, *mpz_t* m1m2, *ulong* c, int sign)

Sets the coefficients in res to the CRT reconstruction modulo $m_1 m_2$ of the residues (poly1, len1) and (poly2, len2) which are images modulo m_1 and m_2 respectively. The caller must supply the precomputed product of the input moduli as $m_1 m_2$, the inverse of m_1 modulo m_2 as c, and the precomputed inverse of m_2 (in the form computed by *n_preinvert_limb*) as m2inv.

If sign = 0, residues $0 \leq r < m_1 m_2$ are computed, while if sign = 1, residues $-m_1 m_2 / 2 \leq r < m_1 m_2 / 2$ are computed.

Coefficients of res are written up to the maximum of len1 and len2.

```
void _fmpz_poly_CRT_ui(fmpz *res, const fmpz *poly1, slong len1, const fmpz_t m1, nn_srcptr
    poly2, slong len2, ulong m2, ulong m2inv, int sign)
```

This function is identical to `_fmpz_poly_CRT_ui_precomp`, apart from automatically computing m_1m_2 and c . It also aborts if c cannot be computed.

```
void fmpz_poly_CRT_ui(fmpz_poly_t res, const fmpz_poly_t poly1, const fmpz_t m, const
    nmod_poly_t poly2, int sign)
```

Given `poly1` with coefficients modulo `m` and `poly2` with modulus n , sets `res` to the CRT reconstruction modulo mn with coefficients satisfying $-mn/2 \leq c < mn/2$ (if `sign = 1`) or $0 \leq c < mn$ (if `sign = 0`).

4.7.45 Products

```
void _fmpz_poly_product_roots_fmpz_vec(fmpz *poly, const fmpz *xs, slong n)
```

Sets `(poly, n + 1)` to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by `xs`.

Aliasing of the input and output is not allowed.

```
void fmpz_poly_product_roots_fmpz_vec(fmpz_poly_t poly, const fmpz *xs, slong n)
```

Sets `poly` to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by `xs`.

```
void _fmpz_poly_product_roots_fmpz_vec(fmpz *poly, const fmpz *xs, slong n)
```

Sets `(poly, n + 1)` to the product of $(q_0x - p_0)(q_1x - p_1) \cdots (q_{n-1}x - p_{n-1})$, the roots p_i/q_i being given by `xs`.

```
void fmpz_poly_product_roots_fmpz_vec(fmpz_poly_t poly, const fmpz *xs, slong n)
```

Sets `poly` to the polynomial which is the product of $(q_0x - p_0)(q_1x - p_1) \cdots (q_{n-1}x - p_{n-1})$, the roots p_i/q_i being given by `xs`.

4.7.46 Subproduct trees

```
fmpz **_fmpz_poly_tree_alloc(slong len)
```

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

Entry i in the tree is a pointer to a single array of limbs, capable of storing $\lfloor n/2^i \rfloor$ subproducts of degree 2^i adjacently, plus a trailing entry if $n/2^i$ is not an integer.

For example, a tree of length 7 built from monic linear factors has the following structure, where spaces have been inserted for illustrative purposes:

```
X1 X1 X1 X1 X1 X1 X1
XX1 XX1 XX1 X1
XXXX1 XX1 X1
XXXXXXX1
```

```
void _fmpz_poly_tree_free(fmpz **tree, slong len)
```

Free the allocated space for the subproduct.

```
void _fmpz_poly_tree_build_fmpz_vec(fmpz **tree, const fmpz *roots, slong len)
```

Builds a subproduct tree in the preallocated space from the `len` monic linear factors $(q_ix - p_i)$ the roots p_i/q_i being given by `xs`. The top level product is not computed.

4.7.47 Roots

void `_fmpz_poly_bound_roots`(*fmpz_t* bound, const *fmpz* *poly, *slong* len)

void `fmpz_poly_bound_roots`(*fmpz_t* bound, const *fmpz_poly_t* poly)

Computes a nonnegative integer `bound` that bounds the absolute value of all complex roots of `poly`.
Uses Fujiwara's bound

$$2 \max \left(\left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{\frac{1}{2}}, \dots, \left| \frac{a_1}{a_n} \right|^{\frac{1}{n-1}}, \left| \frac{a_0}{2a_n} \right|^{\frac{1}{n}} \right)$$

where the coefficients of the polynomial are $a_0, \dots, a_n$.

void `_fmpz_poly_num_real_roots_sturm`(*slong* *n_neg, *slong* *n_pos, const *fmpz* *pol, *slong* len)

Sets `n_neg` and `n_pos` to the number of negative and positive roots of the polynomial (`pol`, `len`) using Sturm sequence. The Sturm sequence is computed via subresultant remainders obtained by repeated call to the function `_fmpz_poly_pseudo_rem_cohen`.

The polynomial is assumed to be squarefree, of degree larger than 1 and with non-zero constant coefficient.

slong `fmpz_poly_num_real_roots_sturm`(const *fmpz_poly_t* pol)

Returns the number of real roots of the squarefree polynomial `pol` using Sturm sequence.

The polynomial is assumed to be squarefree.

slong `_fmpz_poly_num_real_roots`(const *fmpz* *pol, *slong* len)

Returns the number of real roots of the squarefree polynomial (`pol`, `len`).

The polynomial is assumed to be squarefree.

slong `fmpz_poly_num_real_roots`(const *fmpz_poly_t* pol)

Returns the number of real roots of the squarefree polynomial `pol`.

The polynomial is assumed to be squarefree.

4.7.48 Minimal polynomials

void `_fmpz_poly_cyclotomic`(*fmpz* *a, *ulong* n, *nn_ptr* factors, *slong* num_factors, *ulong* phi)

Sets `a` to the lower half of the cyclotomic polynomial $\Phi_n(x)$, given $n \geq 3$ which must be squarefree.

A precomputed array containing the prime factors of n must be provided, as well as the value of the Euler totient function $\phi(n)$ as `phi`. If n is even, 2 must be the first factor in the list.

The degree of $\Phi_n(x)$ is exactly $\phi(n)$. Only the low $(\phi(n) + 1)/2$ coefficients are written; the high coefficients can be obtained afterwards by copying the low coefficients in reverse order, since $\Phi_n(x)$ is a palindrome for $n \neq 1$.

We use the sparse power series algorithm described as Algorithm 4 [ArnoldMonagan2011]. The algorithm is based on the identity

$$\Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)}.$$

Treating the polynomial as a power series, the multiplications and divisions can be done very cheaply using repeated additions and subtractions. The complexity is $O(2^k \phi(n))$ where k is the number of prime factors in n .

To improve efficiency for small n , we treat the `fmpz` coefficients as machine integers when there is no risk of overflow. The following bounds are given in Table 6 of [ArnoldMonagan2011]:

For $n < 10163195$, the largest coefficient in any $\Phi_n(x)$ has 27 bits, so machine arithmetic is safe on 32 bits.

For $n < 169828113$, the largest coefficient in any $\Phi_n(x)$ has 60 bits, so machine arithmetic is safe on 64 bits.

Further, the coefficients are always ± 1 or 0 if there are exactly two prime factors, so in this case machine arithmetic can be used as well.

Finally, we handle two special cases: if there is exactly one prime factor $n = p$, then $\Phi_n(x) = 1 + x + x^2 + \dots + x^{n-1}$, and if $n = 2m$, we use $\Phi_n(x) = \Phi_m(-x)$ to fall back to the case when n is odd.

void **fmpz_poly_cyclotomic**(fmpz_poly_t poly, ulong n)

Sets `poly` to the n -th cyclotomic polynomial, defined as $\Phi_n(x) = \prod_{\omega} (x - \omega)$ where ω runs over all the n -th primitive roots of unity.

We factor n into $n = qs$ where q is squarefree, and compute $\Phi_q(x)$. Then $\Phi_n(x) = \Phi_q(x^s)$.

ulong **fmpz_poly_is_cyclotomic**(const fmpz *poly, slong len)

ulong **fmpz_poly_is_cyclotomic**(const fmpz_poly_t poly)

If `poly` is a cyclotomic polynomial, returns the index n of this cyclotomic polynomial. If `poly` is not a cyclotomic polynomial, returns 0.

void **_fmpz_poly_cos_minpoly**(fmpz *coeffs, ulong n)

void **fmpz_poly_cos_minpoly**(fmpz_poly_t poly, ulong n)

Sets `poly` to the minimal polynomial of $2 \cos(2\pi/n)$. For suitable choice of n , this gives the minimal polynomial of $2 \cos(a\pi)$ or $2 \sin(a\pi)$ for any rational a .

The cosine is multiplied by a factor two since this gives a monic polynomial with integer coefficients. One can obtain the minimal polynomial for $\cos(2\pi/n)$ by making the substitution $x \rightarrow x/2$.

For $n > 2$, the degree of the polynomial is $\varphi(n)/2$. For $n = 1, 2$, the degree is 1. For $n = 0$, we define the output to be the constant polynomial 1.

See [WaktinsZeitlin1993].

void **_fmpz_poly_swinnerton_dyer**(fmpz *coeffs, ulong n)

void **fmpz_poly_swinnerton_dyer**(fmpz_poly_t poly, ulong n)

Sets `poly` to the Swinnerton-Dyer polynomial S_n , defined as the integer polynomial $S_n = \prod (x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \dots \pm \sqrt{p_n})$ where p_n denotes the n -th prime number and all combinations of signs are taken. This polynomial has degree 2^n and is irreducible over the integers (it is the minimal polynomial of $\sqrt{2} + \dots + \sqrt{p_n}$).

4.7.49 Orthogonal polynomials

void **_fmpz_poly_chebyshev_t**(fmpz *coeffs, ulong n)

void **fmpz_poly_chebyshev_t**(fmpz_poly_t poly, ulong n)

Sets `poly` to the Chebyshev polynomial of the first kind $T_n(x)$, defined by $T_n(x) = \cos(n \cos^{-1}(x))$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

void **_fmpz_poly_chebyshev_u**(fmpz *coeffs, ulong n)

void **fmpz_poly_chebyshev_u**(fmpz_poly_t poly, ulong n)

Sets `poly` to the Chebyshev polynomial of the first kind $U_n(x)$, defined by $(n+1)U_n(x) = T'_{n+1}(x)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

void **_fmpz_poly_legendre_pt**(fmpz *coeffs, ulong n)

Sets `coeffs` to the coefficient array of the shifted Legendre polynomial $\tilde{P}_n(x)$, defined by $\tilde{P}_n(x) = P_n(2x - 1)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be `n+1`. See **fmpz_poly** for the Legendre polynomials.

void **fmpz_poly_legendre_pt**(*fmpz_poly_t* poly, *ulong* n)

Sets *poly* to the shifted Legendre polynomial $\tilde{P}_n(x)$, defined by $\tilde{P}_n(x) = P_n(2x - 1)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence. See **fmpz_poly** for the Legendre polynomials.

void **_fmpz_poly_hermite_h**(*fmpz* *coeffs, *ulong* n)

Sets *coeffs* to the coefficient array of the Hermite polynomial $H_n(x)$, defined by $H'_n(x) = 2nH_{n-1}(x)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be *n*+1.

void **fmpz_poly_hermite_h**(*fmpz_poly_t* poly, *ulong* n)

Sets *poly* to the Hermite polynomial $H_n(x)$, defined by $H'_n(x) = 2nH_{n-1}(x)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

void **_fmpz_poly_hermite_he**(*fmpz* *coeffs, *ulong* n)

Sets *coeffs* to the coefficient array of the Hermite polynomial $He_n(x)$, defined by $He_n(x) = 2^{-\frac{n}{2}} H_n\left(\frac{x}{\sqrt{2}}\right)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be *n*+1.

void **fmpz_poly_hermite_he**(*fmpz_poly_t* poly, *ulong* n)

Sets *poly* to the Hermite polynomial $He_n(x)$, defined by $He_n(x) = 2^{-\frac{n}{2}} H_n\left(\frac{x}{\sqrt{2}}\right)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

4.7.50 Fibonacci polynomials

void **_fmpz_poly_fibonacci**(*fmpz* *coeffs, *ulong* n)

Sets *coeffs* to the coefficient array of the *n*-th Fibonacci polynomial. The coefficients are calculated using a hypergeometric recurrence.

void **fmpz_poly_fibonacci**(*fmpz_poly_t* poly, *ulong* n)

Sets *poly* to the *n*-th Fibonacci polynomial. The coefficients are calculated using a hypergeometric recurrence.

4.7.51 Eulerian numbers and polynomials

Eulerian numbers are the coefficients to the Eulerian polynomials

$$A_n(x) = \sum_{m=0}^n A(n, m) x^m,$$

where the Eulerian polynomials are defined by the exponential generating function

$$\frac{x-1}{x-e^{(x-1)t}} = \sum_{n=0}^{\infty} A_n(x) \frac{t^n}{n!}.$$

The Eulerian numbers can be expressed explicitly via the formula

$$A(n, m) = \sum_{k=0}^{m+1} (-1)^k \binom{n+1}{k} (m+1-k)^n.$$

Note: Not to be confused with Euler numbers and polynomials.

void **fmpz_poly_eulerian_polynomial**(*fmpz_poly_t* res, *ulong* n)

Sets *res* to the Eulerian polynomial $A_n(x)$, where we define $A_0(x) = 1$. The polynomial is calculated via a recursive relation.

4.7.52 Modular forms and q-series

void `_fmpz_poly_eta_qexp`(*fmpz* *f, *slong* r, *slong* len)

void `fmpz_poly_eta_qexp`(*fmpz_poly_t* f, *slong* r, *slong* n)

Sets f to the q -expansion to length n of the Dedekind eta function (without the leading factor $q^{1/24}$) raised to the power r , i.e. $(q^{-1/24}\eta(q))^r = \prod_{k=1}^{\infty} (1 - q^k)^r$.

In particular, $r = -1$ gives the generating function of the partition function $p(k)$, and $r = 24$ gives, after multiplication by q , the modular discriminant $\Delta(q)$ which generates the Ramanujan tau function $\tau(k)$.

This function uses sparse formulas for $r = 1, 2, 3, 4, 6$ and otherwise reduces to one of those cases using power series arithmetic.

void `_fmpz_poly_theta_qexp`(*fmpz* *f, *slong* r, *slong* len)

void `fmpz_poly_theta_qexp`(*fmpz_poly_t* f, *slong* r, *slong* n)

Sets f to the q -expansion to length n of the Jacobi theta function raised to the power r , i.e. $\vartheta(q)^r$ where $\vartheta(q) = 1 + 2 \sum_{k=1}^{\infty} q^{k^2}$.

This function uses sparse formulas for $r = 1, 2$ and otherwise reduces to those cases using power series arithmetic.

4.7.53 CLD bounds

void `fmpz_poly_CLD_bound`(*fmpz_t* res, const *fmpz_poly_t* f, *slong* n)

Compute a bound on the n coefficient of fg'/g where g is any factor of f .

4.8 fmpz_poly_mat.h – matrices of polynomials over the integers

The *fmpz_poly_mat_t* data type represents matrices whose entries are integer polynomials.

The *fmpz_poly_mat_t* type is defined as an array of *fmpz_poly_mat_struct*'s of length one. This permits passing parameters of type *fmpz_poly_mat_t* by reference.

An integer polynomial matrix internally consists of a single array of *fmpz_poly_struct*'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

4.8.1 Simple example

The following example constructs the matrix $\begin{pmatrix} 2x+1 & x \\ 1-x & -1 \end{pmatrix}$ and computes its determinant.

```
#include "fmpz_poly.h"
#include "fmpz_poly_mat.h"
int main()
{
    fmpz_poly_mat_t A;
    fmpz_poly_t P;
```

(continues on next page)

(continued from previous page)

```

fmpz_poly_mat_init(A, 2, 2);
fmpz_poly_init(P);

fmpz_poly_set_str(fmpz_poly_mat_entry(A, 0, 0), "2 1 2");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 0, 1), "2 0 1");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 1, 0), "2 1 -1");
fmpz_poly_set_str(fmpz_poly_mat_entry(A, 1, 1), "1 -1");

fmpz_poly_mat_det(P, A);
fmpz_poly_print_pretty(P, "x");

fmpz_poly_clear(P);
fmpz_poly_mat_clear(A);
}

```

The output is:

```
x^2-3*x-1
```

4.8.2 Types, macros and constants

type `fmpz_poly_mat_struct`

type `fmpz_poly_mat_t`

4.8.3 Memory management

void `fmpz_poly_mat_init`(*fmpz_poly_mat_t* mat, *slong* rows, *slong* cols)

Initialises a matrix with the given number of rows and columns for use.

void `fmpz_poly_mat_init_set`(*fmpz_poly_mat_t* mat, const *fmpz_poly_mat_t* src)

Initialises a matrix `mat` of the same dimensions as `src`, and sets it to a copy of `src`.

void `fmpz_poly_mat_clear`(*fmpz_poly_mat_t* mat)

Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

4.8.4 Basic properties

slong `fmpz_poly_mat_nrows`(const *fmpz_poly_mat_t* mat)

Returns the number of rows in `mat`.

slong `fmpz_poly_mat_ncols`(const *fmpz_poly_mat_t* mat)

Returns the number of columns in `mat`.

4.8.5 Basic assignment and manipulation

fmprz_poly_struct ***fmprz_poly_mat_entry**(const *fmprz_poly_mat_t* mat, *slong* i, *slong* j)

Gives a reference to the entry at row *i* and column *j*. The reference can be passed as an input or output variable to any **fmprz_poly** function for direct manipulation of the matrix element. No bounds checking is performed.

void **fmprz_poly_mat_set**(*fmprz_poly_mat_t* mat1, const *fmprz_poly_mat_t* mat2)

Sets *mat1* to a copy of *mat2*.

void **fmprz_poly_mat_swap**(*fmprz_poly_mat_t* mat1, *fmprz_poly_mat_t* mat2)

Swaps *mat1* and *mat2* efficiently.

void **fmprz_poly_mat_swap_entrywise**(*fmprz_poly_mat_t* mat1, *fmprz_poly_mat_t* mat2)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

4.8.6 Input and output

void **fmprz_poly_mat_print**(const *fmprz_poly_mat_t* mat, const char *x)

Prints the matrix *mat* to standard output, using the variable *x*.

4.8.7 Random matrix generation

void **fmprz_poly_mat_randtest**(*fmprz_poly_mat_t* mat, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

This is equivalent to applying **fmprz_poly_randtest** to all entries in the matrix.

void **fmprz_poly_mat_randtest_unsigned**(*fmprz_poly_mat_t* mat, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

This is equivalent to applying **fmprz_poly_randtest_unsigned** to all entries in the matrix.

void **fmprz_poly_mat_randtest_sparse**(*fmprz_poly_mat_t* A, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits, float density)

Creates a random matrix with the amount of nonzero entries given approximately by the **density** variable, which should be a fraction between 0 (most sparse) and 1 (most dense).

The nonzero entries will have random lengths between 1 and *len*.

4.8.8 Special matrices

void **fmprz_poly_mat_zero**(*fmprz_poly_mat_t* mat)

Sets *mat* to the zero matrix.

void **fmprz_poly_mat_one**(*fmprz_poly_mat_t* mat)

Sets *mat* to the unit or identity matrix of given shape, having the element 1 on the main diagonal and zeros elsewhere. If *mat* is nonsquare, it is set to the truncation of a unit matrix.

4.8.9 Basic comparison and properties

`int fmpz_poly_mat_equal(const fmpz_poly_mat_t mat1, const fmpz_poly_mat_t mat2)`
 Returns nonzero if `mat1` and `mat2` have the same shape and all their entries agree, and returns zero otherwise.

`int fmpz_poly_mat_is_zero(const fmpz_poly_mat_t mat)`
 Returns nonzero if all entries in `mat` are zero, and returns zero otherwise.

`int fmpz_poly_mat_is_one(const fmpz_poly_mat_t mat)`
 Returns nonzero if all entries of `mat` on the main diagonal are the constant polynomial 1 and all remaining entries are zero, and returns zero otherwise. The matrix need not be square.

`int fmpz_poly_mat_is_empty(const fmpz_poly_mat_t mat)`
 Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

`int fmpz_poly_mat_is_square(const fmpz_poly_mat_t mat)`
 Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

4.8.10 Norms

`slong fmpz_poly_mat_max_bits(const fmpz_poly_mat_t A)`
 Returns the maximum number of bits among the coefficients of the entries in `A`, or the negative of that value if any coefficient is negative.

`slong fmpz_poly_mat_max_length(const fmpz_poly_mat_t A)`
 Returns the maximum polynomial length among all the entries in `A`.

4.8.11 Transpose

`void fmpz_poly_mat_transpose(fmpz_poly_mat_t B, const fmpz_poly_mat_t A)`
 Sets `B` to A^t . The operands must have compatible dimensions. Aliasing is allowed for square matrices.

4.8.12 Evaluation

`void fmpz_poly_mat_evaluate_fmpz(fmpz_mat_t B, const fmpz_poly_mat_t A, const fmpz_t x)`
 Sets the `fmpz_mat_t B` to `A` evaluated entrywise at the point `x`.

4.8.13 Arithmetic

`void fmpz_poly_mat_scalar_mul_fmpz_poly(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, const fmpz_poly_t c)`
 Sets `B` to `A` multiplied entrywise by the polynomial `c`.

`void fmpz_poly_mat_scalar_mul_fmpz(fmpz_poly_mat_t B, const fmpz_poly_mat_t A, const fmpz_t c)`
 Sets `B` to `A` multiplied entrywise by the integer `c`.

`void fmpz_poly_mat_add(fmpz_poly_mat_t C, const fmpz_poly_mat_t A, const fmpz_poly_mat_t B)`
 Sets `C` to the sum of `A` and `B`. All matrices must have the same shape. Aliasing is allowed.

void **fmpz_poly_mat_sub**(*fmpz_poly_mat_t* C, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B)

Sets C to the sum of A and B. All matrices must have the same shape. Aliasing is allowed.

void **fmpz_poly_mat_neg**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A)

Sets B to the negation of A. The matrices must have the same shape. Aliasing is allowed.

void **fmpz_poly_mat_mul**(*fmpz_poly_mat_t* C, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B)

Sets C to the matrix product of A and B. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

void **fmpz_poly_mat_mul_classical**(*fmpz_poly_mat_t* C, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B)

Sets C to the matrix product of A and B, computed using the classical algorithm. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void **fmpz_poly_mat_mul_KS**(*fmpz_poly_mat_t* C, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B)

Sets C to the matrix product of A and B, computed using Kronecker segmentation. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void **fmpz_poly_mat_mullo**(*fmpz_poly_mat_t* C, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B, *slong* len)

Sets C to the matrix product of A and B, truncating each entry in the result to length *len*. Uses classical matrix multiplication. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void **fmpz_poly_mat_sqr**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function automatically chooses between classical and KS squaring.

void **fmpz_poly_mat_sqr_classical**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

void **fmpz_poly_mat_sqr_KS**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses Kronecker segmentation.

void **fmpz_poly_mat_sqrlo**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A, *slong* len)

Sets B to the square of A, which must be a square matrix, truncating all entries to length *len*. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

void **fmpz_poly_mat_pow**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A, *ulong* exp)

Sets B to A raised to the power *exp*, where A is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

void **fmpz_poly_mat_pow_trunc**(*fmpz_poly_mat_t* B, const *fmpz_poly_mat_t* A, *ulong* exp, *slong* len)

Sets B to A raised to the power *exp*, truncating all entries to length *len*, where A is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

void **fmpz_poly_mat_prod**(*fmpz_poly_mat_t* res, *fmpz_poly_mat_t* *const factors, *slong* n)

Sets *res* to the product of the *n* matrices given in the vector *factors*, all of which must be square and of the same size. Uses binary splitting.

4.8.14 Row reduction

long **fmpz_poly_mat_find_pivot_any**(const *fmpz_poly_mat_t* mat, *long* start_row, *long* end_row, *long* c)

Attempts to find a pivot entry for row reduction. Returns a row index r between **start_row** (inclusive) and **stop_row** (exclusive) such that column c in **mat** has a nonzero entry on row r , or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

long **fmpz_poly_mat_find_pivot_partial**(const *fmpz_poly_mat_t* mat, *long* start_row, *long* end_row, *long* c)

Attempts to find a pivot entry for row reduction. Returns a row index r between **start_row** (inclusive) and **stop_row** (exclusive) such that column c in **mat** has a nonzero entry on row r , or returns -1 if no such entry exists.

This implementation searches all the rows in the column and chooses the nonzero entry of smallest degree. If there are several entries with the same minimal degree, it chooses the entry with the smallest coefficient bit bound. This heuristic typically reduces coefficient growth when the matrix entries vary in size.

long **fmpz_poly_mat_fflu**(*fmpz_poly_mat_t* B, *fmpz_poly_t* den, *long* *perm, const *fmpz_poly_mat_t* A, int rank_check)

Uses fraction-free Gaussian elimination to set (B, den) to a fraction-free LU decomposition of A and returns the rank of A. Aliasing of A and B is allowed.

Pivot elements are chosen with **fmpz_poly_mat_find_pivot_partial**. If **perm** is non-NULL, the permutation of rows in the matrix will also be applied to **perm**.

If **rank_check** is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator **den** is set to $\pm \det(A)$, where the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

long **fmpz_poly_mat_rref**(*fmpz_poly_mat_t* B, *fmpz_poly_t* den, const *fmpz_poly_mat_t* A)

Sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.

The denominator **den** is set to $\pm \det(A)$. Note that the determinant is not generally the minimal denominator.

4.8.15 Trace

void **fmpz_poly_mat_trace**(*fmpz_poly_t* trace, const *fmpz_poly_mat_t* mat)

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

4.8.16 Determinant and rank

void **fmpz_poly_mat_det**(*fmpz_poly_t* det, const *fmpz_poly_mat_t* A)

Sets *det* to the determinant of the square matrix *A*. Uses a direct formula, fraction-free LU decomposition, or interpolation, depending on the size of the matrix.

void **fmpz_poly_mat_det_fflu**(*fmpz_poly_t* det, const *fmpz_poly_mat_t* A)

Sets *det* to the determinant of the square matrix *A*. The determinant is computed by performing a fraction-free LU decomposition on a copy of *A*.

void **fmpz_poly_mat_det_interpolate**(*fmpz_poly_t* det, const *fmpz_poly_mat_t* A)

Sets *det* to the determinant of the square matrix *A*. The determinant is computed by determining a bound n for its length, evaluating the matrix at n distinct points, computing the determinant of each integer matrix, and forming the interpolating polynomial.

slong **fmpz_poly_mat_rank**(const *fmpz_poly_mat_t* A)

Returns the rank of *A*. Performs fraction-free LU decomposition on a copy of *A*.

4.8.17 Inverse

int **fmpz_poly_mat_inv**(*fmpz_poly_mat_t* Ainv, *fmpz_poly_t* den, const *fmpz_poly_mat_t* A)

Sets (Ainv, den) to the inverse matrix of *A*. Returns 1 if *A* is nonsingular and 0 if *A* is singular. Aliasing of Ainv and *A* is allowed.

More precisely, *det* will be set to the determinant of *A* and Ainv will be set to the adjugate matrix of *A*. Note that the determinant is not necessarily the minimal denominator.

Uses fraction-free LU decomposition, followed by solving for the identity matrix.

4.8.18 Nullspace

slong **fmpz_poly_mat_nullspace**(*fmpz_poly_mat_t* res, const *fmpz_poly_mat_t* mat)

Computes the right rational nullspace of the matrix *mat* and returns the nullity.

More precisely, assume that *mat* has rank r and nullity n . Then this function sets the first n columns of *res* to linearly independent vectors spanning the nullspace of *mat*. As a result, we always have $\text{rank}(\text{res}) = n$, and $\text{mat} \times \text{res}$ is the zero matrix.

The computed basis vectors will not generally be in a reduced form. In general, the polynomials in each column vector in the result will have a nontrivial common GCD.

4.8.19 Solving

int **fmpz_poly_mat_solve**(*fmpz_poly_mat_t* X, *fmpz_poly_t* den, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B)

Solves the equation $AX = B$ for nonsingular *A*. More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if *A* is nonsingular and 0 if *A* is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

int **fmpz_poly_mat_solve_fflu**(*fmpz_poly_mat_t* X, *fmpz_poly_t* den, const *fmpz_poly_mat_t* A, const *fmpz_poly_mat_t* B)

Solves the equation $AX = B$ for nonsingular *A*. More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if *A* is nonsingular and 0 if *A* is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.


```
void fmpz_poly_mat_solve_fflu_precomp(fmpz_poly_mat_t X, const slong *perm, const
                                     fmpz_poly_mat_t FFLU, const fmpz_poly_mat_t B)
```

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

4.9 fmpz_poly_factor.h – factorisation of polynomials over the integers

4.9.1 Types, macros and constants

```
type fmpz_poly_factor_struct
```

```
type fmpz_poly_factor_t
```

4.9.2 Memory management

```
void fmpz_poly_factor_init(fmpz_poly_factor_t fac)
```

Initialises a new factor structure.

```
void fmpz_poly_factor_init2(fmpz_poly_factor_t fac, slong alloc)
```

Initialises a new factor structure, providing space for at least `alloc` factors.

```
void fmpz_poly_factor_realloc(fmpz_poly_factor_t fac, slong alloc)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fmpz_poly_factor_fit_length(fmpz_poly_factor_t fac, slong len)
```

Ensures that the factor structure has space for at least `len` factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

```
void fmpz_poly_factor_clear(fmpz_poly_factor_t fac)
```

Releases all memory occupied by the factor structure.

4.9.3 Manipulating factors

```
void fmpz_poly_factor_set(fmpz_poly_factor_t res, const fmpz_poly_factor_t fac)
```

Sets `res` to the same factorisation as `fac`.

```
void fmpz_poly_factor_insert(fmpz_poly_factor_t fac, const fmpz_poly_t p, slong e)
```

Adds the primitive polynomial p^e to the factorisation `fac`.

Assumes that $\deg(p) \geq 2$ and $e \neq 0$.

```
void fmpz_poly_factor_concat(fmpz_poly_factor_t res, const fmpz_poly_factor_t fac)
```

Concatenates two factorisations.

This is equivalent to calling `fmpz_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

4.9.4 Input and output

void **fmpz_poly_factor_print**(const *fmpz_poly_factor_t* fac)
 Prints the entries of **fac** to standard output.

4.9.5 Factoring algorithms

void **fmpz_poly_factor_squarefree**(*fmpz_poly_factor_t* fac, const *fmpz_poly_t* F)
 Takes as input a polynomial F and a freshly initialized factor structure **fac**. Updates **fac** to contain a factorization of F into (not necessarily irreducible) factors that themselves have no repeated factors. None of the returned factors will have the same exponent. That is we return g_i and unique e_i such that

$$F = c \prod_i g_i^{e_i}$$

where c is the signed content of F and $\gcd(g_i, g'_i) = 1$.

void **fmpz_poly_factor_zassenhaus_recombination**(*fmpz_poly_factor_t* final_fac, const *fmpz_poly_factor_t* lifted_fac, const *fmpz_poly_t* F, const *fmpz_t* P, *slong* exp)

Takes as input a factor structure **lifted_fac** containing a squarefree factorization of the polynomial $F \bmod p$. The algorithm does a brute force search for irreducible factors of F over the integers, and each factor is raised to the power **exp**.

The impact of the algorithm is to augment a factorization of F^{exp} to the factor structure **final_fac**.

void **_fmpz_poly_factor_zassenhaus**(*fmpz_poly_factor_t* final_fac, *slong* exp, const *fmpz_poly_t* f, *slong* cutoff, int use_van_hoeij)

This is the internal wrapper of Zassenhaus.

It will attempt to find a small prime such that f modulo p has a minimal number of factors. If it cannot find a prime giving less than **cutoff** factors it aborts. Then it decides a p -adic precision to lift the factors to, Hensel lifts, and finally calls Zassenhaus recombination.

Assumes that $\text{len}(f) \geq 2$.

Assumes that f is primitive.

Assumes that the constant coefficient of f is non-zero. Note that this can be easily achieved by taking out factors of the form x^k before calling this routine.

If the final flag is set, the function will use the van Hoeij factorisation algorithm with gradual feeding and mod 2^k data truncation to find factors when the number of local factors is large.

void **fmpz_poly_factor_zassenhaus**(*fmpz_poly_factor_t* final_fac, const *fmpz_poly_t* F)

A wrapper of the Zassenhaus factoring algorithm, which takes as input any polynomial F , and stores a factorization in **final_fac**.

The complexity will be exponential in the number of local factors we find for the components of a squarefree factorization of F .

void **_fmpz_poly_factor_quadratic**(*fmpz_poly_factor_t* fac, const *fmpz_poly_t* f, *slong* exp)

void **_fmpz_poly_factor_cubic**(*fmpz_poly_factor_t* fac, const *fmpz_poly_t* f, *slong* exp)

Inserts the factorisation of the quadratic (resp. cubic) polynomial f into **fac** with multiplicity **exp**. This function requires that the content of f has been removed, and does not update the content of **fac**. The factorization is calculated over $\mathbb{R}$ or $\mathbb{Q}_2$ and then tested over $\mathbb{Z}$.

void **fmpz_poly_factor**(*fmpz_poly_factor_t* final_fac, const *fmpz_poly_t* F)

A wrapper of the Zassenhaus and van Hoeij factoring algorithms, which takes as input any polynomial F , and stores a factorization in **final_fac**.

4.10 fmpz_mpoly.h – multivariate polynomials over the integers

The exponents follow the `mpoly` interface. A coefficient may be referenced as a `fmpz *`.

4.10.1 Types, macros and constants

type `fmpz_mpoly_struct`

A structure holding a multivariate integer polynomial.

type `fmpz_mpoly_t`

An array of length 1 of `fmpz_mpoly_struct`.

type `fmpz_mpoly_ctx_struct`

Context structure representing the parent ring of an `fmpz_mpoly`.

type `fmpz_mpoly_ctx_t`

An array of length 1 of `fmpz_mpoly_ctx_struct`.

4.10.2 Context object

void `fmpz_mpoly_ctx_init(fmpz_mpoly_ctx_t ctx, slong nvars, const ordering_t ord)`

Initialise a context object for a polynomial ring with the given number of variables and the given ordering. The possibilities for the ordering are `ORD_LEX`, `ORD_DEGLEX` and `ORD_DEGREVLEX`.

slong `fmpz_mpoly_ctx_nvars(const fmpz_mpoly_ctx_t ctx)`

Return the number of variables used to initialize the context.

ordering_t `fmpz_mpoly_ctx_ord(const fmpz_mpoly_ctx_t ctx)`

Return the ordering used to initialize the context.

void `fmpz_mpoly_ctx_clear(fmpz_mpoly_ctx_t ctx)`

Release up any space allocated by *ctx*.

4.10.3 Memory management

void `fmpz_mpoly_init(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)`

Initialise *A* for use with the given and initialised context object. Its value is set to zero.

void `fmpz_mpoly_init2(fmpz_mpoly_t A, slong alloc, const fmpz_mpoly_ctx_t ctx)`

Initialise *A* for use with the given and initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and at least `MPOLY_MIN_BITS` bits for the exponents.

void `fmpz_mpoly_init3(fmpz_mpoly_t A, slong alloc, flint_bitcnt_t bits, const fmpz_mpoly_ctx_t ctx)`

Initialise *A* for use with the given and initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and *bits* bits for the exponents.

void `fmpz_mpoly_fit_length(fmpz_mpoly_t A, slong len, const fmpz_mpoly_ctx_t ctx)`

Ensure that *A* has space for at least *len* terms.

void `fmpz_mpoly_fit_bits(fmpz_mpoly_t A, flint_bitcnt_t bits, const fmpz_mpoly_ctx_t ctx)`

Ensure that the exponent fields of *A* have at least *bits* bits.

void `fmpz_mpoly_realloc(fmpz_mpoly_t A, slong alloc, const fmpz_mpoly_ctx_t ctx)`

Reallocate *A* to have space for *alloc* terms. Assumes the current length of the polynomial is not greater than *alloc*.

```
void fmpz_mpoly_clear(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
```

Release any space allocated for A .

4.10.4 Input/Output

The variable strings in x start with the variable of most significance at index 0. If x is NULL, the variables are named x_1 , x_2 , etc.

```
char *fmpz_mpoly_get_str_pretty(const fmpz_mpoly_t A, const char **x, const fmpz_mpoly_ctx_t ctx)
```

Return a string, which the user is responsible for cleaning up, representing A , given an array of variable strings x .

```
int fmpz_mpoly_fprint_pretty(FILE *file, const fmpz_mpoly_t A, const char **x, const fmpz_mpoly_ctx_t ctx)
```

Print a string representing A to *file*.

```
int fmpz_mpoly_print_pretty(const fmpz_mpoly_t A, const char **x, const fmpz_mpoly_ctx_t ctx)
```

Print a string representing A to *stdout*.

```
int fmpz_mpoly_set_str_pretty(fmpz_mpoly_t A, const char *str, const char **x, const fmpz_mpoly_ctx_t ctx)
```

Set A to the polynomial in the null-terminates string *str* given an array x of variable strings. If parsing *str* fails, A is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in x . The character $\wedge$ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

4.10.5 Basic manipulation

```
void fmpz_mpoly_gen(fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)
```

Set A to the variable of index *var*, where *var* = 0 corresponds to the variable with the most significance with respect to the ordering.

```
int fmpz_mpoly_is_gen(const fmpz_mpoly_t A, slong var, const fmpz_mpoly_ctx_t ctx)
```

If *var* ≥ 0 , return 1 if A is equal to the *var*-th generator, otherwise return 0. If *var* < 0 , return 1 if the polynomial is equal to any generator, otherwise return 0.

```
void fmpz_mpoly_set(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

Set A to B .

```
int fmpz_mpoly_equal(const fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

Return 1 if A is equal to B , else return 0.

```
void fmpz_mpoly_swap(fmpz_mpoly_t poly1, fmpz_mpoly_t poly2, const fmpz_mpoly_ctx_t ctx)
```

Efficiently swap A and B .

```
int _fmpz_mpoly_fits_small(const fmpz *poly, slong len)
```

Return 1 if the array of coefficients of length *len* consists entirely of values that are small *fmpz* values, i.e. of at most $\text{FLINT\_BITS} - 2$ bits plus a sign bit.

```
slong fmpz_mpoly_max_bits(const fmpz_mpoly_t A)
```

Computes the maximum number of bits b required to represent the absolute values of the coefficients of A . If all of the coefficients are positive, b is returned, otherwise $-b$ is returned.

4.10.6 Constants

int `fmpz_mpoly_is_fmpz`(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if *A* is a constant, else return 0.

void `fmpz_mpoly_get_fmpz`(*fmpz_t* c, const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Assuming that *A* is a constant, set *c* to this constant. This function throws if *A* is not a constant.

void `fmpz_mpoly_set_fmpz`(*fmpz_mpoly_t* A, const *fmpz_t* c, const *fmpz_mpoly_ctx_t* ctx)

void `fmpz_mpoly_set_ui`(*fmpz_mpoly_t* A, *ulong* c, const *fmpz_mpoly_ctx_t* ctx)

void `fmpz_mpoly_set_si`(*fmpz_mpoly_t* A, *slong* c, const *fmpz_mpoly_ctx_t* ctx)
 Set *A* to the constant *c*.

void `fmpz_mpoly_zero`(*fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Set *A* to the constant 0.

void `fmpz_mpoly_one`(*fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Set *A* to the constant 1.

int `fmpz_mpoly_equal_fmpz`(const *fmpz_mpoly_t* A, const *fmpz_t* c, const *fmpz_mpoly_ctx_t* ctx)

int `fmpz_mpoly_equal_ui`(const *fmpz_mpoly_t* A, *ulong* c, const *fmpz_mpoly_ctx_t* ctx)

int `fmpz_mpoly_equal_si`(const *fmpz_mpoly_t* A, *slong* c, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if *A* is equal to the constant *c*, else return 0.

int `fmpz_mpoly_is_zero`(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if *A* is the constant 0, else return 0.

int `fmpz_mpoly_is_one`(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if *A* is the constant 1, else return 0.

4.10.7 Degrees

int `fmpz_mpoly_degrees_fit_si`(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if the degrees of *A* with respect to each variable fit into an *slong*, otherwise return 0.

void `fmpz_mpoly_degrees_fmpz`(*fmpz_t* *degs, const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)

void `fmpz_mpoly_degrees_si`(*slong* *degs, const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Set *degs* to the degrees of *A* with respect to each variable. If *A* is zero, all degrees are set to -1 .

void `fmpz_mpoly_degree_fmpz`(*fmpz_t* deg, const *fmpz_mpoly_t* A, *slong* var, const *fmpz_mpoly_ctx_t* ctx)

slong `fmpz_mpoly_degree_si`(const *fmpz_mpoly_t* A, *slong* var, const *fmpz_mpoly_ctx_t* ctx)
 Either return or set *deg* to the degree of *A* with respect to the variable of index *var*. If *A* is zero, the degree is defined to be -1 .

int `fmpz_mpoly_total_degree_fits_si`(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if the total degree of *A* fits into an *slong*, otherwise return 0.

void `fmpz_mpoly_total_degree_fmpz`(*fmpz_t* tdeg, const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)

slong `fmpz_mpoly_total_degree_si`(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Either return or set *tdeg* to the total degree of *A*. If *A* is zero, the total degree is defined to be -1 .

void `fmpz_mpoly_used_vars`(int *used, const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 For each variable index *i*, set *used*[*i*] to nonzero if the variable of index *i* appears in *A* and to zero otherwise.

4.10.8 Coefficients

```
void fmpz_mpoly_get_coeff_fmpz_monomial(fmpz_t c, const fmpz_mpoly_t A, const fmpz_mpoly_t
                                         M, const fmpz_mpoly_ctx_t ctx)
```

Assuming that M is a monomial, set c to the coefficient of the corresponding monomial in A . This function throws if M is not a monomial.

```
void fmpz_mpoly_set_coeff_fmpz_monomial(fmpz_mpoly_t poly, const fmpz_t c, const
                                         fmpz_mpoly_t poly2, const fmpz_mpoly_ctx_t ctx)
```

Assuming that M is a monomial, set the coefficient of the corresponding monomial in A to c . This function throws if M is not a monomial.

```
void fmpz_mpoly_get_coeff_fmpz_fmpz(fmpz_t c, const fmpz_mpoly_t A, fmpz *const *exp, const
                                     fmpz_mpoly_ctx_t ctx)
```

```
ulong fmpz_mpoly_get_coeff_ui_fmpz(const fmpz_mpoly_t A, fmpz *const *exp, const
                                   fmpz_mpoly_ctx_t ctx)
```

```
slong fmpz_mpoly_get_coeff_si_fmpz(const fmpz_mpoly_t A, fmpz *const *exp, const
                                   fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_get_coeff_fmpz_ui(fmpz_t c, const fmpz_mpoly_t A, const ulong *exp, const
                                  fmpz_mpoly_ctx_t ctx)
```

```
ulong fmpz_mpoly_get_coeff_ui_ui(const fmpz_mpoly_t A, const ulong *exp, const
                                  fmpz_mpoly_ctx_t ctx)
```

```
slong fmpz_mpoly_get_coeff_si_ui(const fmpz_mpoly_t A, const ulong *exp, const
                                  fmpz_mpoly_ctx_t ctx)
```

Either return or set c to the coefficient of the monomial with exponent vector exp .

```
void fmpz_mpoly_set_coeff_fmpz_fmpz(fmpz_mpoly_t A, const fmpz_t c, fmpz *const *exp, const
                                     fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_set_coeff_ui_fmpz(fmpz_mpoly_t A, ulong c, fmpz *const *exp, const
                                   fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_set_coeff_si_fmpz(fmpz_mpoly_t A, slong c, fmpz *const *exp, const
                                   fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_set_coeff_fmpz_ui(fmpz_mpoly_t A, const fmpz_t c, const ulong *exp, const
                                  fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_set_coeff_ui_ui(fmpz_mpoly_t A, ulong c, const ulong *exp, const
                                  fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_set_coeff_si_ui(fmpz_mpoly_t A, slong c, const ulong *exp, const
                                  fmpz_mpoly_ctx_t ctx)
```

Set the coefficient of the monomial with exponent vector exp to c .

```
void fmpz_mpoly_get_coeff_vars_ui(fmpz_mpoly_t C, const fmpz_mpoly_t A, const slong *vars,
                                  const ulong *exps, slong length, const fmpz_mpoly_ctx_t ctx)
```

Set C to the coefficient of A with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \leq nvars(A)$ and that the variables in $vars$ are distinct.

4.10.9 Comparison

int **fmpz_mpoly_cmp**(const *fmpz_mpoly_t* A, const *fmpz_mpoly_t* B, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 (resp. -1, or 0) if *A* is after (resp. before, same as) *B* in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when *A* and *B* are both monomials.

4.10.10 Conversion

int **fmpz_mpoly_is_fmpz_poly**(const *fmpz_mpoly_t* A, *slong* var, const *fmpz_mpoly_ctx_t* ctx)
 Return whether *A* is a univariate polynomial in the variable with index *var*.

int **fmpz_mpoly_get_fmpz_poly**(*fmpz_poly_t* A, const *fmpz_mpoly_t* B, *slong* var, const *fmpz_mpoly_ctx_t* ctx)
 If *B* is a univariate polynomial in the variable with index *var*, set *A* to this polynomial and return 1; otherwise return 0.

void **fmpz_mpoly_set_fmpz_poly**(*fmpz_mpoly_t* A, const *fmpz_poly_t* B, *slong* var, const *fmpz_mpoly_ctx_t* ctx)
 Set *A* to the univariate polynomial *B* in the variable with index *var*.

void **fmpz_mpoly_set_gen_fmpz_poly**(*fmpz_mpoly_t* A, *slong* var, const *fmpz_poly_t* B, const *fmpz_mpoly_ctx_t* ctx)
 Set *A* to the univariate polynomial *B* in the variable with index *var*.

4.10.11 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

fmpz ***fmpz_mpoly_term_coeff_ref**(*fmpz_mpoly_t* A, *slong* i, const *fmpz_mpoly_ctx_t* ctx)
 Return a reference to the coefficient of index *i* of *A*.

int **fmpz_mpoly_is_canonical**(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return 1 if *A* is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficient, and the terms must be sorted from greatest to least.

slong **fmpz_mpoly_length**(const *fmpz_mpoly_t* A, const *fmpz_mpoly_ctx_t* ctx)
 Return the number of terms in *A*. If the polynomial is in canonical form, this will be the number of nonzero coefficients.

void **fmpz_mpoly_resize**(*fmpz_mpoly_t* A, *slong* new_length, const *fmpz_mpoly_ctx_t* ctx)
 Set the length of *A* to *new_length*. Terms are either deleted from the end, or new zero terms are appended.

void **fmpz_mpoly_get_term_coeff_fmpz**(*fmpz_t* c, const *fmpz_mpoly_t* A, *slong* i, const *fmpz_mpoly_ctx_t* ctx)
 Either return or set *c* to the coefficient of the term of index *i*.

ulong **fmpz_mpoly_get_term_coeff_ui**(const *fmpz_mpoly_t* A, *slong* i, const *fmpz_mpoly_ctx_t* ctx)
 Return the coefficient of the term of index *i* as an unsigned integer.

slong **fmpz_mpoly_get_term_coeff_si**(const *fmpz_mpoly_t* poly, *slong* i, const *fmpz_mpoly_ctx_t* ctx)
 Return the coefficient of the term of index *i* as a signed integer.

void **fmpz_mpoly_set_term_coeff_fmpz**(*fmpz_mpoly_t* A, *slong* i, const *fmpz_t* c, const *fmpz_mpoly_ctx_t* ctx)
 Set the coefficient of the term of index *i* to *c*.

void **fmpz_mpoly_set_term_coeff_ui**(*fmpz_mpoly_t* A, *slong* i, *ulong* c, const *fmpz_mpoly_ctx_t* ctx)
 Set the coefficient of the term of index *i* to *c*.


```
void fmpz_mpoly_set_term_coeff_si(fmpz_mpoly_t A, slong i, slong c, const fmpz_mpoly_ctx_t ctx)
```

Set the coefficient of the term of index i to c .

```
int fmpz_mpoly_term_exp_fits_si(const fmpz_mpoly_t poly, slong i, const fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_term_exp_fits_ui(const fmpz_mpoly_t poly, slong i, const fmpz_mpoly_ctx_t ctx)
```

Return 1 if all entries of the exponent vector of the term of index i fit into an `slong` (resp. a `ulong`). Otherwise, return 0.

```
void fmpz_mpoly_get_term_exp_fmpz(fmpz **exp, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_get_term_exp_ui(ulong *exp, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_get_term_exp_si(slong *exp, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
```

Set exp to the exponent vector of the term of index i . The `_ui` (resp. `_si`) version throws if any entry does not fit into a `ulong` (resp. `slong`).

```
ulong fmpz_mpoly_get_term_var_exp_ui(const fmpz_mpoly_t A, slong i, slong var, const fmpz_mpoly_ctx_t ctx)
```

```
slong fmpz_mpoly_get_term_var_exp_si(const fmpz_mpoly_t A, slong i, slong var, const fmpz_mpoly_ctx_t ctx)
```

Return the exponent of the variable var of the term of index i . This function throws if the exponent does not fit into a `ulong` (resp. `slong`).

```
void fmpz_mpoly_set_term_exp_fmpz(fmpz_mpoly_t A, slong i, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_set_term_exp_ui(fmpz_mpoly_t A, slong i, const ulong *exp, const fmpz_mpoly_ctx_t ctx)
```

Set the exponent vector of the term of index i to exp .

```
void fmpz_mpoly_get_term(fmpz_mpoly_t M, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
```

Set M to the term of index i in A .

```
void fmpz_mpoly_get_term_monomial(fmpz_mpoly_t M, const fmpz_mpoly_t A, slong i, const fmpz_mpoly_ctx_t ctx)
```

Set M to the monomial of the term of index i in A . The coefficient of M will be one.

```
void fmpz_mpoly_push_term_fmpz_fmpz(fmpz_mpoly_t A, const fmpz_t c, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_fmpz_ffmpz(fmpz_mpoly_t A, const fmpz_t c, const fmpz *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_ui_fmpz(fmpz_mpoly_t A, ulong c, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_ui_ffmpz(fmpz_mpoly_t A, ulong c, const fmpz *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_si_fmpz(fmpz_mpoly_t A, slong c, fmpz *const *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_si_ffmpz(fmpz_mpoly_t A, slong c, const fmpz *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_fmpz_ui(fmpz_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_ui_ui(fmpz_mpoly_t A, ulong c, const ulong *exp, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_push_term_si_ui(fmpz_mpoly_t A, slong c, const ulong *exp, const
                               fmpz_mpoly_ctx_t ctx)
```

Append a term to A with coefficient c and exponent vector exp . This function runs in constant average time.

```
void fmpz_mpoly_sort_terms(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
```

Sort the terms of A into the canonical ordering dictated by the ordering in ctx . This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the size of A .

```
void fmpz_mpoly_combine_like_terms(fmpz_mpoly_t A, const fmpz_mpoly_ctx_t ctx)
```

Combine adjacent like terms in A and delete terms with coefficient zero. If the terms of A were sorted to begin with, the result will be in canonical form. This function runs in linear time in the size of A .

```
void fmpz_mpoly_reverse(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

Set A to the reversal of B .

4.10.12 Random generation

```
void fmpz_mpoly_randtest_bound(fmpz_mpoly_t A, flint_rand_t state, slong length, ulong
                               coeff_bits, ulong exp_bound, const fmpz_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents in the range $[0, exp\_bound - 1]$. The exponents of each variable are generated by calls to `n_randint(state, exp_bound)`.

```
void fmpz_mpoly_randtest_bounds(fmpz_mpoly_t A, flint_rand_t state, slong length, ulong
                                coeff_bits, ulong *exp_bounds, const fmpz_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents in the range $[0, exp\_bounds[i] - 1]$. The exponents of the variable of index i are generated by calls to `n_randint(state, exp_bounds[i])`.

```
void fmpz_mpoly_randtest_bits(fmpz_mpoly_t A, flint_rand_t state, slong length, ulong coeff_bits,
                              ulong exp_bits, const fmpz_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to the given length and exponents whose packed form does not exceed the given bit count.

The parameter `coeff_bits` to the three functions `fmpz_mpoly_randtest_{bound|bounds|bits}` is merely a suggestion for the approximate bit count of the resulting signed coefficients. The function `fmpz_mpoly_max_bits()` will give the exact bit count of the result.

4.10.13 Addition/Subtraction

```
void fmpz_mpoly_add_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const
                          fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_add_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t
                       ctx)
```

```
void fmpz_mpoly_add_si(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong c, const fmpz_mpoly_ctx_t
                       ctx)
```

Set A to $B + c$. If A and B are aliased, this function will probably run quickly.

```
void fmpz_mpoly_sub_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const
                          fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_sub_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t
                       ctx)
```

```
void fmpz_mpoly_sub_si(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong c, const fmpz_mpoly_ctx_t ctx)
```

Set A to $B - c$. If A and B are aliased, this function will probably run quickly.

```
void fmpz_mpoly_add(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx)
```

Set A to $B + C$. If A and B are aliased, this function might run in time proportional to the size of C .

```
void fmpz_mpoly_sub(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const fmpz_mpoly_ctx_t ctx)
```

Set A to $B - C$. If A and B are aliased, this function might run in time proportional to the size of C .

4.10.14 Scalar operations

```
void fmpz_mpoly_neg(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

Set A to $-B$.

```
void fmpz_mpoly_scalar_mul_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_scalar_mul_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_scalar_mul_si(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong c, const fmpz_mpoly_ctx_t ctx)
```

Set A to $B \times c$.

```
void fmpz_mpoly_scalar_fmms(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const fmpz_mpoly_t D, const fmpz_t e, const fmpz_mpoly_ctx_t ctx)
```

Sets A to $B \times c + D \times e$.

```
void fmpz_mpoly_scalar_divexact_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_scalar_divexact_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_scalar_divexact_si(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong c, const fmpz_mpoly_ctx_t ctx)
```

Set A to B divided by c . The division is assumed to be exact.

```
int fmpz_mpoly_scalar_divides_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t c, const fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_scalar_divides_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong c, const fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_scalar_divides_si(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong c, const fmpz_mpoly_ctx_t ctx)
```

If B is divisible by c , set A to the exact quotient and return 1, otherwise set A to zero and return 0.

4.10.15 Differentiation/Integration

```
void fmpz_mpoly_derivative(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong var, const
                          fmpz_mpoly_ctx_t ctx)
```

Set A to the derivative of B with respect to the variable of index var .

```
void fmpz_mpoly_integral(fmpz_mpoly_t A, fmpz_t scale, const fmpz_mpoly_t B, slong var, const
                        fmpz_mpoly_ctx_t ctx)
```

Set A and $scale$ so that A is an integral of $scale \times B$ with respect to the variable of index var , where $scale$ is positive and as small as possible.

4.10.16 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int fmpz_mpoly_evaluate_all_fmpz(fmpz_t ev, const fmpz_mpoly_t A, fmpz_t *const *vals, const
                                fmpz_mpoly_ctx_t ctx)
```

Set ev to the evaluation of A where the variables are replaced by the corresponding elements of the array $vals$. Return 1 for success and 0 for failure.

```
int fmpz_mpoly_evaluate_one_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, slong var, const
                                fmpz_t val, const fmpz_mpoly_ctx_t ctx)
```

Set A to the evaluation of B where the variable of index var is replaced by val . Return 1 for success and 0 for failure.

```
int fmpz_mpoly_compose_fmpz_poly(fmpz_poly_t A, const fmpz_mpoly_t B, fmpz_poly_struct
                                 *const *C, const fmpz_mpoly_ctx_t ctxB)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . The context object of B is $ctxB$. Return 1 for success and 0 for failure.

```
int fmpz_mpoly_compose_fmpz_mpoly_geobucket(fmpz_mpoly_t A, const fmpz_mpoly_t B,
                                             fmpz_mpoly_struct *const *C, const
                                             fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t
                                             ctxAC)
```

```
int fmpz_mpoly_compose_fmpz_mpoly_horner(fmpz_mpoly_t A, const fmpz_mpoly_t B,
                                           fmpz_mpoly_struct *const *C, const
                                           fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t
                                           ctxAC)
```

```
int fmpz_mpoly_compose_fmpz_mpoly(fmpz_mpoly_t A, const fmpz_mpoly_t B, fmpz_mpoly_struct
                                   *const *C, const fmpz_mpoly_ctx_t ctxB, const
                                   fmpz_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . Both A and the elements of C have context object $ctxAC$, while B has context object $ctxB$. The length of the array C is the number of variables in $ctxB$. Neither A nor B is allowed to alias any other polynomial. Return 1 for success and 0 for failure. The main method attempts to perform the calculation using matrices and chooses heuristically between the `geobucket` and `horner` methods if needed.

```
void fmpz_mpoly_compose_fmpz_mpoly_gen(fmpz_mpoly_t A, const fmpz_mpoly_t B, const slong *c,
                                       const fmpz_mpoly_ctx_t ctxB, const fmpz_mpoly_ctx_t
                                       ctxAC)
```

Set A to the evaluation of B where the variable of index i in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array C is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.

4.10.17 Multiplication

```
void fmpz_mpoly_mul(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const
    fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_mul_threaded(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C,
    const fmpz_mpoly_ctx_t ctx, slong thread_limit)
```

Set A to $B \times C$.

```
void fmpz_mpoly_mul_johnson(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C,
    const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_mul_heap_threaded(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t
    C, const fmpz_mpoly_ctx_t ctx)
```

Set A to $B \times C$ using Johnson's heap-based method. The first version always uses one thread.

```
int fmpz_mpoly_mul_array(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const
    fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_mul_array_threaded(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t
    C, const fmpz_mpoly_ctx_t ctx)
```

Try to set A to $B \times C$ using arrays. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1. The first version always uses one thread.

```
int fmpz_mpoly_mul_dense(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_t C, const
    fmpz_mpoly_ctx_t ctx)
```

Try to set A to $B \times C$ using dense arithmetic. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1.

4.10.18 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int fmpz_mpoly_pow_fmpz(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_t k, const
    fmpz_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

```
int fmpz_mpoly_pow_ui(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong k, const fmpz_mpoly_ctx_t
    ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

4.10.19 Division

```
int fmpz_mpoly_divides(fmpz_mpoly_t Q, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const
    fmpz_mpoly_ctx_t ctx)
```

If A is divisible by B , set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0.

```
void fmpz_mpoly_divrem(fmpz_mpoly_t Q, fmpz_mpoly_t R, const fmpz_mpoly_t A, const
    fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

Set Q and R to the quotient and remainder of A divided by B . The monomials in R divisible by the leading monomial of B will have coefficients reduced modulo the absolute value of the leading coefficient of B . Note that this function is not very useful if the leading coefficient B is not a unit.

```
void fmpz_mpoly_quasidivrem(fmpz_t scale, fmpz_mpoly_t Q, fmpz_mpoly_t R, const
    fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t
    ctx)
```

Set $scale$, Q and R so that Q and R are the quotient and remainder of $scale \times A$ divided by B . No monomials in R will be divisible by the leading monomial of B .

```
void fmpz_mpoly_div(fmpz_mpoly_t Q, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const
    fmpz_mpoly_ctx_t ctx)
```

Perform the operation of `fmpz_mpoly_divrem()` and discard R . Note that this function is not very useful if the division is not exact and the leading coefficient B is not a unit.

```
void fmpz_mpoly_quasidiv(fmpz_t scale, fmpz_mpoly_t Q, const fmpz_mpoly_t A, const
    fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

Perform the operation of `fmpz_mpoly_quasidivrem()` and discard R .

```
void fmpz_mpoly_divrem_ideal(fmpz_mpoly_struct **Q, fmpz_mpoly_t R, const fmpz_mpoly_t A,
    fmpz_mpoly_struct *const *B, slong len, const fmpz_mpoly_ctx_t
    ctx)
```

This function is as per `fmpz_mpoly_divrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q . The number of divisor (and hence quotient) polynomials is given by len . Note that this function is not very useful if there is no unit among the leading coefficients in the array B .

```
void fmpz_mpoly_quasidivrem_ideal(fmpz_t scale, fmpz_mpoly_struct **Q, fmpz_mpoly_t R, const
    fmpz_mpoly_t A, fmpz_mpoly_struct *const *B, slong len,
    const fmpz_mpoly_ctx_t ctx)
```

This function is as per `fmpz_mpoly_quasidivrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q . The number of divisor (and hence quotient) polynomials is given by len .

4.10.20 Greatest Common Divisor

```
void fmpz_mpoly_term_content(fmpz_mpoly_t M, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t
    ctx)
```

Set M to the GCD of the terms of A . If A is zero, M will be zero. Otherwise, M will be a monomial with positive coefficient.

```
int fmpz_mpoly_content_vars(fmpz_mpoly_t g, const fmpz_mpoly_t A, slong *vars, slong
    vars_length, const fmpz_mpoly_ctx_t ctx)
```

Set g to the GCD of the coefficients of A when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, g will be independent of the variables $vars$.

```
int fmpz_mpoly_gcd(fmpz_mpoly_t G, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const
    fmpz_mpoly_ctx_t ctx)
```

Try to set G to the GCD of A and B with positive leading coefficient. The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and G is left untouched.

```
int fmpz_mpoly_gcd_cofactors(fmpz_mpoly_t G, fmpz_mpoly_t Abar, fmpz_mpoly_t Bbar, const
    fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz_mpoly_ctx_t
    ctx)
```

Do the operation of `fmpz_mpoly_gcd()` and also compute $Abar = A/G$ and $Bbar = B/G$ if successful.

```
int fmpz_mpoly_gcd_brown(fmpz_mpoly_t G, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const
    fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_gcd_hensel(fmpz_mpoly_t G, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const
    fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_gcd_subresultant(fmpz_mpoly_t G, const fmpz_mpoly_t A, const fmpz_mpoly_t
    B, const fmpz_mpoly_ctx_t ctx)
```

```
int fmpz_mpoly_gcd_zippel(fmpz_mpoly_t G, const fmpz_mpoly_t A, const fmpz_mpoly_t B, const
    fmpz_mpoly_ctx_t ctx)
```



```
int fmpz_poly_gcd_zippel2(fmpz_poly_t G, const fmpz_poly_t A, const fmpz_poly_t B,
                        const fmpz_poly_ctx_t ctx)
```

Try to set G to the GCD of A and B using various algorithms.

```
int fmpz_poly_resultant(fmpz_poly_t R, const fmpz_poly_t A, const fmpz_poly_t B, slong
                      var, const fmpz_poly_ctx_t ctx)
```

Try to set R to the resultant of A and B with respect to the variable of index var .

```
int fmpz_poly_discriminant(fmpz_poly_t D, const fmpz_poly_t A, slong var, const
                          fmpz_poly_ctx_t ctx)
```

Try to set D to the discriminant of A with respect to the variable of index var .

```
void fmpz_poly_primitive_part(fmpz_poly_t res, const fmpz_poly_t f, const
                             fmpz_poly_ctx_t ctx)
```

Sets res to the primitive part of f , obtained by dividing out the content of all coefficients and normalizing the leading coefficient to be positive. The zero polynomial is unchanged.

4.10.21 Square Root

```
int fmpz_poly_sqrt_heap(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_ctx_t ctx,
                      int check)
```

If A is a perfect square return 1 and set Q to the square root with positive leading coefficient. Otherwise return 0 and set Q to the zero polynomial. If $check = 0$ the polynomial is assumed to be a perfect square. This can be significantly faster, but it will not detect non-squares with any reliability, and in the event of being passed a non-square the result is meaningless.

```
int fmpz_poly_sqrt(fmpz_poly_t q, const fmpz_poly_t A, const fmpz_poly_ctx_t ctx)
```

If A is a perfect square return 1 and set Q to the square root with positive leading coefficient. Otherwise return 0 and set Q to zero.

```
int fmpz_poly_is_square(const fmpz_poly_t A, const fmpz_poly_ctx_t ctx)
```

Return 1 if A is a perfect square, otherwise return 0.

4.10.22 Univariate Functions

An `fmpz_poly_univar_t` holds a univariate polynomial in some main variable with `fmpz_poly_t` coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Z}[x_1, \dots, x_m]$ as an element of $(\mathbb{Z}[x_1, \dots, x_{v-1}, x_{v+1}, \dots, x_m])[x_v]$ and vice versa.

```
void fmpz_poly_univar_init(fmpz_poly_univar_t A, const fmpz_poly_ctx_t ctx)
```

Initialize A .

```
void fmpz_poly_univar_clear(fmpz_poly_univar_t A, const fmpz_poly_ctx_t ctx)
```

Clear A .

```
void fmpz_poly_univar_swap(fmpz_poly_univar_t A, fmpz_poly_univar_t B, const
                          fmpz_poly_ctx_t ctx)
```

Swap A and B .

```
void fmpz_poly_to_univar(fmpz_poly_univar_t A, const fmpz_poly_t B, slong var, const
                       fmpz_poly_ctx_t ctx)
```

Set A to a univariate form of B by pulling out the variable of index var . The coefficients of A will still belong to the content ctx but will not depend on the variable of index var .


```
void fmpz_mpoly_from_univar(fmpz_mpoly_t A, const fmpz_mpoly_univar_t B, slong var, const
                           fmpz_mpoly_ctx_t ctx)
```

Set A to the normal form of B by putting in the variable of index var . This function is undefined if the coefficients of B depend on the variable of index var .

```
int fmpz_mpoly_univar_degree_fits_si(const fmpz_mpoly_univar_t A, const fmpz_mpoly_ctx_t
                                     ctx)
```

Return 1 if the degree of A with respect to the main variable fits an *slong*. Otherwise, return 0.

```
slong fmpz_mpoly_univar_length(const fmpz_mpoly_univar_t A, const fmpz_mpoly_ctx_t ctx)
```

Return the number of terms in A with respect to the main variable.

```
slong fmpz_mpoly_univar_get_term_exp_si(fmpz_mpoly_univar_t A, slong i, const
                                         fmpz_mpoly_ctx_t ctx)
```

Return the exponent of the term of index i of A .

```
void fmpz_mpoly_univar_get_term_coeff(fmpz_mpoly_t c, const fmpz_mpoly_univar_t A, slong i,
                                      const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_univar_swap_term_coeff(fmpz_mpoly_t c, fmpz_mpoly_univar_t A, slong i,
                                        const fmpz_mpoly_ctx_t ctx)
```

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A .

4.10.23 Internal Functions

```
void fmpz_mpoly_inflate(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz *shift, const fmpz
                       *stride, const fmpz_mpoly_ctx_t ctx)
```

Apply the function $e \rightarrow \text{shift}[v] + \text{stride}[v]*e$ to each exponent e corresponding to the variable v . It is assumed that each shift and stride is not negative.

```
void fmpz_mpoly_deflate(fmpz_mpoly_t A, const fmpz_mpoly_t B, const fmpz *shift, const fmpz
                       *stride, const fmpz_mpoly_ctx_t ctx)
```

Apply the function $e \rightarrow (e - \text{shift}[v])/\text{stride}[v]$ to each exponent e corresponding to the variable v . If any $\text{stride}[v]$ is zero, the corresponding numerator $e - \text{shift}[v]$ is assumed to be zero, and the quotient is defined as zero. This allows the function to undo the operation performed by `fmpz_mpoly_inflate()` when possible.

```
void fmpz_mpoly_deflation(fmpz *shift, fmpz *stride, const fmpz_mpoly_t A, const
                          fmpz_mpoly_ctx_t ctx)
```

For each variable v let S_v be the set of exponents appearing on v . Set $\text{shift}[v]$ to $\min(S_v)$ and set $\text{stride}[v]$ to $\gcd(S - \min(S_v))$. If A is zero, all shifts and strides are set to zero.

```
void fmpz_mpoly_pow_fps(fmpz_mpoly_t A, const fmpz_mpoly_t B, ulong k, const
                       fmpz_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power, using the Monagan and Pearce FPS algorithm. It is assumed that B is not zero and $k \geq 2$.

```
slong _fmpz_mpoly_divides_array(fmpz **poly1, ulong **exp1, slong *alloc, const fmpz *poly2,
                                const ulong *exp2, slong len2, const fmpz *poly3, const ulong
                                *exp3, slong len3, slong *mults, slong num, slong bits)
```

Use dense array exact division to set $(\text{poly1}, \text{exp1}, \text{alloc})$ to $(\text{poly2}, \text{exp3}, \text{len2})$ divided by $(\text{poly3}, \text{exp3}, \text{len3})$ in num variables, given a list of multipliers to tightly pack exponents and a number of bits for the fields of the exponents of the result. The array “mults” is a list of bases to be used in encoding the array indices from the exponents. The function reallocates its output, hence the double indirection, and returns the length of its output if the quotient is exact, or zero if not. It is assumed that poly2 is not zero. No aliasing is allowed.

```
int fmpz_mpoly_divides_array(fmpz_mpoly_t poly1, const fmpz_mpoly_t poly2, const
                             fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)
```

Set `poly1` to `poly2` divided by `poly3`, using a big dense array to accumulate coefficients, and return 1 if the quotient is exact. Otherwise, return 0 if the quotient is not exact. If the array will be larger than some internally set parameter, the function fails silently and returns `-1` so that some other method may be called. This function is most efficient on dense inputs. Note that the function `fmpz_mpoly_div_monagan_pearce` below may be much faster if the quotient is known to be exact.

```
slong _fmpz_mpoly_divides_monagan_pearce(fmpz **poly1, ulong **exp1, slong *alloc, const fmpz
                                         *poly2, const ulong *exp2, slong len2, const fmpz
                                         *poly3, const ulong *exp3, slong len3, ulong bits, slong
                                         N, const ulong *cmpmask)
```

Set `(poly1, exp1, alloc)` to `(poly2, exp3, len2)` divided by `(poly3, exp3, len3)` and return 1 if the quotient is exact. Otherwise return 0. The function assumes exponent vectors that each fit in N words, and are packed into fields of the given number of bits. Assumes input polys are nonzero. Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. No aliasing is allowed.

```
int fmpz_mpoly_divides_monagan_pearce(fmpz_mpoly_t poly1, const fmpz_mpoly_t poly2, const
                                       fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)
```

Set `poly1` to `poly2` divided by `poly3` and return 1 if the quotient is exact. Otherwise return 0. The function uses the algorithm of Michael Monagan and Roman Pearce. Note that the function `fmpz_mpoly_div_monagan_pearce` below may be much faster if the quotient is known to be exact.

```
int fmpz_mpoly_divides_heap_threaded(fmpz_mpoly_t Q, const fmpz_mpoly_t A, const
                                     fmpz_mpoly_t B, const fmpz_mpoly_ctx_t ctx)
```

The same method as used as in `fmpz_mpoly_divides_monagan_pearce()`, but is also multi-threaded.

Note: This function is only defined if the machine is known to be strongly ordered during the configuration. To check whether this function is defined during compilation-time, use the C preprocessor macro `#ifdef fmpz_mpoly_divides_heap_threaded`.

Note that, if the system is known to be strongly ordered, the underlying algorithm for this function is utilized in `fmpz_mpoly_divides()`. Hence, you may find it easier to use this function instead if the C preprocessor is not available.

```
slong _fmpz_mpoly_div_monagan_pearce(fmpz **polyq, ulong **expq, slong *allocq, const fmpz
                                     *poly2, const ulong *exp2, slong len2, const fmpz *poly3,
                                     const ulong *exp3, slong len3, slong bits, slong N, const
                                     ulong *cmpmask)
```

Set `(polyq, expq, allocq)` to the quotient of `(poly2, exp2, len2)` by `(poly3, exp3, len3)` discarding remainder (with notional remainder coefficients reduced modulo the leading coefficient of `(poly3, exp3, len3)`), and return the length of the quotient. The function reallocates its output, hence the double indirection. The function assumes the exponent vectors all fit in N words. The exponent vectors are assumed to have fields with the given number of bits. Assumes input polynomials are nonzero. Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. No aliasing is allowed.

```
void fmpz_mpoly_div_monagan_pearce(fmpz_mpoly_t polyq, const fmpz_mpoly_t poly2, const
                                   fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)
```

Set `polyq` to the quotient of `poly2` by `poly3`, discarding the remainder (with notional remainder coefficients reduced modulo the leading coefficient of `poly3`). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. This function is exceptionally efficient if the division is known to be exact.

```
slong _fmpz_mpoly_divrem_monagan_pearce(slong *lenr, fmpz **polyq, ulong **expq, slong *allocq,
                                         fmpz **polyr, ulong **expr, slong *allocr, const fmpz
                                         *poly2, const ulong *exp2, slong len2, const fmpz
                                         *poly3, const ulong *exp3, slong len3, slong bits, slong
                                         N, const ulong *cmpmask)
```

Set (polyq, expq, allocq) and (polyr, expr, allocr) to the quotient and remainder of (poly2, exp2, len2) by (poly3, exp3, len3) (with remainder coefficients reduced modulo the leading coefficient of (poly3, exp3, len3)), and return the length of the quotient. The function reallocates its outputs, hence the double indirection. The function assumes the exponent vectors all fit in N words. The exponent vectors are assumed to have fields with the given number of bits. Assumes input polynomials are nonzero. Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. No aliasing is allowed.

```
void fmpz_mpoly_divrem_monagan_pearce(fmpz_mpoly_t q, fmpz_mpoly_t r, const fmpz_mpoly_t
                                         poly2, const fmpz_mpoly_t poly3, const
                                         fmpz_mpoly_ctx_t ctx)
```

Set polyq and polyr to the quotient and remainder of poly2 divided by poly3 (with remainder coefficients reduced modulo the leading coefficient of poly3). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce.

```
slong _fmpz_mpoly_divrem_array(slong *lenr, fmpz **polyq, ulong **expq, slong *allocq, fmpz
                               **polyr, ulong **expr, slong *allocr, const fmpz *poly2, const
                               ulong *exp2, slong len2, const fmpz *poly3, const ulong *exp3,
                               slong len3, slong *mults, slong num, slong bits)
```

Use dense array division to set (polyq, expq, allocq) and (polyr, expr, allocr) to the quotient and remainder of (poly2, exp2, len2) divided by (poly3, exp3, len3) in num variables, given a list of multipliers to tightly pack exponents and a number of bits for the fields of the exponents of the result. The function reallocates its outputs, hence the double indirection. The array mults is a list of bases to be used in encoding the array indices from the exponents. The function returns the length of the quotient. It is assumed that the input polynomials are not zero. No aliasing is allowed.

```
int fmpz_mpoly_divrem_array(fmpz_mpoly_t q, fmpz_mpoly_t r, const fmpz_mpoly_t poly2, const
                               fmpz_mpoly_t poly3, const fmpz_mpoly_ctx_t ctx)
```

Set polyq and polyr to the quotient and remainder of poly2 divided by poly3 (with remainder coefficients reduced modulo the leading coefficient of poly3). The function is implemented using dense arrays, and is efficient when the inputs are fairly dense. If the array will be larger than some internally set parameter, the function silently returns 0 so that another function can be called, otherwise it returns 1.

```
void fmpz_mpoly_quasidivrem_heap(fmpz_t scale, fmpz_mpoly_t q, fmpz_mpoly_t r, const
                                fmpz_mpoly_t poly2, const fmpz_mpoly_t poly3, const
                                fmpz_mpoly_ctx_t ctx)
```

Set scale, q and r so that $\text{scale} \cdot \text{poly2} = \text{q} \cdot \text{poly3} + \text{r}$ and no monomial in r is divisible by the leading monomial of poly3, where scale is positive and as small as possible. This function throws an exception if poly3 is zero or if an exponent overflow occurs.

```
slong _fmpz_mpoly_divrem_ideal_monagan_pearce(fmpz_mpoly_struct **polyq, fmpz **polyr, ulong
                                                **expr, const slong *allocr, const fmpz *poly2, const
                                                ulong *exp2, slong len2, fmpz_mpoly_struct
                                                *const *poly3, ulong *const *exp3, slong len,
                                                slong N, slong bits, const fmpz_mpoly_ctx_t ctx,
                                                const ulong *cmpmask)
```

This function is as per `_fmpz_mpoly_divrem_monagan_pearce` except that it takes an array of divisor polynomials poly3 and an array of repacked exponent arrays exp3, which may alias the exponent arrays of poly3, and it returns an array of quotient polynomials polyq. The number of divisor (and hence quotient) polynomials is given by len. The function computes polynomials q_i such that $r = a - \sum_{i=0}^{\text{len}-1} q_i b_i$, where the q_i are the quotient polynomials and the b_i are the divisor polynomials.

```
void fmpz_mpoly_divrem_ideal_monagan_pearce(fmpz_mpoly_struct **q, fmpz_mpoly_t r, const
                                           fmpz_mpoly_t poly2, fmpz_mpoly_struct *const
                                           *poly3, slong len, const fmpz_mpoly_ctx_t ctx)
```

This function is as per `fmpz_mpoly_divrem_monagan_pearce` except that it takes an array of divisor polynomials `poly3`, and it returns an array of quotient polynomials `q`. The number of divisor (and hence quotient) polynomials is given by `len`. The function computes polynomials $q_i = q[i]$ such that `poly2` is $r + \sum_{i=0}^{\text{len}-1} q_i b_i$, where $b_i = \text{poly3}[i]$.

4.10.24 Vectors

```
type fmpz_mpoly_vec_struct
```

```
type fmpz_mpoly_vec_t
```

A type holding a vector of `fmpz_mpoly_t`.

```
fmpz_mpoly_vec_entry(vec, i)
```

Macro for accessing the entry at position *i* in *vec*.

```
void fmpz_mpoly_vec_init(fmpz_mpoly_vec_t vec, slong len, const fmpz_mpoly_ctx_t ctx)
```

Initializes *vec* to a vector of length *len*, setting all entries to the zero polynomial.

```
void fmpz_mpoly_vec_clear(fmpz_mpoly_vec_t vec, const fmpz_mpoly_ctx_t ctx)
```

Clears *vec*, freeing its allocated memory.

```
void fmpz_mpoly_vec_print(const fmpz_mpoly_vec_t vec, const fmpz_mpoly_ctx_t ctx)
```

Prints *vec* to standard output.

```
void fmpz_mpoly_vec_swap(fmpz_mpoly_vec_t x, fmpz_mpoly_vec_t y, const fmpz_mpoly_ctx_t
                        ctx)
```

Swaps *x* and *y* efficiently.

```
void fmpz_mpoly_vec_fit_length(fmpz_mpoly_vec_t vec, slong len, const fmpz_mpoly_ctx_t ctx)
```

Allocates room for *len* entries in *vec*.

```
void fmpz_mpoly_vec_set(fmpz_mpoly_vec_t dest, const fmpz_mpoly_vec_t src, const
                      fmpz_mpoly_ctx_t ctx)
```

Sets *dest* to a copy of *src*.

```
void fmpz_mpoly_vec_append(fmpz_mpoly_vec_t vec, const fmpz_mpoly_t f, const
                          fmpz_mpoly_ctx_t ctx)
```

Appends *f* to the end of *vec*.

```
slong fmpz_mpoly_vec_insert_unique(fmpz_mpoly_vec_t vec, const fmpz_mpoly_t f, const
                                  fmpz_mpoly_ctx_t ctx)
```

Inserts *f* without duplication into *vec* and returns its index. If this polynomial already exists, *vec* is unchanged. If this polynomial does not exist in *vec*, it is appended.

```
void fmpz_mpoly_vec_set_length(fmpz_mpoly_vec_t vec, slong len, const fmpz_mpoly_ctx_t ctx)
```

Sets the length of *vec* to *len*, truncating or zero-extending as needed.

```
void fmpz_mpoly_vec_randtest_not_zero(fmpz_mpoly_vec_t vec, flint_rand_t state, slong len,
                                     slong poly_len, slong bits, ulong exp_bound,
                                     fmpz_mpoly_ctx_t ctx)
```

Sets *vec* to a random vector with exactly *len* entries, all nonzero, with random parameters defined by *poly_len*, *bits* and *exp_bound*.

```
void fmpz_mpoly_vec_set_primitive_unique(fmpz_mpoly_vec_t res, const fmpz_mpoly_vec_t src,
                                         const fmpz_mpoly_ctx_t ctx)
```

Sets *res* to a vector containing all polynomials in *src* reduced to their primitive parts, without duplication. The zero polynomial is skipped if present. The output order is arbitrary.

4.10.25 Ideals and Gröbner bases

The following methods deal with ideals in $\mathbb{Q}[X_1, \dots, X_n]$. We use primitive integer polynomials as normalised generators in place of monic rational polynomials.

void **fmpz_mpoly_spoly**(*fmpz_mpoly_t* res, const *fmpz_mpoly_t* f, const *fmpz_mpoly_t* g, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the *S*-polynomial of *f* and *g*, scaled to an integer polynomial by computing the LCM of the leading coefficients.

void **fmpz_mpoly_reduction_primitive_part**(*fmpz_mpoly_t* res, const *fmpz_mpoly_t* f, const *fmpz_mpoly_vec_t* vec, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the primitive part of the reduction (remainder of multivariate quasidivision with remainder) with respect to the polynomials *vec*.

int **fmpz_mpoly_vec_is_groebner**(const *fmpz_mpoly_vec_t* G, const *fmpz_mpoly_vec_t* F, const *fmpz_mpoly_ctx_t* ctx)

If *F* is *NULL*, checks if *G* is a Gröbner basis. If *F* is not *NULL*, checks if *G* is a Gröbner basis for *F*.

int **fmpz_mpoly_vec_is_autoreduced**(const *fmpz_mpoly_vec_t* F, const *fmpz_mpoly_ctx_t* ctx)

Checks whether the vector *F* is autoreduced (or inter-reduced).

void **fmpz_mpoly_vec_autoreduction**(*fmpz_mpoly_vec_t* H, const *fmpz_mpoly_vec_t* F, const *fmpz_mpoly_ctx_t* ctx)

Sets *H* to the autoreduction (inter-reduction) of *F*.

void **fmpz_mpoly_vec_autoreduction_groebner**(*fmpz_mpoly_vec_t* H, const *fmpz_mpoly_vec_t* G, const *fmpz_mpoly_ctx_t* ctx)

Sets *H* to the autoreduction (inter-reduction) of *G*. Assumes that *G* is a Gröbner basis. This produces a reduced Gröbner basis, which is unique (up to the sort order of the entries in the vector).

void **fmpz_mpoly_buchberger_naive**(*fmpz_mpoly_vec_t* G, const *fmpz_mpoly_vec_t* F, const *fmpz_mpoly_ctx_t* ctx)

Sets *G* to a Gröbner basis for *F*, computed using a naive implementation of Buchberger's algorithm.

int **fmpz_mpoly_buchberger_naive_with_limits**(*fmpz_mpoly_vec_t* G, const *fmpz_mpoly_vec_t* F, *slong* ideal_len_limit, *slong* poly_len_limit, *slong* poly_bits_limit, const *fmpz_mpoly_ctx_t* ctx)

As *fmpz_mpoly_buchberger_naive()*, but halts if during the execution of Buchberger's algorithm the length of the ideal basis set exceeds *ideal_len_limit*, the length of any polynomial exceeds *poly_len_limit*, or the size of the coefficients of any polynomial exceeds *poly_bits_limit*. Returns 1 for success and 0 for failure. On failure, *G* is a valid basis for *F* but it might not be a Gröbner basis.

4.10.26 Special polynomials

void **fmpz_mpoly_symmetric_gens**(*fmpz_mpoly_t* res, *ulong* k, *slong* *vars, *slong* n, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_symmetric**(*fmpz_mpoly_t* res, *ulong* k, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the elementary symmetric polynomial $e_k(X_1, \dots, X_n)$.

The *gens* version takes $X_1, \dots, X_n$ to be the subset of generators given by *vars* and *n*. The indices in *vars* start from zero. Currently, the indices in *vars* must be distinct.

4.11 fmpz_mpoly_factor.h – factorisation of multivariate polynomials over the integers

4.11.1 Types, macros and constants

type **fmpz_mpoly_factor_struct**

A struct for holding a factored integer polynomial. There is a single constant and a product of bases to corresponding exponents.

type **fmpz_mpoly_factor_t**

An array of length 1 of **fmpz_mpoly_factor_struct**.

4.11.2 Memory management

void **fmpz_mpoly_factor_init**(*fmpz_mpoly_factor_t* f, const *fmpz_mpoly_ctx_t* ctx)

Initialise *f*.

void **fmpz_mpoly_factor_clear**(*fmpz_mpoly_factor_t* f, const *fmpz_mpoly_ctx_t* ctx)

Clear *f*.

4.11.3 Basic manipulation

void **fmpz_mpoly_factor_swap**(*fmpz_mpoly_factor_t* f, *fmpz_mpoly_factor_t* g, const *fmpz_mpoly_ctx_t* ctx)

Efficiently swap *f* and *g*.

slong **fmpz_mpoly_factor_length**(const *fmpz_mpoly_factor_t* f, const *fmpz_mpoly_ctx_t* ctx)

Return the length of the product in *f*.

void **fmpz_mpoly_factor_get_constant_fmpz**(*fmpz_t* c, const *fmpz_mpoly_factor_t* f, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_factor_get_constant_fmpq**(*fmpq_t* c, const *fmpz_mpoly_factor_t* f, const *fmpz_mpoly_ctx_t* ctx)

Set *c* to the constant of *f*.

void **fmpz_mpoly_factor_get_base**(*fmpz_mpoly_t* B, const *fmpz_mpoly_factor_t* f, *slong* i, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_factor_swap_base**(*fmpz_mpoly_t* B, *fmpz_mpoly_factor_t* f, *slong* i, const *fmpz_mpoly_ctx_t* ctx)

Set (resp. swap) *B* to (resp. with) the base of the term of index *i* in *A*.

slong **fmpz_mpoly_factor_get_exp_si**(*fmpz_mpoly_factor_t* f, *slong* i, const *fmpz_mpoly_ctx_t* ctx)

Return the exponent of the term of index *i* in *A*. It is assumed to fit an *slong*.

void **fmpz_mpoly_factor_sort**(*fmpz_mpoly_factor_t* f, const *fmpz_mpoly_ctx_t* ctx)

Sort the product of *f* first by exponent and then by base.

4.11.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A : f is simply set to a factorisation of A , and thus these functions should not depend on the initial value of the output f .

```
int fmpz_mpoly_factor_squarefree(fmpz_mpoly_factor_t f, const fmpz_mpoly_t A, const
                                fmpz_mpoly_ctx_t ctx)
```

Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call `fmpz_mpoly_factor_sort()` and then multiply the bases with the desired exponent.

```
int fmpz_mpoly_factor(fmpz_mpoly_factor_t f, const fmpz_mpoly_t A, const fmpz_mpoly_ctx_t
                     ctx)
```

Set f to a factorization of A where the bases are irreducible.

4.12 long_extras.h – support functions for signed word arithmetic

4.12.1 Properties

```
size_t z_sizeinbase(slong n, int b)
```

Returns the number of digits in the base b representation of the absolute value of the integer n .

Assumes that $b \geq 2$.

4.12.2 Checked Arithmetic

```
int z_mul_checked(slong *a, slong b, slong c)
```

Set $*a$ to b times c and return 1 if the product overflowed. Otherwise, return 0.

4.12.3 Random functions

```
slong z_randtest(flint_rand_t state)
```

Returns a pseudo random number with a random number of bits, from 0 to FLINT_BITS. The probability of the special values 0, ± 1 , COEFF_MAX, COEFF_MIN, WORD_MAX and WORD_MIN is increased.

This random function is mainly used for testing purposes.

```
slong z_randtest_not_zero(flint_rand_t state)
```

As for `z_randtest(state)`, but does not return 0.

```
slong z_randint(flint_rand_t state, ulong limit)
```

Returns a pseudo random number of absolute value less than `limit`. If `limit` is zero or exceeds WORD_MAX, it is interpreted as WORD_MAX.

4.12.4 Modular arithmetic

`int z_kronecker(slong a, slong n)`

Return the Kronecker symbol $\left(\frac{a}{n}\right)$ for any a and any n .

4.13 longlong.h – support functions for multi-word arithmetic

4.13.1 Bit manipulation

`flint_clz(x)`

Returns the number of zero-bits from the msb to the first non-zero bit in the limb x . This is the number of steps x needs to be shifted left to set the most significant bit in x . If x is zero then the return value is undefined.

`flint_ctz(x)`

As for `flint_clz()`, but counts from the least significant end. If x is zero then the return value is undefined.

`flint_bitcnt_t FLINT_BIT_COUNT(ulong x)`

Returns the number of binary bits required to represent x . If x is zero it returns 0. This is an inline-function only.

`FLINT_FLOG2(x)`

`FLINT_CLOG2(x)`

For $x \geq 1$, it returns $\lfloor \log_2 x \rfloor$ and $\lceil \log_2 x \rceil$, respectively.

4.13.2 Addition and subtraction

Note: When aliasing inputs with outputs in these addition and subtraction macros, make sure to have s_i aliased with a_i for addition macros, and d_i aliased with m_i for optimal performance. Moreover, keep immediates (in other words, constants known to the compiler) in the b_i variables for addition and s_i for subtraction.

`add_ssaaaa(s1, s0, a1, a0, b1, b0)`

Sets s_1 and s_0 according to $cB^2 + s_1B + s_0 = (a_1B + a_0) + (b_1B + b_0)$, where $B = 2^{\text{FLINT\_BITS}}$ is the base, and c is the carry from the addition which is not stored anywhere.

`add_sssaaaaa(s2, s1, s0, a2, a1, a0, b2, b1, b0)`

Works like `add_ssaaaa`, but for two three-limbed integers. Carry is lost.

`sub_ddmmss(d1, d0, m1, m0, s1, s0)`

Sets d_1 and d_0 to the difference between the two-limbed integers $m_1B + m_0$ and $s_1B + s_0$, where $B = 2^{\text{FLINT\_BITS}}$. Borrow from the subtraction is not stored anywhere.

`sub_dddmmssss(d2, d1, d0, m2, m1, m0, s2, s1, s0)`

Works like `sub_dddmmssss`, but for two three-limbed integers. Borrow is lost.

4.13.3 Multiplication

umul_ppmm(p1, p0, u, v)

Computes $p_1B + p_0 = uv$, where $B = 2^{\text{FLINT\_BITS}}$.

smul_ppmm(p1, p0, u, v)

Works like **umul_ppmm** but for signed numbers.

4.13.4 Division

udiv_qrnnd(q, r, n1, n0, d)

Computes the non-negative integers q and r in $dq + r = n_1B + n_0$, where $B = 2^{\text{FLINT\_BITS}}$. Assumes that $n_1 < d$.

sdiv_qrnnd(quotient, remainder, high_numerator, low_numerator, denominator)

Works like **udiv_qrnnd**, but for signed numbers.

udiv_qrnnd_preinv(q, r, n1, n0, d, di)

Works like **udiv_qrnnd**, but takes a precomputed inverse **di** as computed by **n_preinvert_limb_prenorm()**. This function assumes **d** is normalised, i.e. with most significant bit set.

4.14 mpn_extras.h – support functions for limb arrays

4.14.1 Macros

MPN_NORM(a, an)

Normalise (**a**, **an**) so that either **an** is zero or **a[an - 1]** is nonzero.

MPN_SWAP(a, an, b, bn)

Swap (**a**, **an**) and (**b**, **bn**), i.e. swap pointers and sizes.

4.14.2 Utility functions

void **flint_mpn_debug**(mp_srcptr x, mp_size_t xsize)

Prints debug information about (**x**, **xsize**) to **stdout**. In particular, this will print binary representations of all the limbs.

char ***flint_mpn_get_str**(char *res, int base, mp_srcptr x, mp_size_t xn, int negative)

Returns the string representation of (**x**, **xn**) (or its negation if **negative** is set to 1) in base *base* which must be a base supported by GMP. If **res** is **NULL**, a new string will be allocated; otherwise, the given pointer **res** will be used and is assumed to have sufficient space to represent the full output, one extra digit, minus sign (if negative), and null terminator.

int **flint_mpn_zero_p**(mp_srcptr x, mp_size_t xsize)

Returns 1 if all limbs of (**x**, **xsize**) are zero, otherwise 0.

int **flint_mpn_equal_p**(mp_srcptr x, mp_srcptr y, mp_size_t xsize)

Returns 1 if all limbs of (**x**, **xsize**) and (**y**, **xsize**) are equal, otherwise 0.

4.14.3 Addition and subtraction

`mp_limb_t flint_mpn_sumdiff_n(mp_ptr s, mp_ptr d, mp_srcptr x, mp_srcptr y, mp_size_t n)`

Simultaneously computes the sum s and difference d of (x, n) and (y, n) , returning carry multiplied by two plus borrow.

`void flint_mpn_negmod_n(mp_ptr res, mp_srcptr x, mp_srcptr m, mp_size_t n)`

`void flint_mpn_addmod_n(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_srcptr m, mp_size_t n)`

`void flint_mpn_submod_n(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_srcptr m, mp_size_t n)`

`void flint_mpn_addmod_n_m(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_size_t yn, mp_srcptr m, mp_size_t n)`

`void flint_mpn_submod_n_m(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_size_t yn, mp_srcptr m, mp_size_t n)`

Arithmetic modulo (m, n) . These functions assume that (x, n) and (y, n) are already reduced modulo (m, n) . The `n_m` variants accept (y, yn) with $yn \leq n$, where (y, yn) is already reduced modulo (m, n) .

`void flint_mpn_negmod_2(mp_ptr res, mp_srcptr x, mp_srcptr m)`

`void flint_mpn_addmod_2(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_srcptr m)`

`void _flint_mpn_addmod_2(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_srcptr m)`

`void flint_mpn_submod_2(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_srcptr m)`

Modular arithmetic specialized for two limbs. The `_flint_mpn_addmod_2` version assumes that the most significant bit of $m[1]$ is not set.

`int flint_mpn_signed_sub_n(mp_ptr res, mp_srcptr x, mp_srcptr y, mp_size_t n)`

Sets res to $|x - y|$, returning 0 if the result equals $x - y$ and returning 1 if the result equals $y - x$.

4.14.4 Multiplication

`mp_limb_t flint_mpn_mul(mp_ptr z, mp_srcptr x, mp_size_t xn, mp_srcptr y, mp_size_t yn)`

Sets $(z, xn+yn)$ to the product of (x, xn) and (y, yn) and returns the top limb of the result. We require $xn \geq yn \geq 1$ and that z is not aliased with either input operand. This function is intended for all operand sizes. It will automatically select an appropriate algorithm out of the following:

- A hardcoded multiplication function for small sizes.
- Karatsuba or Toom-Cook multiplication for intermediate sizes.
- FFT multiplication for huge sizes.
- A GMP fallback for cases where we do currently not have optimized code.

`void flint_mpn_mul_n(mp_ptr z, mp_srcptr x, mp_srcptr y, mp_size_t n)`

Sets z to the product of (x, n) and (y, n) . We require $n \geq 1$ and that z is not aliased with either input operand. The algorithm selection is similar to `flint_mpn_mul()`.

`void flint_mpn_sqr(mp_ptr z, mp_srcptr x, mp_size_t n)`

Sets z to the square of (x, n) . We require $n \geq 1$ and that z is not aliased with the input operand. The algorithm selection is similar to `flint_mpn_sqr()`.

`mp_size_t flint_mpn_fmms1(mp_ptr y, mp_limb_t a1, mp_srcptr x1, mp_limb_t a2, mp_srcptr x2, mp_size_t n)`

Given not-necessarily-normalized x_1 and x_2 of length $n > 0$ and output y of length n , try to compute $y = a_1 \cdot x_1 - a_2 \cdot x_2$. Return the normalized length of y if $y \geq 0$ and y fits into n limbs. Otherwise, return -1 . y may alias x_1 but is not allowed to alias x_2 .

```
void flint_mpn_mul_toom22(mp_ptr pp, mp_srcptr ap, mp_size_t an, mp_srcptr bp, mp_size_t
                        bn, mp_ptr scratch)
```

Toom-22 (Karatsuba) multiplication. The *scratch* space must have room for $2an + k$ limbs where k is the number of limbs. If *NULL* is passed, space will be allocated internally.

4.14.5 Truncating multiplication

Given two n -limb integers, a *high product* (or *mulhigh*) is an approximation of the leading n limbs of the full $2n$ -limb product. In the basecase regime, a high product can be computed in roughly half the time of the full product, and in some fraction $0.5 < c < 1$ of the time in the Toom-Cook regime. This speedup vanishes asymptotically in the FFT regime. Contrary to polynomial high products or integer low products, integer high products are not uniquely defined due to carry propagation. We make the following definitions:

- *Rough mulhigh* accumulates at least $n + 1$ limbs of partial products, outputting n limbs where the $n - 1$ most significant limbs are essentially correct and the n -th most significant limb may have an error of $O(n)$ ulp. This is the version of mulhigh used in [HZ2011].
- *Precise mulhigh* accumulates at least $n + 2$ limbs of partial products, outputting $n + 1$ limbs where the n most significant limbs are essentially correct and the $(n + 1)$ -th most significant limb may have an error of $O(n)$ ulp.
- *Exact mulhigh* is the exact truncation of the full product. This cannot be computed faster than the full product in the worst case, but it can be computed faster on average by performing a precise mulhigh, inspecting the low output limb, and correcting with a low product when necessary.

In all cases, a high product is either equal to or smaller than the high part of the full product.

More generally, we can define n -limb high products of m -limb and p -limb integers where $m + p > n$, but this is not currently implemented.

```
void _flint_mpn_mulhigh_n_mulders_recursive(mp_ptr res, mp_srcptr u, mp_srcptr v,
                                           mp_size_t n)
```

```
void _flint_mpn_sqrhigh_mulders_recursive(mp_ptr res, mp_srcptr u, mp_size_t n)
```

Rough mulhigh implemented using Mulders' recursive algorithm as described in [HZ2011]. Puts in $res[n], \dots, res[2n-1]$ an approximation of the n high limbs of $\{u, n\}$ times $\{v, n\}$. The error is less than n ulps of $res[n]$. Assumes $2n$ limbs are allocated at *res*; the low limbs will be used as scratch space. The *sqrhigh* version implements squaring.

```
mp_limb_t flint_mpn_mulhigh_basecase(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

```
mp_limb_t flint_mpn_mulhigh_n_mulders(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

```
mp_limb_t flint_mpn_mulhigh_n_mul(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

```
mp_limb_t flint_mpn_mulhigh_n(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

Precise mulhigh. Puts in $res[0], \dots, res[n-1]$ an approximation of the n high limbs of $\{u, n\}$ times $\{v, n\}$. and returns the $(n + 1)$ -th most significant limb. The error is at most $n + 2$ ulp in the returned limb.

- The *basecase* version implements the $O(n^2)$ schoolbook algorithm. On x86-64 machines with ADX, the basecase version currently assumes that $n \geq 6$.
- The *mulders* version computes a rough mulhigh with one extra limb of precision in temporary scratch space using `_flint_mpn_mulhigh_n_mulders_recursive()` and then copies the high limbs to the output.
- The *mul* version computes a full product in temporary scratch space and copies the high limbs to the output. The output is actually the exact mulhigh.
- The default version looks up a hardcoded basecase multiplication routine in a table for small n , and otherwise calls the *basecase*, *mulders* or *mul* implementations.

```
mp_limb_t flint_mpn_sqrhigh_basecase(mp_ptr res, mp_srcptr u, mp_size_t n)
```

```
mp_limb_t flint_mpn_sqrhigh_mulders(mp_ptr res, mp_srcptr u, mp_size_t n)
mp_limb_t flint_mpn_sqrhigh_sqr(mp_ptr res, mp_srcptr u, mp_size_t n)
mp_limb_t flint_mpn_sqrhigh(mp_ptr res, mp_srcptr u, mp_size_t n)
```

Squaring counterparts of *flint_mpn_mulhigh_n()*.

On x86-64 machines with ADX, the basecase version currently assumes that $n \geq 8$.

```
void flint_mpn_mullow_n_mulders_recursive(mp_ptr rp, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

```
mp_limb_t flint_mpn_mullow_basecase(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
mp_limb_t flint_mpn_mullow_n_mulders(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
mp_limb_t flint_mpn_mullow_n_mul(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
mp_limb_t flint_mpn_mullow_n(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
mp_limb_t flint_mpn_mullow_n(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

Compute the low n limbs of the product.

The $(n + 1)$ -th limb is also computed and returned. Warning: this extra limb of output may be removed in the future.

```
void flint_mpn_mul_or_mullow_n(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

Write the low $n + 1$ limbs of the product uv to **res**. The output is assumed to have space for $2n$ limbs so that the high limbs can be used as scratch space or to write the whole product when this is the fastest method.

Warning: the one extra limb of output may be removed in the future.

```
void flint_mpn_mul_or_mulhigh_n(mp_ptr res, mp_srcptr u, mp_srcptr v, mp_size_t n)
```

Write the high $n + 1$ limbs of the product uv to **res** + $(n - 1)$ (with possible error of a few ulps as for *flint_mpn_mulhigh_n()*). The low $n - 1$ limbs of the output may be used as scratch space or to write the whole product when this is the fastest method.

4.14.6 Divisibility

```
int flint_mpn_divisible_1_odd(mp_srcptr x, mp_size_t xsize, mp_limb_t d)
```

Expression determining whether $(x, xsize)$ is divisible by the `mp_limb_t` d which is assumed to be odd-valued and at least 3.

This function is implemented as a macro.

```
mp_size_t flint_mpn_remove_2exp(mp_ptr x, mp_size_t xsize, flint_bitcnt_t *bits)
```

Divides $(x, xsize)$ by 2^n where n is the number of trailing zero bits in x . The new size of x is returned, and n is stored in the `bits` argument. x may not be zero.

```
mp_size_t flint_mpn_remove_power_ascending(mp_ptr x, mp_size_t xsize, mp_ptr p, mp_size_t psize, ulong *exp)
```

Divides $(x, xsize)$ by the largest power n of $(p, psize)$ that is an exact divisor of x . The new size of x is returned, and n is stored in the `exp` argument. x may not be zero, and p must be greater than 2.

This function works by testing divisibility by ascending squares $p, p^2, p^4, p^8, \dots$, making it efficient for removing potentially large powers. Because of its high overhead, it should not be used as the first stage of trial division.

```
int flint_mpn_factor_trial(mp_srcptr x, mp_size_t xsize, slong start, slong stop)
```

Searches for a factor of $(x, xsize)$ among the primes in positions `start`, ..., `stop-1` of `flint_primes`. Returns i if `flint_primes[i]` is a factor, otherwise returns 0 if no factor is found. It is assumed that `start` ≥ 1 .

```
int flint_mpn_factor_trial_tree(slong *factors, mp_srcptr x, mp_size_t xsize, slong
                               num_primes)
```

Searches for a factor of $(x, xsize)$ among the primes in positions approximately in the range $0, \dots, \text{num\_primes} - 1$ of `flint_primes`.

Returns the number of prime factors found and fills `factors` with their indices in `flint_primes`. It is assumed that `num_primes` is in the range $0, \dots, 3512$.

If the input fits in a small `fmpz` the number is fully factored instead.

The algorithm used is a tree based gcd with a product of primes, the tree for which is cached globally (it is threadsafe).

4.14.7 Division

```
void flint_mpn_signed_div2(mp_ptr res, mp_srcptr x, mp_size_t n)
```

Sets `res` to (x, n) divided by two, where x is viewed as a signed integer in two's complement form.

```
int flint_mpn_divides(mp_ptr q, mp_srcptr array1, mp_size_t limbs1, mp_srcptr arrayg,
                     mp_size_t limbsg, mp_ptr temp)
```

If $(arrayg, limbsg)$ divides $(array1, limbs1)$ then $(q, limbs1 - limbsg + 1)$ is set to the quotient and 1 is returned, otherwise 0 is returned. The temporary space `temp` must have space for `limbsg` limbs.

Assumes `limbs1 >= limbsg > 0`.

4.14.8 Division and modular arithmetic with precomputed inverses

```
mp_limb_t flint_mpn_preinv1(mp_limb_t d, mp_limb_t d2)
```

Computes a precomputed inverse from the leading two limbs of the divisor `b, n` to be used with the `preinv1` functions. We require the most significant bit of `b, n` to be 1.

```
mp_limb_t flint_mpn_divrem_preinv1(mp_ptr q, mp_ptr a, mp_size_t m, mp_srcptr b,
                                   mp_size_t n, mp_limb_t dinv)
```

Divide `a, m` by `b, n`, returning the high limb of the quotient (which will either be 0 or 1), storing the remainder in-place in `a, n` and the rest of the quotient in `q, m - n`. We require the most significant bit of `b, n` to be 1. `dinv` must be computed from `b[n - 1], b[n - 2]` by `flint_mpn_preinv1`. We also require `m >= n >= 2`.

```
mp_limb_t flint_mpn_divrem_1_preinv(mp_ptr q, mp_srcptr a, mp_size_t n, mp_limb_t d,
                                   mp_limb_t dinv, unsigned int norm)
```

Divide `a, n` by the limb `d`, writing the quotient to `q, n` and returning the remainder. Requires `n` and `d` to be positive. Allows `a` and `q` to be aliased. Requires a single-limb inverse `dinv` precomputed by `n_preinvert_limb()` and the number of leading zero bits of `d` as `norm`.

This is equivalent to `mpn_divrem_1(q, 0, a, n, d)` but faster for small `n`. Typically `mpn_divrem_1` will be faster for large `n` as it has dedicated assembly code on many architectures whereas `flint_mpn_divrem_1_preinv` currently does not.

```
mp_limb_t flint_mpn_divrem_2_1_preinv_norm(mp_ptr qp, mp_srcptr up, mp_limb_t d,
                                           mp_limb_t dinv)
```

```
mp_limb_t flint_mpn_divrem_2_1_preinv_unnorm(mp_ptr qp, mp_srcptr up, mp_limb_t d,
                                              mp_limb_t dinv, unsigned int norm)
```

```
mp_limb_t flint_mpn_divrem_3_1_preinv_norm(mp_ptr qp, mp_srcptr up, mp_limb_t d,
                                           mp_limb_t dinv)
```

```
mp_limb_t flint_mpn_divrem_3_1_preinvn_unnorm(mp_ptr qp, mp_srcptr up, mp_limb_t d,
                                              mp_limb_t dinv, unsigned int norm)
```

Versions of `flint_mpn_divrem_1_preinvn()` specialized for length 2 and 3. The `_norm` functions require a normalised divisor while the `_unnorm` functions require an unnormalised divisor with positive `norm`.

```
void flint_mpn_mulmod_preinvn1(mp_ptr r, mp_srcptr a, mp_srcptr b, mp_size_t n, mp_srcptr d,
                               mp_limb_t dinv, ulong norm)
```

Given a normalised integer d with precomputed inverse `dinv` provided by `flint_mpn_preinvn1`, computes $ab \pmod{d}$ and stores the result in r . Each of a , b and r is expected to have n limbs of space, with zero padding if necessary.

The value `norm` is provided for convenience. If a , b and d have been shifted left by `norm` bits so that d is normalised, then r will be shifted right by `norm` bits so that it has the same shift as all the inputs.

We require a and b to be reduced modulo n before calling the function.

```
void flint_mpn_preinvn(mp_ptr dinv, mp_srcptr d, mp_size_t n)
```

Compute an n limb precomputed inverse `dinv` of the n limb integer d .

We require that d is normalised, i.e. with the most significant bit of the most significant limb set.

```
void flint_mpn_mod_preinvn(mp_ptr r, mp_srcptr a, mp_size_t m, mp_srcptr d, mp_size_t n,
                          mp_srcptr dinv)
```

Given a normalised integer d of n limbs, with precomputed inverse `dinv` provided by `flint_mpn_preinvn` and integer a of m limbs, computes $a \pmod{d}$ and stores the result in-place in the lower n limbs of a . The remaining limbs of a are destroyed.

We require $m \geq n$. No aliasing of a with any of the other operands is permitted.

Note that this function is not always as fast as ordinary division.

```
mp_limb_t flint_mpn_divrem_preinvn(mp_ptr q, mp_ptr r, mp_srcptr a, mp_size_t m,
                                   mp_srcptr d, mp_size_t n, mp_srcptr dinv)
```

Given a normalised integer d with precomputed inverse `dinv` provided by `flint_mpn_preinvn`, computes the quotient of a by d and stores the result in q and the remainder in the lower n limbs of a . The remaining limbs of a are destroyed.

The value q is expected to have space for $m - n$ limbs and we require $m \geq n$. No aliasing is permitted between q and a or between these and any of the other operands.

Note that this function is not always as fast as ordinary division.

```
void flint_mpn_mulmod_preinvn2(mp_ptr r, mp_srcptr a, mp_srcptr b, mp_size_t n, mp_srcptr d,
                              mp_srcptr dinv, ulong norm)
```

Given a normalised integer d with precomputed inverse `dinv` provided by `flint_mpn_preinvn`, computes $ab \pmod{d}$ and stores the result in r . Each of a , b and r is expected to have n limbs of space, with zero padding if necessary.

The value `norm` is provided for convenience. If a , b and d have been shifted left by `norm` bits so that d is normalised, then r will be shifted right by `norm` bits so that it has the same shift as all the inputs.

We require a and b to be reduced modulo d before calling the function.

```
void flint_mpn_mulmod_preinvn_2(mp_ptr r, mp_srcptr a, mp_srcptr b, mp_srcptr d, mp_srcptr
                                dinv, ulong norm)
```

Version of `flint_mpn_mulmod_preinvn1()` specialized for two limbs. The behavior is not exactly the same: a and b are assumed to be unshifted, and the output is unshifted.

```
void flint_mpn_fmmod_preinvn(mp_ptr r, mp_srcptr a1, mp_srcptr b1, mp_srcptr a2, mp_srcptr
                             b2, mp_size_t n, mp_srcptr dnormed, mp_srcptr dinv, ulong
                             norm)
```



```
void flint_mpn_fmmod_preinvn_2(mp_ptr r, mp_srcptr a1, mp_srcptr b1, mp_srcptr a2,
                             mp_srcptr b2, mp_srcptr dnormed, mp_srcptr dinv, ulong
                             norm)
```

Given `dnormed` containing a normalised integer $d2^{norm}$ with precomputed inverse `dinv` provided by `flint_mpn_preinvn`, computes $a_1b_1 + a_2b_2 \pmod{d}$. We require all operands to be reduced modulo d .

4.14.9 Preconditioned modular multiplication

Currently two algorithms are implemented for preconditioned multiplication: Shoup multiplication and the matrix algorithm. An FFT variant may be added in the future.

```
int flint_mpn_mulmod_want_precond(mp_size_t n, slong num, ulong norm)
```

Assuming a precision of n limbs and that one wants to perform num multiplications with a fixed (preconditioned) operand with norm `norm`, return one of the following constants indicating which algorithm is better (accounting for the cost of pretransforming the operand).

- `MPN_MULMOD_PRECOND_NONE` - should use `flint_mpn_mulmod_preinvn()` (no precomputation)
- `MPN_MULMOD_PRECOND_SHOUP` - should use `flint_mpn_mulmod_precond_shoup()`
- `MPN_MULMOD_PRECOND_MATRIX` - should use `flint_mpn_mulmod_precond_matrix()`

```
void flint_mpn_mulmod_precond_shoup(mp_ptr res, mp_srcptr a, mp_srcptr apre, mp_srcptr b,
                                   mp_size_t n, mp_srcptr d, ulong norm)
```

Compute $ab \pmod{d}$ given precomputed data for `apre` generated with `flint_mpn_mulmod_precond_shoup_precompute()`. We require that b is reduced modulo d .

```
void flint_mpn_mulmod_precond_shoup_precompute(mp_ptr apre, mp_srcptr a, mp_size_t n,
                                              mp_srcptr dnormed, mp_srcptr dinv, ulong
                                              norm)
```

Given $0 \leq a < d$, precompute data for multiplication by a modulo d using Shoup's method. The modulus is given as `dnormed` containing $d2^{norm}$ together with precomputed inverse `dinv`. The destination `apre` must have space for n limbs.

```
void flint_mpn_mulmod_precond_matrix(mp_ptr rp, mp_srcptr apre, mp_srcptr b, mp_size_t n,
                                    mp_srcptr dnormed, mp_srcptr dinv, ulong norm)
```

Given `dnormed` containing a normalised integer $d2^{norm}$ with precomputed inverse `dinv` provided by `flint_mpn_preinvn()`, computes $ab \pmod{d}$. We require b to be reduced modulo d . The user provides the operand a via the `apre` argument in the pretransformed representation returned by `flint_mpn_mulmod_precond_matrix_precompute()`. The complexity of this function is $O(n^2)$. Requires $n \geq 2$.

```
void flint_mpn_mulmod_precond_matrix_precompute(mp_ptr apre, mp_srcptr a, mp_size_t n,
                                              mp_srcptr dnormed, mp_srcptr dinv, ulong
                                              norm)
```

Given `dnormed` containing a normalised integer $d2^{norm}$ with precomputed inverse `dinv` and an integer a which is reduced modulo d , write to `apre` a pretransformed representation of a for use with `flint_mpn_mulmod_precond_matrix()`. Currently, the output consists of $n \times n$ limbs storing $a2^{norm}\beta^i \pmod{d2^{norm}}$ for $0 \leq i < n$ where β is the limb radix, plus one junk limb.

```
mp_size_t flint_mpn_mulmod_precond_matrix_alloc(mp_size_t n)
```

The `alloc` function returns the number of limbs of space required for `flint_mpn_mulmod_precond_matrix_precompute()` given a modulus with n limbs.

```
void flint_mpn_fmmod_precond_matrix(mp_ptr rp, mp_srcptr alpre, mp_srcptr b1, mp_srcptr
                                   a2pre, mp_srcptr b2, mp_size_t n, mp_srcptr dnormed,
                                   mp_srcptr dinv, ulong norm)
```

Analogous to `flint_mpn_mulmod_precond_matrix()`, but computes $a_1b_1 + a_2b_2 \pmod{d}$.

4.14.10 GCD

`mp_size_t flint_mpn_gcd_full2(mp_ptr arrayg, mp_srcptr array1, mp_size_t limbs1, mp_srcptr array2, mp_size_t limbs2, mp_ptr temp)`

Sets (`arrayg`, `retvalue`) to the gcd of (`array1`, `limbs1`) and (`array2`, `limbs2`).

The only assumption is that neither `limbs1` nor `limbs2` is zero.

The function must be supplied with `limbs1 + limbs2` limbs of temporary space, or `NULL` must be passed to `temp` if the function should allocate its own space.

`mp_size_t flint_mpn_gcd_full(mp_ptr arrayg, mp_srcptr array1, mp_size_t limbs1, mp_srcptr array2, mp_size_t limbs2)`

Sets (`arrayg`, `retvalue`) to the gcd of (`array1`, `limbs1`) and (`array2`, `limbs2`).

The only assumption is that neither `limbs1` nor `limbs2` is zero.

4.14.11 Random Number Generation

`void flint_mpn_urandomb(mp_ptr rp, flint_rand_t state, flint_bitcnt_t n)`

Generates a uniform random number of `n` bits and stores it on `rp`.

`void flint_mpn_urandomm(mp_ptr rp, flint_rand_t state, mp_srcptr xp, mp_size_t xn)`

Generates a uniform random number between 0 inclusive and (`xp`, `xn`) exclusive `[0, x)` and stores it on `rp`. The most significant limb of `xp` is required to be nonzero. This function will write `xn` limbs to `rp` even if the largest possible value has one fewer limb.

`void flint_mpn_rrandom(mp_ptr rp, flint_rand_t state, mp_size_t n)`

Generates a random number with `n` limbs and stores it on `rp`. The number it generates will tend to have long strings of zeros and ones in the binary representation.

Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs.

`void flint_mpn_rrandomb(mp_ptr rp, flint_rand_t state, flint_bitcnt_t nbits)`

Generates a random number with `nbits` bits and stores it on `rp`. The number it generates will tend to have long strings of zeros and ones in the binary representation.

4.15 aprcl.h – APRCL primality testing

This module implements the rigorous APRCL primality test, suitable for integers up to a few thousand digits.

The APR-CL test uses the Jacobi sums that belong to $\mathbb{Z}[\zeta]/(n)$, so we have `unity_zp` struct and some useful operations. `unity_zp` is just a wrapper over `fmpz_mod_poly` with additional fields.

Also provides Gauss sum test, which is not very useful in practice, but can be useful for people who want to see an implementation of these. Gauss sums belong $\mathbb{Z}[\zeta_q, \zeta_p]/(n)$ and implemented in `unity_zpq` struct.

Authors:

- Vladimir Glazachev (Google Summer of Code, 2015)

4.15.1 Primality test functions

int **aprcl_is_prime**(const *fmpz_t* n)

Tests n for primality using the APRCL test. This is the same as *aprcl_is_prime_jacobi()*.

int **aprcl_is_prime_jacobi**(const *fmpz_t* n)

If n is prime returns 1; otherwise returns 0. The algorithm is well described in “Implementation of a New Primality Test” by H. Cohen and A.K. Lenstra and “A Course in Computational Algebraic Number Theory” by H. Cohen.

It is theoretically possible that this function fails to prove that n is prime. In this event, *flint_abort()* is called. To handle this condition, the *_aprcl_is_prime_jacobi()* function can be used.

int **aprcl_is_prime_gauss**(const *fmpz_t* n)

If n is prime returns 1; otherwise returns 0. Uses the cyclotomic primality testing algorithm described in “Four primality testing algorithms” by Rene Schoof. The minimum required numbers s and R are computed automatically.

By default $R \geq 180$. In some cases this function fails to prove that n is prime. This means that we select a too small R value. In this event, *flint_abort()* is called. To handle this condition, the *_aprcl_is_prime_jacobi()* function can be used.

primality_test_status **_aprcl_is_prime_jacobi**(const *fmpz_t* n, const *aprcl_config* config)

Jacobi sum test for n . Possible return values: PRIME, COMPOSITE and UNKNOWN (if we cannot prove primality).

primality_test_status **_aprcl_is_prime_gauss**(const *fmpz_t* n, const *aprcl_config* config)

Tests n for primality with fixed *config*. Possible return values: PRIME, COMPOSITE and PROBABPRIME (if we cannot prove primality).

int **aprcl_is_prime_gauss_min_R**(const *fmpz_t* n, *ulong* R)

Same as *aprcl_is_prime_gauss()* with fixed minimum value of R .

int **aprcl_is_prime_final_division**(const *fmpz_t* n, const *fmpz_t* s, *ulong* r)

Returns 0 if for some $a = n^k \bmod s$, where $k \in [1, r - 1]$, we have that $a \mid n$; otherwise returns 1.

4.15.2 Configuration functions

type **_aprcl_config**

type **aprcl_config**

Holds precomputed parameters.

void **aprcl_config_gauss_init**(*aprcl_config* conf, const *fmpz_t* n)

Computes the s and R values used in the cyclotomic primality test, $s^2 > n$ and $s = \prod_{\substack{q-1 \mid R \\ q \text{ prime}}} q$. Also stores factors of R and s .

void **aprcl_config_gauss_init_min_R**(*aprcl_config* conf, const *fmpz_t* n, *ulong* R)

Computes the s with fixed minimum R such that $a^R \equiv 1 \pmod s$ for all integers a coprime to s .

void **aprcl_config_gauss_clear**(*aprcl_config* conf)

Clears the given *aprcl_config* element. It must be reinitialised in order to be used again.

ulong **aprcl_R_value**(const *fmpz_t* n)

Returns a precomputed R value for APRCL, such that the corresponding s value is greater than $\sqrt{n}$. The maximum stored value 6983776800 allows to test numbers up to 6000 digits.

void **aprcl_config_jacobi_init**(*aprcl_config* conf, const *fmpz_t* n)

Computes the s and R values used in the cyclotomic primality test, $s^2 > n$ and $a^R \equiv 1 \pmod s$ for all a coprime to s . Also stores factors of R and s .

void **aprcl_config_jacobi_clear**(*aprcl_config* conf)

Clears the given **aprcl_config** element. It must be reinitialised in order to be used again.

4.15.3 Cyclotomic arithmetic

This code implements arithmetic in cyclotomic rings.

Types

type **_unity_zp**

type **unity_zp**

Represents an element of $\mathbb{Z}[\zeta_{p^{\text{exp}}}]/(n)$ as an *fmpz_mod_poly_t* reduced modulo a cyclotomic polynomial.

type **_unity_zpq**

type **unity_zpq**

Represents an element of $\mathbb{Z}[\zeta_q, \zeta_p]/(n)$ as an array of *fmpz_mod_poly_t*.

Memory management

void **unity_zp_init**(*unity_zp* f, *ulong* p, *ulong* exp, const *fmpz_t* n)

Initializes f as an element of $\mathbb{Z}[\zeta_{p^{\text{exp}}}]/(n)$.

void **unity_zp_clear**(*unity_zp* f)

Clears the given element. It must be reinitialised in order to be used again.

void **unity_zp_copy**(*unity_zp* f, const *unity_zp* g)

Sets f to g . f and g must be initialized with same p and n .

void **unity_zp_swap**(*unity_zp* f, *unity_zp* q)

Swaps f and g . f and g must be initialized with same p and n .

void **unity_zp_set_zero**(*unity_zp* f)

Sets f to zero.

Comparison

slong **unity_zp_is_unity**(*unity_zp* f)

If $f = \zeta^h$ returns h ; otherwise returns -1.

int **unity_zp_equal**(*unity_zp* f, *unity_zp* g)

Returns nonzero if $f = g$ reduced by the p^{exp} -th cyclotomic polynomial.

Coefficient management

void **unity_zp_coeff_set_fmpz**(*unity_zp* f, *ulong* ind, const *fmpz_t* x)

void **unity_zp_coeff_set_ui**(*unity_zp* f, *ulong* ind, *ulong* x)

Sets the coefficient of ζ^{ind} to x . ind must be less than p^{exp} .

void **unity_zp_coeff_add_fmpz**(*unity_zp* f, *ulong* ind, const *fmpz_t* x)

void **unity_zp_coeff_add_ui**(*unity_zp* f, *ulong* ind, *ulong* x)

Adds x to the coefficient of ζ^{ind} . x must be less than n . ind must be less than p^{exp} .

void **unity_zp_coeff_inc**(*unity_zp* f, *ulong* ind)

Increments the coefficient of ζ^{ind} . ind must be less than p^{exp} .

void **unity_zp_coeff_dec**(*unity_zp* f, *ulong* ind)

Decrements the coefficient of ζ^{ind} . ind must be less than p^{exp} .

Scalar multiplication

void **unity_zp_mul_scalar_ui**(*unity_zp* f, const *unity_zp* g, *ulong* s)

Sets f to $s \cdot g$. f and g must be initialized with same p , exp and n .

Addition and multiplication

void **unity_zp_add**(*unity_zp* f, const *unity_zp* g, const *unity_zp* h)

Sets f to $g + h$. f , g and h must be initialized with same p , exp and n .

void **unity_zp_mul**(*unity_zp* f, const *unity_zp* g, const *unity_zp* h)

Sets f to $g \cdot h$. f , g and h must be initialized with same p , exp and n .

void **unity_zp_sqr**(*unity_zp* f, const *unity_zp* g)

Sets f to $g \cdot g$. f , g and h must be initialized with same p , exp and n .

void **unity_zp_mul_inplace**(*unity_zp* f, const *unity_zp* g, const *unity_zp* h, *fmpz_t* *t)

Sets f to $g \cdot h$. If $p^{exp} = 3, 4, 5, 7, 8, 9, 11, 16$ special multiplication functions are used. The preallocated array t of *fmpz_t* is used for all computations in this case. f , g and h must be initialized with same p , exp and n .

void **unity_zp_sqr_inplace**(*unity_zp* f, const *unity_zp* g, *fmpz_t* *t)

Sets f to $g \cdot g$. If $p^{exp} = 3, 4, 5, 7, 8, 9, 11, 16$ special multiplication functions are used. The preallocated array t of *fmpz_t* is used for all computations in this case. f and g must be initialized with same p , exp and n .

Powering functions

void **unity_zp_pow_fmpz**(*unity_zp* f, const *unity_zp* g, const *fmpz_t* pow)

Sets f to g^{pow} . f and g must be initialized with same p , exp and n .

void **unity_zp_pow_ui**(*unity_zp* f, const *unity_zp* g, *ulong* pow)

Sets f to g^{pow} . f and g must be initialized with same p , exp and n .

ulong **unity_zp_pow_select_k**(const *fmpz_t* n)

Returns the smallest integer k satisfying $\log(n) < (k(k+1)2^{2k})/(2^{k+1} - k - 2) + 1$

void **unity_zp_pow_2k_fmpz**(*unity_zp* f, const *unity_zp* g, const *fmpz_t* pow)

Sets f to g^{pow} using the 2^k -ary exponentiation method. f and g must be initialized with same p , exp and n .

void **unity_zp_pow_2k_ui**(*unity_zp* f, const *unity_zp* g, *ulong* pow)

Sets f to g^{pow} using the 2^k -ary exponentiation method. f and g must be initialized with same p , exp and n .

void **unity_zp_pow_sliding_fmpz**(*unity_zp* f, *unity_zp* g, const *fmpz_t* pow)

Sets f to g^{pow} using the sliding window exponentiation method. f and g must be initialized with same p , exp and n .

Cyclotomic reduction

void **_unity_zp_reduce_cyclotomic_divmod**(*unity_zp* f)

void **_unity_zp_reduce_cyclotomic**(*unity_zp* f)

Sets $f = f \bmod \Phi_{p^{exp}}$. $\Phi_{p^{exp}}$ is the p^{exp} -th cyclotomic polynomial. g must be reduced by $x^{p^{exp}} - 1$ poly. f and g must be initialized with same p , exp and n .

void **unity_zp_reduce_cyclotomic**(*unity_zp* f, const *unity_zp* g)

Sets $f = g \bmod \Phi_{p^{exp}}$. $\Phi_{p^{exp}}$ is the p^{exp} -th cyclotomic polynomial.

Automorphism and inverse

void **unity_zp_aut**(*unity_zp* f, const *unity_zp* g, *ulong* x)

Sets $f = \sigma_x(g)$, the automorphism $\sigma_x(\zeta) = \zeta^x$. f and g must be initialized with the same p , exp and n .

void **unity_zp_aut_inv**(*unity_zp* f, const *unity_zp* g, *ulong* x)

Sets $f = \sigma_x^{-1}(g)$, so $\sigma_x(f) = g$. g must be reduced by $\Phi_{p^{exp}}$. f and g must be initialized with the same p , exp and n .

Jacobi sum

Here $\chi_{p,q}$ is the character defined by $\chi_{p,q}(g^x) = \zeta_{p^k}^x$, where g is a primitive root modulo q .

void **unity_zp_jacobi_sum_pq**(*unity_zp* f, *ulong* q, *ulong* p)

Sets f to the Jacobi sum $J(p, q) = j(\chi_{p,q}, \chi_{p,q})$.

void **unity_zp_jacobi_sum_2q_one**(*unity_zp* f, *ulong* q)

Sets f to the Jacobi sum $J_2(q) = j(\chi_{2,q}^{2^{k-3}}, \chi_{2,q}^{3 \cdot 2^{k-3}})^2$.

void **unity_zp_jacobi_sum_2q_two**(*unity_zp* f, *ulong* q)

Sets f to the Jacobi sum $J_3(1) = j(\chi_{2,q}, \chi_{2,q}, \chi_{2,q}) = J(2, q) \cdot j(\chi_{2,q}^2, \chi_{2,q})$.

Extended rings

void **unity_zpq_init**(*unity_zpq* f, *ulong* q, *ulong* p, const *fmpz_t* n)

Initializes f as an element of $\mathbb{Z}[\zeta_q, \zeta_p]/(n)$.

void **unity_zpq_clear**(*unity_zpq* f)

Clears the given element. It must be reinitialized in order to be used again.

void **unity_zpq_copy**(*unity_zpq* f, const *unity_zpq* g)

Sets f to g . f and g must be initialized with same p , q and n .

void **unity_zpq_swap**(*unity_zpq* f, *unity_zpq* q)

Swaps f and g . f and g must be initialized with same p , q and n .

```

int unity_zpq_equal(const unity_zpq f, const unity_zpq g)
    Returns nonzero if  $f = g$ .

void unity_zpq_coeff_set_fmpz(unity_zpq f, slong i, slong j, const fmpz_t x)
    Sets the coefficient of  $\zeta_q^i \zeta_p^j$  to  $x$ .  $i$  must be less than  $q$  and  $j$  must be less than  $p$ .

void unity_zpq_coeff_set_ui(unity_zpq f, slong i, slong j, ulong x)
    Sets the coefficient of  $\zeta_q^i \zeta_p^j$  to  $x$ .  $i$  must be less than  $q$  and  $j$  must be less than  $p$ .

void unity_zpq_coeff_add(unity_zpq f, slong i, slong j, const fmpz_t x)
    Adds  $x$  to the coefficient of  $\zeta_q^i \zeta_p^j$ .  $x$  must be less than  $n$ .

void unity_zpq_add(unity_zpq f, const unity_zpq g, const unity_zpq h)
    Sets  $f$  to  $g + h$ .  $f$ ,  $g$  and  $h$  must be initialized with same  $q$ ,  $p$  and  $n$ .

void unity_zpq_mul(unity_zpq f, const unity_zpq g, const unity_zpq h)
    Sets the  $f$  to  $g \cdot h$ .  $f$ ,  $g$  and  $h$  must be initialized with same  $q$ ,  $p$  and  $n$ .

void _unity_zpq_mul_unity_p(unity_zpq f)
    Sets  $f = f \cdot \zeta_p$ .

void unity_zpq_mul_unity_p_pow(unity_zpq f, const unity_zpq g, slong k)
    Sets  $f$  to  $g \cdot \zeta_p^k$ .

void unity_zpq_pow(unity_zpq f, const unity_zpq g, const fmpz_t p)
    Sets  $f$  to  $g^p$ .  $f$  and  $g$  must be initialized with same  $p$ ,  $q$  and  $n$ .

void unity_zpq_pow_ui(unity_zpq f, const unity_zpq g, ulong p)
    Sets  $f$  to  $g^p$ .  $f$  and  $g$  must be initialized with same  $p$ ,  $q$  and  $n$ .

void unity_zpq_gauss_sum(unity_zpq f, ulong q, ulong p)
    Sets  $f = \tau(\chi_{p,q})$ .

void unity_zpq_gauss_sum_sigma_pow(unity_zpq f, ulong q, ulong p)
    Sets  $f = \tau^{\sigma_n}(\chi_{p,q})$ .

```

4.16 arith.h – arithmetic and special functions

This module implements arithmetic functions, number-theoretic and combinatorial special number sequences and polynomials.

4.16.1 Harmonic numbers

```

void _arith_harmonic_number(fmpz_t num, fmpz_t den, slong n)
void arith_harmonic_number(fmpq_t x, slong n)
    These are aliases for the functions in the fmpq module.

```


4.16.2 Stirling numbers

void **arith_stirling_number_1u**(*mpz_t* s, *ulong* n, *ulong* k)

void **arith_stirling_number_1**(*mpz_t* s, *ulong* n, *ulong* k)

void **arith_stirling_number_2**(*mpz_t* s, *ulong* n, *ulong* k)

Sets s to $S(n, k)$ where $S(n, k)$ denotes an unsigned Stirling number of the first kind $|S_1(n, k)|$, a signed Stirling number of the first kind $S_1(n, k)$, or a Stirling number of the second kind $S_2(n, k)$. The Stirling numbers are defined using the generating functions

$$\begin{aligned} x_{(n)} &= \sum_{k=0}^n S_1(n, k) x^k \\ x^{(n)} &= \sum_{k=0}^n |S_1(n, k)| x^k \\ x^n &= \sum_{k=0}^n S_2(n, k) x_{(k)} \end{aligned}$$

where $x_{(n)} = x(x-1)(x-2)\cdots(x-n+1)$ is a falling factorial and $x^{(n)} = x(x+1)(x+2)\cdots(x+n-1)$ is a rising factorial. $S(n, k)$ is taken to be zero if $n < 0$ or $k < 0$.

These three functions are useful for computing isolated Stirling numbers efficiently. To compute a range of numbers, the vector or matrix versions should generally be used.

void **arith_stirling_number_1u_vec**(*mpz_t* *row, *ulong* n, *slong* klen)

void **arith_stirling_number_1_vec**(*mpz_t* *row, *ulong* n, *slong* klen)

void **arith_stirling_number_2_vec**(*mpz_t* *row, *ulong* n, *slong* klen)

Computes the row of Stirling numbers $S(n, 0)$, $S(n, 1)$, $S(n, 2)$, ..., $S(n, klen-1)$.

To compute a full row, this function can be called with $klen = n+1$. It is assumed that $klen$ is at most $n+1$.

void **arith_stirling_matrix_1u**(*mpz_mat_t* mat)

void **arith_stirling_matrix_1**(*mpz_mat_t* mat)

void **arith_stirling_matrix_2**(*mpz_mat_t* mat)

For an arbitrary m -by- n matrix, writes the truncation of the infinite Stirling number matrix:

```
row 0 : S(0,0)
row 1 : S(1,0), S(1,1)
row 2 : S(2,0), S(2,1), S(2,2)
row 3 : S(3,0), S(3,1), S(3,2), S(3,3)
```

up to row $m-1$ and column $n-1$ inclusive. The upper triangular part of the matrix is zeroed.

For any n , the S_1 and S_2 matrices thus obtained are inverses of each other.

4.16.3 Bell numbers

```
void arith_bell_number(fmpz_t b, ulong n)
void arith_bell_number_dobinski(fmpz_t res, ulong n)
void arith_bell_number_multi_mod(fmpz_t res, ulong n)
```

Sets b to the Bell number B_n , defined as the number of partitions of a set with n members. Equivalently, $B_n = \sum_{k=0}^n S_2(n, k)$ where $S_2(n, k)$ denotes a Stirling number of the second kind.

The default version automatically selects between table lookup, Dobinski's formula, and the multimodular algorithm.

The `dobinski` version evaluates a precise truncation of the series $B_n = e^{-1} \sum_{k=0}^{\infty} \frac{k^n}{k!}$ (Dobinski's formula). In fact, we compute $P = N! \sum_{k=0}^N \frac{k^n}{k!}$ and $Q = N! \sum_{k=0}^N \frac{1}{k!} \approx N!e$ and evaluate $B_n = \lceil P/Q \rceil$, avoiding the use of floating-point arithmetic.

The `multi_mod` version computes the result modulo several limb-size primes and reconstructs the integer value using the fast Chinese remainder algorithm. A bound for the number of needed primes is computed using `arith_bell_number_size`.

```
void arith_bell_number_vec(fmpz *b, slong n)
void arith_bell_number_vec_recursive(fmpz *b, slong n)
void arith_bell_number_vec_multi_mod(fmpz *b, slong n)
```

Sets b to the vector of Bell numbers $B_0, B_1, \dots, B_{n-1}$ inclusive. The `recursive` version uses the $O(n^3 \log n)$ triangular recurrence, while the `multi_mod` version implements multimodular evaluation of the exponential generating function, running in time $O(n^2 \log^{O(1)} n)$. The default version chooses an algorithm automatically.

```
ulong arith_bell_number_nmod(ulong n, nmod_t mod)
```

Computes the Bell number B_n modulo an integer given by `mod`.

After handling special cases, we use the formula

$$B_n = \sum_{k=0}^n \frac{(n-k)^n}{(n-k)!} \sum_{j=0}^k \frac{(-1)^j}{j!}.$$

We arrange the operations in such a way that we only have to multiply (and not divide) in the main loop. As a further optimisation, we use sieving to reduce the number of powers that need to be evaluated. This results in $O(n)$ memory usage.

If the divisions by factorials are impossible, we fall back to calling `arith_bell_number_nmod_vec` and reading the last coefficient.

```
void arith_bell_number_nmod_vec(nn_ptr b, slong n, nmod_t mod)
void arith_bell_number_nmod_vec_recursive(nn_ptr b, slong n, nmod_t mod)
void arith_bell_number_nmod_vec_ogf(nn_ptr b, slong n, nmod_t mod)
int arith_bell_number_nmod_vec_series(nn_ptr b, slong n, nmod_t mod)
```

Sets b to the vector of Bell numbers $B_0, B_1, \dots, B_{n-1}$ inclusive modulo an integer given by `mod`.

The `recursive` version uses the $O(n^2)$ triangular recurrence. The `ogf` version expands the ordinary generating function using binary splitting, which is $O(n \log^2 n)$.

The `series` version uses the exponential generating function $\sum_{k=0}^{\infty} \frac{B_k}{k!} x^k = \exp(e^x - 1)$, running in $O(n \log n)$. This only works if division by $n!$ is possible, and the function returns whether it is successful. All other versions support any modulus.

The default version of this function selects an algorithm automatically.

```
double arith_bell_number_size(ulong n)
```

Returns b such that $B_n < 2^{[b]}$. A previous version of this function used the inequality $B_n < \left(\frac{0.792n}{\log(n+1)}\right)^n$ which is given in [BerTas2010]; we now use a slightly better bound based on an asymptotic expansion.

4.16.4 Bernoulli numbers and polynomials

void `_arith_bernoulli_number`(*fmpz_t* num, *fmpz_t* den, *ulong* n)

Sets (num, den) to the reduced numerator and denominator of the n -th Bernoulli number.

void `arith_bernoulli_number`(*fmpq_t* x, *ulong* n)

Sets x to the n -th Bernoulli number. This function is equivalent to `_arith_bernoulli_number` apart from the output being a single `fmpq_t` variable.

void `_arith_bernoulli_number_vec`(*fmpz_t* num, *fmpz_t* den, *ulong* n)

Sets the elements of num and den to the reduced numerators and denominators of the Bernoulli numbers $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive. This function automatically chooses between the `recursive`, `zeta` and `multi_mod` algorithms according to the size of n .

void `arith_bernoulli_number_vec`(*fmpq_t* x, *ulong* n)

Sets the x to the vector of Bernoulli numbers $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive. This function is equivalent to `_arith_bernoulli_number_vec` apart from the output being a single `fmpq` vector.

void `arith_bernoulli_number_denom`(*fmpz_t* den, *ulong* n)

Sets den to the reduced denominator of the n -th Bernoulli number B_n . For even n , the denominator is computed as the product of all primes p for which $p-1$ divides n ; this property is a consequence of the von Staudt-Clausen theorem. For odd n , the denominator is trivial (den is set to 1 whenever $B_n = 0$). The initial sequence of values smaller than 2^{32} are looked up directly from a table.

double `arith_bernoulli_number_size`(*ulong* n)

Returns b such that $|B_n| < 2^{[b]}$, using the inequality $|B_n| < \frac{4n!}{(2\pi)^n}$ and $n! \leq (n+1)^{n+1}e^{-n}$. No special treatment is given to odd n . Accuracy is not guaranteed if $n > 10^{14}$.

void `arith_bernoulli_polynomial`(*fmpq_poly_t* poly, *ulong* n)

Sets poly to the Bernoulli polynomial of degree n , $B_n(x) = \sum_{k=0}^n \binom{n}{k} B_k x^{n-k}$ where B_k is a Bernoulli number. This function basically calls `arith_bernoulli_number_vec` and then rescales the coefficients efficiently.

void `_arith_bernoulli_number_vec_recursive`(*fmpz_t* num, *fmpz_t* den, *ulong* n)

Sets the elements of num and den to the reduced numerators and denominators of $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive.

The first few entries are computed using `arith_bernoulli_number`, and then Ramanujan's recursive formula expressing B_m as a sum over B_k for k congruent to m modulo 6 is applied repeatedly.

To avoid costly GCDs, the numerators are transformed internally to a common denominator and all operations are performed using integer arithmetic. This makes the algorithm fast for small n , say $n < 1000$. The common denominator is calculated directly as the primorial of $n+1$.

%[1] https://en.wikipedia.org/w/index.php?title=Bernoulli_number&oldid=405938876

void `_arith_bernoulli_number_vec_multi_mod`(*fmpz_t* num, *fmpz_t* den, *ulong* n)

Sets the elements of num and den to the reduced numerators and denominators of $B_0, B_1, B_2, \dots, B_{n-1}$ inclusive. Uses the generating function

$$\frac{x^2}{\cosh(x) - 1} = \sum_{k=0}^{\infty} \frac{(2-4k)B_{2k}}{(2k)!} x^{2k}$$

which is evaluated modulo several limb-size primes using `nmod_poly` arithmetic to yield the numerators of the Bernoulli numbers after multiplication by the denominators and CRT reconstruction. This formula, given (incorrectly) in [BuhlerCrandallSompolski1992], saves about half of the time compared to the usual generating function $x/(e^x - 1)$ since the odd terms vanish.

4.16.5 Euler numbers and polynomials

Euler numbers are the integers E_n defined by $\frac{1}{\cosh(t)} = \sum_{n=0}^{\infty} \frac{E_n}{n!} t^n$. With this convention, the odd-indexed numbers are zero and the even ones alternate signs, viz. $E_0, E_1, E_2, \dots = 1, 0, -1, 0, 5, 0, -61, 0, 1385, 0, \dots$. The corresponding Euler polynomials are defined by $\frac{2e^{xt}}{e^t+1} = \sum_{n=0}^{\infty} \frac{E_n(x)}{n!} t^n$.

void **arith_euler_number**(*fmpz_t* res, *ulong* n)

Sets **res** to the Euler number E_n .

void **arith_euler_number_vec**(*fmpz_t** res, *slong* n)

Computes the Euler numbers $E_0, E_1, \dots, E_{n-1}$ for $n \geq 0$ and stores the result in **res**, which must be an initialised **fmpz** vector of sufficient size.

This function evaluates the even-index E_k modulo several limb-size primes using the generating function and **nmod_poly** arithmetic. A tight bound for the number of needed primes is computed using **arith_euler_number_size**, and the final integer values are recovered using balanced CRT reconstruction.

double **arith_euler_number_size**(*ulong* n)

Returns b such that $|E_n| < 2^{\lfloor b \rfloor}$, using the inequality $|E_n| < \frac{2^{n+2} n!}{\pi^{n+1}}$ and $n! \leq (n+1)^{n+1} e^{-n}$. No special treatment is given to odd n . Accuracy is not guaranteed if $n > 10^{14}$.

void **arith_euler_polynomial**(*fmpz_poly_t* poly, *ulong* n)

Sets **poly** to the Euler polynomial $E_n(x)$. Uses the formula

$$E_n(x) = \frac{2}{n+1} \left(B_{n+1}(x) - 2^{n+1} B_{n+1}\left(\frac{x}{2}\right) \right),$$

with the Bernoulli polynomial $B_{n+1}(x)$ evaluated once using **bernoulli_polynomial** and then rescaled.

4.16.6 Multiplicative functions

void **arith_divisors**(*fmpz_poly_t* res, const *fmpz_t* n)

Set the coefficients of the polynomial **res** to the divisors of n , including 1 and n itself, in ascending order.

void **arith_ramanujan_tau**(*fmpz_t* res, const *fmpz_t* n)

Sets **res** to the Ramanujan tau function $\tau(n)$ which is the coefficient of q^n in the series expansion of $f(q) = q \prod_{k \geq 1} (1 - q^k)^{24}$.

We factor n and use the identity $\tau(pq) = \tau(p)\tau(q)$ along with the recursion $\tau(p^{r+1}) = \tau(p)\tau(p^r) - p^{11}\tau(p^{r-1})$ for prime powers.

The base values $\tau(p)$ are obtained using the function **arith_ramanujan_tau_series**(). Thus the speed of **arith_ramanujan_tau**() depends on the largest prime factor of n .

Future improvement: optimise this function for small n , which could be accomplished using a lookup table or by calling **arith_ramanujan_tau_series**() directly.

void **arith_ramanujan_tau_series**(*fmpz_poly_t* res, *slong* n)

Sets **res** to the polynomial with coefficients $\tau(0), \tau(1), \dots, \tau(n-1)$, giving the initial n terms in the series expansion of $f(q) = q \prod_{k \geq 1} (1 - q^k)^{24}$.

We use the theta function identity

$$f(q) = q \left(\sum_{k \geq 0} (-1)^k (2k+1) q^{k(k+1)/2} \right)^8$$

which is evaluated using three squarings. The first squaring is done directly since the polynomial is very sparse at this point.

4.16.7 Landau's function

void **arith_landau_function_vec**(*fmpr* *res, *slong* len)

Computes the first **len** values of Landau's function $g(n)$ starting with $g(0)$. Landau's function gives the largest order of an element of the symmetric group S_n .

Implements the “basic algorithm” given in [DelegliseNicolasZimmermann2009]. The running time is $O(n^{3/2}/\sqrt{\log n})$.

4.16.8 Number of partitions

void **arith_number_of_partitions_vec**(*fmpr* *res, *slong* len)

Computes first **len** values of the partition function $p(n)$ starting with $p(0)$. Uses inversion of Euler's pentagonal series.

void **arith_number_of_partitions_nmod_vec**(*nn_ptr* res, *slong* len, *nmod_t* mod)

Computes first **len** values of the partition function $p(n)$ starting with $p(0)$, modulo the modulus defined by **mod**. Uses inversion of Euler's pentagonal series.

void **trig_prod_init**(*trig_prod_t* prod)

Initializes **prod**. This is an inline function only.

void **arith_hrr_expsum_factored**(*trig_prod_t* prod, *ulong* k, *ulong* n)

Symbolically evaluates the exponential sum

$$A_k(n) = \sum_{h=0}^{k-1} \exp\left(\pi i \left[s(h, k) - \frac{2hn}{k}\right]\right)$$

appearing in the Hardy-Ramanujan-Rademacher formula, where $s(h, k)$ is a Dedekind sum.

Rather than evaluating the sum naively, we factor $A_k(n)$ into a product of cosines based on the prime factorisation of k . This process is based on the identities given in [Whiteman1956].

The special **trig_prod_t** structure **prod** represents a product of cosines of rational arguments, multiplied by an algebraic prefactor. It must be pre-initialised with **trig_prod_init**.

This function assumes that $24k$ and $24n$ do not overflow a single limb. If n is larger, it can be pre-reduced modulo k , since $A_k(n)$ only depends on the value of $n \bmod k$.

void **arith_number_of_partitions_mpfr**(*mpfr_t* x, *ulong* n)

Sets the pre-initialised MPFR variable x to the exact value of $p(n)$. The value is computed using the Hardy-Ramanujan-Rademacher formula.

The precision of x will be changed to allow $p(n)$ to be represented exactly. The interface of this function may be updated in the future to allow computing an approximation of $p(n)$ to smaller precision.

The Hardy-Ramanujan-Rademacher formula is given with error bounds in [Rademacher1937]. We evaluate it in the form

$$p(n) = \sum_{k=1}^N B_k(n) U(C/k) + R(n, N)$$

where

$$U(x) = \cosh(x) + \frac{\sinh(x)}{x}, \quad C = \frac{\pi}{6} \sqrt{24n-1}$$

$$B_k(n) = \sqrt{\frac{3}{k}} \frac{4}{24n-1} A_k(n)$$

and where $A_k(n)$ is a certain exponential sum. The remainder satisfies

$$|R(n, N)| < \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left(\frac{N}{n-1}\right)^{1/2} \sinh\left(\pi\sqrt{\frac{2}{3}} \frac{\sqrt{n}}{N}\right).$$

We choose N such that $|R(n, N)| < 0.25$, and a working precision at term k such that the absolute error of the term is expected to be less than $0.25/N$. We also use a summation variable with increased precision, essentially making additions exact. Thus the sum of errors adds up to less than 0.5, giving the correct value of $p(n)$ when rounding to the nearest integer.

The remainder estimate at step k provides an upper bound for the size of the k -th term. We add $\log_2 N$ bits to get low bits in the terms below $0.25/N$ in magnitude.

Using `arith_hrr_expsum_factored`, each $B_k(n)$ evaluation is broken down to a product of cosines of exact rational multiples of π . We transform all angles to $(0, \pi/4)$ for optimal accuracy.

Since the evaluation of each term involves only $O(\log k)$ multiplications and evaluations of trigonometric functions of small angles, the relative rounding error is at most a few bits. We therefore just add an additional $\log_2(C/k)$ bits for the $U(x)$ when x is large. The cancellation of terms in $U(x)$ is of no concern, since Rademacher's bound allows us to terminate before x becomes small.

This analysis should be performed in more detail to give a rigorous error bound, but the precision currently implemented is almost certainly sufficient, not least considering that Rademacher's remainder bound significantly overshoots the actual values.

To improve performance, we switch to doubles when the working precision becomes small enough. We also use a separate accumulator variable which gets added to the main sum periodically, in order to avoid costly updates of the full-precision result when n is large.

void `arith_number_of_partitions`(*fmpz_t* x, *ulong* n)

Sets x to $p(n)$, the number of ways that n can be written as a sum of positive integers without regard to order.

This function uses a lookup table for $n < 128$ (where $p(n) < 2^{32}$), and otherwise calls `arith_number_of_partitions_mpfr`.

4.16.9 Sums of squares

void `arith_sum_of_squares`(*fmpz_t* r, *ulong* k, const *fmpz_t* n)

Sets r to the number of ways $r_k(n)$ in which n can be represented as a sum of k squares.

If $k = 2$ or $k = 4$, we write $r_k(n)$ as a divisor sum.

Otherwise, we either recurse on k or compute the theta function expansion up to $O(x^{n+1})$ and read off the last coefficient. This is generally optimal.

void `arith_sum_of_squares_vec`(*fmpz_t** r, *ulong* k, *slong* n)

For $i = 0, 1, \dots, n-1$, sets r_i to the number of representations of i as a sum of k squares, $r_k(i)$. This effectively computes the q -expansion of $\vartheta_3(q)$ raised to the k -th power, i.e.

$$\vartheta_3^k(q) = \left(\sum_{i=-\infty}^{\infty} q^{i^2} \right)^k.$$

4.17 fft.h – Schoenhage-Strassen FFT

4.17.1 Split/combine FFT coefficients

```
mp_size_t fft_split_limbs(mp_limb_t **poly, mp_srcptr limbs, mp_size_t total_limbs,  
                           mp_size_t coeff_limbs, mp_size_t output_limbs)
```

Split an integer (**limbs**, **total_limbs**) into coefficients of length **coeff_limbs** limbs and store as the coefficients of **poly** which are assumed to have space for **output_limbs** + 1 limbs per coefficient. The coefficients of the polynomial do not need to be zeroed before calling this function, however the number of coefficients written is returned by the function and any coefficients beyond this point are not touched.

```
mp_size_t fft_split_bits(mp_limb_t **poly, mp_srcptr limbs, mp_size_t total_limbs,  
                          flint_bitcnt_t bits, mp_size_t output_limbs)
```

Split an integer (**limbs**, **total_limbs**) into coefficients of the given number of **bits** and store as the coefficients of **poly** which are assumed to have space for **output_limbs** + 1 limbs per coefficient. The coefficients of the polynomial do not need to be zeroed before calling this function, however the number of coefficients written is returned by the function and any coefficients beyond this point are not touched.

```
void fft_combine_limbs(mp_limb_t *res, mp_limb_t **poly, slong length, mp_size_t coeff_limbs,  
                       mp_size_t output_limbs, mp_size_t total_limbs)
```

Evaluate the polynomial **poly** of the given **length** at $B^{\text{coeff\_limbs}}$, where $B = 2^{\text{FLINT\_BITS}}$, and add the result to the integer (**res**, **total_limbs**) throwing away any bits that exceed the given number of limbs. The polynomial coefficients are assumed to have at least **output_limbs** limbs each, however any additional limbs are ignored.

If the integer is initially zero the result will just be the evaluation of the polynomial.

```
void fft_combine_bits(mp_limb_t *res, mp_limb_t **poly, slong length, flint_bitcnt_t bits,  
                      mp_size_t output_limbs, mp_size_t total_limbs)
```

Evaluate the polynomial **poly** of the given **length** at 2^{bits} and add the result to the integer (**res**, **total_limbs**) throwing away any bits that exceed the given number of limbs. The polynomial coefficients are assumed to have at least **output_limbs** limbs each, however any additional limbs are ignored. If the integer is initially zero the result will just be the evaluation of the polynomial.

4.17.2 Test helper functions

```
void fermat_to_mpz(mpz_t m, mp_limb_t *i, mp_size_t limbs)
```

Convert the Fermat number (**i**, **limbs**) modulo $B^{\text{limbs}} + 1$ to an **mpz_t** **m**. Assumes **m** has been initialised. This function is used only in test code.

4.17.3 Arithmetic modulo a generalised Fermat number

```
void mpn_negmod_2expp1(mp_limb_t *z, const mp_limb_t *a, mp_size_t limbs)
```

Set **z** to the negation of the Fermat number **a** modulo $B^{\text{limbs}} + 1$. The input **a** is expected to be fully reduced, and the output is fully reduced. Aliasing is permitted.

```
void mpn_addmod_2expp1_1(mp_limb_t *r, mp_size_t limbs, mp_limb_signed_t c)
```

Adds the signed limb **c** to the generalised Fermat number **r** modulo $B^{\text{limbs}} + 1$. The compiler should be able to inline this for the case that there is no overflow from the first limb.

```
void mpn_normmod_2expp1(mp_limb_t *t, mp_size_t limbs)
```

Given **t** a signed integer of **limbs** + 1 limbs in two's complement format, reduce **t** to the corresponding value modulo the generalised Fermat number $B^{\text{limbs}} + 1$, where $B = 2^{\text{FLINT\_BITS}}$.

void **mpn_mul_2expmod_2expp1**(mp_limb_t *t, mp_limb_t *i1, mp_size_t limbs, *flint_bitcnt_t* d)

Given $i1$ a signed integer of $\text{limbs} + 1$ limbs in two's complement format reduced modulo $B^{\text{limbs}} + 1$ up to some overflow, compute $t = i1 \cdot 2^d$ modulo p . The result will not necessarily be fully reduced. The number of bits d must be nonnegative and less than `FLINT_BITS`. Aliasing is permitted.

void **mpn_div_2expmod_2expp1**(mp_limb_t *t, mp_limb_t *i1, mp_size_t limbs, *flint_bitcnt_t* d)

Given $i1$ a signed integer of $\text{limbs} + 1$ limbs in two's complement format reduced modulo $B^{\text{limbs}} + 1$ up to some overflow, compute $t = i1/2^d$ modulo p . The result will not necessarily be fully reduced. The number of bits d must be nonnegative and less than `FLINT_BITS`. Aliasing is permitted.

4.17.4 Generic butterflies

void **fft_adjust**(mp_limb_t *r, mp_limb_t *i1, mp_size_t i, mp_size_t limbs, *flint_bitcnt_t* w)

Set r to $i1$ times z^i modulo $B^{\text{limbs}} + 1$ where z corresponds to multiplication by 2^w . This can be thought of as part of a butterfly operation. We require $0 \leq i < n$ where $nw = \text{limbs} \cdot \text{FLINT\_BITS}$. Aliasing is not supported.

void **fft_adjust_sqrt2**(mp_limb_t *r, mp_limb_t *i1, mp_size_t i, mp_size_t limbs, *flint_bitcnt_t* w, mp_limb_t *temp)

Set r to $i1$ times z^i modulo $B^{\text{limbs}} + 1$ where z corresponds to multiplication by $\sqrt{2}^w$. This can be thought of as part of a butterfly operation. We require $0 \leq i < 2 \cdot n$ and odd where $nw = \text{limbs} \cdot \text{FLINT\_BITS}$.

void **butterfly_lshB**(mp_limb_t *t, mp_limb_t *u, mp_limb_t *i1, mp_limb_t *i2, mp_size_t limbs, mp_size_t x, mp_size_t y)

We are given two integers $i1$ and $i2$ modulo $B^{\text{limbs}} + 1$ which are not necessarily normalised. We compute $t = (i1 + i2) \cdot B^x$ and $u = (i1 - i2) \cdot B^y$ modulo p . Aliasing between inputs and outputs is not permitted. We require x and y to be less than `limbs` and nonnegative.

void **butterfly_rshB**(mp_limb_t *t, mp_limb_t *u, mp_limb_t *i1, mp_limb_t *i2, mp_size_t limbs, mp_size_t x, mp_size_t y)

We are given two integers $i1$ and $i2$ modulo $B^{\text{limbs}} + 1$ which are not necessarily normalised. We compute $t = (i1 + i2)/B^x$ and $u = (i1 - i2)/B^y$ modulo p . Aliasing between inputs and outputs is not permitted. We require x and y to be less than `limbs` and nonnegative.

4.17.5 Radix 2 transforms

void **fft_butterfly**(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, *flint_bitcnt_t* w)

Set $s = i1 + i2$, $t = z1^i \cdot (i1 - i2)$ modulo $B^{\text{limbs}} + 1$ where $z1 = \exp(\text{Pi} \cdot \text{I}/n)$ corresponds to multiplication by 2^w . Requires $0 \leq i < n$ where $nw = \text{limbs} \cdot \text{FLINT\_BITS}$.

void **ifft_butterfly**(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, *flint_bitcnt_t* w)

Set $s = i1 + z1^i \cdot i2$, $t = i1 - z1^i \cdot i2$ modulo $B^{\text{limbs}} + 1$ where $z1 = \exp(-\text{Pi} \cdot \text{I}/n)$ corresponds to division by 2^w . Requires $0 \leq i < 2n$ where $nw = \text{limbs} \cdot \text{FLINT\_BITS}$.

void **fft_radix2**(mp_limb_t **ii, mp_size_t n, *flint_bitcnt_t* w, mp_limb_t **t1, mp_limb_t **t2)

The radix 2 DIF FFT works as follows:

Input: $[i_0, i_1, \dots, i_{(m-1)}]$, for $m = 2n$ a power of 2.

Output: $[r_0, r_1, \dots, r_{(m-1)}] = \text{FFT}[i_0, i_1, \dots, i_{(m-1)}]$.

Algorithm:

- Recursively compute $[r_0, r_2, r_4, \dots, r_{(m-2)}]$
 $= \text{FFT}[i_0+i(m/2), i_1+i(m/2+1), \dots, i_{(m/2-1)+i(m-1)}]$
- Let $[t_0, t_1, \dots, t_{(m/2-1)}]$
 $= [i_0-i(m/2), i_1-i(m/2+1), \dots, i_{(m/2-1)-i(m-1)}]$
- Let $[u_0, u_1, \dots, u_{(m/2-1)}]$
 $= [z_1^0 t_0, z_1^1 t_1, \dots, z_1^{(m/2-1)} t_{(m/2-1)}]$
 where $z_1 = \exp(2\pi i I/m)$ corresponds to multiplication by 2^w .
- Recursively compute $[r_1, r_3, \dots, r_{(m-1)}]$
 $= \text{FFT}[u_0, u_1, \dots, u_{(m/2-1)}]$

The parameters are as follows:

- $2*n$ is the length of the input and output arrays
- w is such that 2^w is an $2n$ -th root of unity in the ring $\mathbb{Z}/p\mathbb{Z}$ that we are working in, i.e. $p = 2^{wn} + 1$ (here n is divisible by `GMP_LIMB_BITS`)
- ii is the array of inputs (each input is an array of limbs of length `wn/GMP_LIMB_BITS + 1` (the extra limbs being a “carry limb”). Outputs are written in-place.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void fft_truncate(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t
                 **t2, mp_size_t trunc)
```

As for `fft_radix2` except that only the first `trunc` coefficients of the output are computed and the input is regarded as having (implied) zero coefficients from coefficient `trunc` onwards. The coefficients must exist as the algorithm needs to use this extra space, but their value is irrelevant. The value of `trunc` must be divisible by 2.

```
void fft_truncate1(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t
                  **t2, mp_size_t trunc)
```

As for `fft_radix2` except that only the first `trunc` coefficients of the output are computed. The transform still needs all $2n$ input coefficients to be specified.

```
void ifft_radix2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t
                 **t2)
```

The radix 2 DIF IFFT works as follows:

Input: $[i_0, i_1, \dots, i_{(m-1)}]$, for $m = 2n$ a power of 2.

Output: $[r_0, r_1, \dots, r_{(m-1)}]$
 $= \text{IFFT}[i_0, i_1, \dots, i_{(m-1)}]$.

Algorithm:

- Recursively compute $[s_0, s_1, \dots, s_{(m/2-1)}]$
 $= \text{IFFT}[i_0, i_2, \dots, i_{(m-2)}]$
- Recursively compute $[t_{(m/2)}, t_{(m/2+1)}, \dots, t_{(m-1)}]$
 $= \text{IFFT}[i_1, i_3, \dots, i_{(m-1)}]$
- Let $[r_0, r_1, \dots, r_{(m/2-1)}]$
 $= [s_0+z_1^0 t_0, s_1+z_1^1 t_1, \dots, s_{(m/2-1)}+z_1^{(m/2-1)} t_{(m/2-1)}]$ where $z_1 = \exp(-2\pi i I/m)$ corresponds to division by 2^w .

- Let $[r(m/2), r(m/2+1), \dots, r(m-1)]$
 $= [s_0 - z_1^0 t_0, s_1 - z_1^1 t_1, \dots, s(m/2-1) - z_1^{(m/2-1)} t(m/2-1)]$

The parameters are as follows:

- **2*n** is the length of the input and output arrays
- w is such that 2^w is an $2n$ -th root of unity in the ring $\mathbb{Z}/p\mathbb{Z}$ that we are working in, i.e. $p = 2^{wn} + 1$ (here n is divisible by `GMP_LIMB_BITS`)
- **ii** is the array of inputs (each input is an array of limbs of length `wn/GMP_LIMB_BITS + 1` (the extra limbs being a “carry limb”). Outputs are written in-place.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void ifft_truncate(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t trunc)
```

As for `ifft_radix2` except that the output is assumed to have zeros from coefficient `trunc` onwards and only the first `trunc` coefficients of the input are specified. The remaining coefficients need to exist as the extra space is needed, but their value is irrelevant. The value of `trunc` must be divisible by 2.

Although the implementation does not require it, we assume for simplicity that `trunc` is greater than n . The algorithm begins by computing the inverse transform of the first n coefficients of the input array. The unspecified coefficients of the second half of the array are then written: coefficient `trunc + i` is computed as a twist of coefficient `i` by a root of unity. The values of these coefficients are then equal to what they would have been if the inverse transform of the right hand side of the input array had been computed with full data from the start. The function `ifft_truncate1` is then called on the entire right half of the input array with this auxiliary data filled in. Finally a single layer of the IFFT is completed on all the coefficients up to `trunc` being careful to note that this involves doubling the coefficients from `trunc - n` up to n .

```
void ifft_truncate1(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_size_t trunc)
```

Computes the first `trunc` coefficients of the radix 2 inverse transform assuming the first `trunc` coefficients are given and that the remaining coefficients have been set to the value they would have if an inverse transform had already been applied with full data.

The algorithm is the same as for `ifft_truncate` except that the coefficients from `trunc` onwards after the inverse transform are not inferred to be zero but the supplied values.

```
void fft_butterfly_sqrt2(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w, mp_limb_t *temp)
```

Let $w = 2k + 1$, $i = 2j + 1$. Set $s = i_1 + i_2$, $t = z_1^i(i_1 - i_2)$ modulo $B^{\text{limbs}} + 1$ where $z_1^2 = \exp(\pi i/n)$ corresponds to multiplication by 2^w . Requires $0 \leq i < 2n$ where $nw = \text{limbs} * \text{FLINT\_BITS}$.

Here z_1 corresponds to multiplication by 2^k then multiplication by $(2^{(3nw/4)} - 2^{(nw/4)})$. We see z_1^i corresponds to multiplication by $(2^{(3nw/4)} - 2^{(nw/4)}) * 2^{(j+ik)}$.

We first multiply by $2^{(j + ik + nw/4)}$ then multiply by an additional $2^{(nw/2)}$ and subtract.

```
void ifft_butterfly_sqrt2(mp_limb_t *s, mp_limb_t *t, mp_limb_t *i1, mp_limb_t *i2, mp_size_t i, mp_size_t limbs, flint_bitcnt_t w, mp_limb_t *temp)
```

Let $w = 2k + 1$, $i = 2j + 1$. Set $s = i_1 + z_1^i i_2$, $t = i_1 - z_1^i i_2$ modulo $B^{\text{limbs}} + 1$ where $z_1^2 = \exp(-\pi i/n)$ corresponds to division by 2^w . Requires $0 \leq i < 2n$ where $nw = \text{limbs} * \text{FLINT\_BITS}$.

Here z_1 corresponds to division by 2^k then division by $(2^{(3nw/4)} - 2^{(nw/4)})$. We see z_1^i corresponds to division by $(2^{(3nw/4)} - 2^{(nw/4)}) * 2^{(j+ik)}$ which is the same as division by $2^{(j+ik + 1)}$ then multiplication by $(2^{(3nw/4)} - 2^{(nw/4)})$.

Of course, division by $2^{(j+ik + 1)}$ is the same as multiplication by $2^{(2*wn - j - ik - 1)}$. The exponent is positive as $i \leq 2 \cdot n$, $j < n$, $k < w/2$.

We first multiply by $2^{(2*wn - j - ik - 1 + wn/4)}$ then multiply by an additional $2^{(nw/2)}$ and subtract.

```
void fft_truncate_sqrt2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1,
    mp_limb_t **t2, mp_limb_t **temp, mp_size_t trunc)
```

As per `fft_truncate` except that the transform is twice the usual length, i.e. length $4n$ rather than $2n$. This is achieved by making use of twiddles by powers of a square root of 2, not powers of 2 in the first layer of the transform.

We require nw to be at least 64 and the three temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void ifft_truncate_sqrt2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1,
    mp_limb_t **t2, mp_limb_t **temp, mp_size_t trunc)
```

As per `ifft_truncate` except that the transform is twice the usual length, i.e. length $4n$ instead of $2n$. This is achieved by making use of twiddles by powers of a square root of 2, not powers of 2 in the final layer of the transform.

We require nw to be at least 64 and the three temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

4.17.6 Matrix Fourier Transforms

```
void fft_butterfly_twiddle(mp_limb_t *u, mp_limb_t *v, mp_limb_t *s, mp_limb_t *t,
    mp_size_t limbs, flint_bitcnt_t b1, flint_bitcnt_t b2)
```

Set $u = 2^{b1}*(s + t)$, $v = 2^{b2}*(s - t)$ modulo $B^{limbs} + 1$. This is used to compute $u = 2^{(ws*tw1)}*(s + t)$, $v = 2^{(w+ws*tw2)}*(s - t)$ in the matrix Fourier algorithm, i.e. effectively computing an ordinary butterfly with additional twiddles by $z^{1^{rc}}$ for row r and column c of the matrix of coefficients. Aliasing is not allowed.

```
void ifft_butterfly_twiddle(mp_limb_t *u, mp_limb_t *v, mp_limb_t *s, mp_limb_t *t,
    mp_size_t limbs, flint_bitcnt_t b1, flint_bitcnt_t b2)
```

Set $u = s/2^{b1} + t/2^{b1}$, $v = s/2^{b1} - t/2^{b1}$ modulo $B^{limbs} + 1$. This is used to compute $u = 2^{(-ws*tw1)}*s + 2^{(-ws*tw2)}*t$, $v = 2^{(-ws*tw1)}*s + 2^{(-ws*tw2)}*t$ in the matrix Fourier algorithm, i.e. effectively computing an ordinary butterfly with additional twiddles by $z^{1^{(-rc)}}$ for row r and column c of the matrix of coefficients. Aliasing is not allowed.

```
void fft_radix2_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w,
    mp_limb_t **t1, mp_limb_t **t2, mp_size_t ws, mp_size_t r,
    mp_size_t c, mp_size_t rs)
```

As for `fft_radix2` except that the coefficients are spaced by is in the array ii and an additional twist by z^{c*i} is applied to each coefficient where i starts at r and increases by rs as one moves from one coefficient to the next. Here z corresponds to multiplication by 2^{ws} .

```
void ifft_radix2_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w,
    mp_limb_t **t1, mp_limb_t **t2, mp_size_t ws, mp_size_t r,
    mp_size_t c, mp_size_t rs)
```

As for `ifft_radix2` except that the coefficients are spaced by is in the array ii and an additional twist by $z^{(-c*i)}$ is applied to each coefficient where i starts at r and increases by rs as one moves from one coefficient to the next. Here z corresponds to multiplication by 2^{ws} .

```
void fft_truncate1_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w,
    mp_limb_t **t1, mp_limb_t **t2, mp_size_t ws, mp_size_t r,
    mp_size_t c, mp_size_t rs, mp_size_t trunc)
```

As per `fft_radix2_twiddle` except that the transform is truncated as per `fft_truncate1`.

```
void ifft_truncate1_twiddle(mp_limb_t **ii, mp_size_t is, mp_size_t n, flint_bitcnt_t w,
                           mp_limb_t **t1, mp_limb_t **t2, mp_size_t ws, mp_size_t r,
                           mp_size_t c, mp_size_t rs, mp_size_t trunc)
```

As per `ifft_radix2_twiddle` except that the transform is truncated as per `ifft_truncate1`.

```
void fft_mfa_truncate_sqrt2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1,
                           mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t
                           trunc)
```

This is as per the `fft_truncate_sqrt2` function except that the matrix Fourier algorithm is used for the left and right FFTs. The total transform length is $4n$ where $n = 2^{\text{depth}}$ so that the left and right transforms are both length $2n$. We require `trunc > 2*n` and that `trunc` is divisible by $2*n1$ (explained below). The coefficients are produced in an order different from `fft_truncate_sqrt2`.

The matrix Fourier algorithm, which is applied to each transform of length $2n$, works as follows. We set `n1` to a power of 2 about the square root of n . The data is then thought of as a set of `n2` rows each with `n1` columns (so that $n1*n2 = 2n$).

The length $2n$ transform is then computed using a whole pile of short transforms. These comprise `n1` column transforms of length `n2` followed by some twiddles by roots of unity (namely z^{rc} where r is the row and c the column within the data) followed by `n2` row transforms of length `n1`. Along the way the data needs to be rearranged due to the fact that the short transforms output the data in binary reversed order compared with what is needed.

The matrix Fourier algorithm provides better cache locality by decomposing the long length $2n$ transforms into many transforms of about the square root of the original length.

For better cache locality the `sqrt2` layer of the full length $4n$ transform is folded in with the column FFTs performed as part of the first matrix Fourier algorithm on the left half of the data.

The second half of the data requires a truncated version of the matrix Fourier algorithm. This is achieved by truncating to an exact multiple of the row length so that the row transforms are full length. Moreover, the column transforms will then be truncated transforms and their truncated length needs to be a multiple of 2. This explains the condition on `trunc` given above.

To improve performance, the extra twiddles by roots of unity are combined with the butterflies performed at the last layer of the column transforms.

We require `nw` to be at least 64 and the three temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void ifft_mfa_truncate_sqrt2(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1,
                           mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t
                           trunc)
```

This is as per the `ifft_truncate_sqrt2` function except that the matrix Fourier algorithm is used for the left and right IFFTs. The total transform length is $4n$ where $n = 2^{\text{depth}}$ so that the left and right transforms are both length $2n$. We require `trunc > 2*n` and that `trunc` is divisible by $2*n1$.

We set `n1` to a power of 2 about the square root of n .

As per the matrix fourier FFT the `sqrt2` layer is folded into the final column IFFTs for better cache locality and the extra twiddles that occur in the matrix Fourier algorithm are combined with the butterflies performed at the first layer of the final column transforms.

We require `nw` to be at least 64 and the three temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void fft_mfa_truncate_sqrt2_outer(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t
                                **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1,
                                mp_size_t trunc)
```

Just the outer layers of `fft_mfa_truncate_sqrt2`.

```
void fft_mfa_truncate_sqrt2_inner(mp_limb_t **ii, mp_limb_t **jj, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc, mp_limb_t **tt)
```

The inner layers of `fft_mfa_truncate_sqrt2` and `ifft_mfa_truncate_sqrt2` combined with pointwise mults.

```
void ifft_mfa_truncate_sqrt2_outer(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp, mp_size_t n1, mp_size_t trunc)
```

The outer layers of `ifft_mfa_truncate_sqrt2` combined with normalisation.

4.17.7 Negacyclic multiplication

```
void fft_negacyclic(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp)
```

As per `fft_radix2` except that it performs a sqrt2 negacyclic transform of length $2n$. This is the same as the radix 2 transform except that the i -th coefficient of the input is first multiplied by $\sqrt{2}^{iw}$.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void ifft_negacyclic(mp_limb_t **ii, mp_size_t n, flint_bitcnt_t w, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **temp)
```

As per `ifft_radix2` except that it performs a sqrt2 negacyclic inverse transform of length $2n$. This is the same as the radix 2 inverse transform except that the i -th coefficient of the output is finally divided by $\sqrt{2}^{iw}$.

We require nw to be at least 64 and the two temporary space pointers to point to blocks of size $n*w + \text{FLINT\_BITS}$ bits.

```
void fft_naive_convolution_1(mp_limb_t *r, mp_limb_t *ii, mp_limb_t *jj, mp_size_t m)
```

Performs a naive negacyclic convolution of `ii` with `jj`, both of length m , and sets `r` to the result. This is essentially multiplication of polynomials modulo $x^m + 1$.

```
void _fft_mulmod_2expp1(mp_limb_t *r1, mp_limb_t *i1, mp_limb_t *i2, mp_size_t r_limbs, flint_bitcnt_t depth, flint_bitcnt_t w)
```

Multiply `i1` by `i2` modulo $B^{r\_limbs} + 1$ where $r\_limbs = nw/\text{FLINT\_BITS}$ with $n = 2^{\text{depth}}$. Uses the negacyclic FFT convolution CRT'd with a 1 limb naive convolution. We require that `depth` and `w` have been selected as per the wrapper `fft_mulmod_2expp1` below.

slong **fft_adjust_limbs**(mp_size_t limbs)

Given a number of limbs, returns a new number of limbs (no more than the next power of 2) which will work with the Nussbaumer code. It is only necessary to make this adjustment if `limbs > FFT_MULMOD_2EXPP1_CUTOFF`.

```
void fft_mulmod_2expp1(mp_limb_t *r, mp_limb_t *i1, mp_limb_t *i2, mp_size_t n, mp_size_t w, mp_limb_t *tt)
```

As per `_fft_mulmod_2expp1` but with a tuned cutoff below which more classical methods are used for the convolution. The temporary space is required to fit $n*w + \text{FLINT\_BITS}$ bits. There are no restrictions on n , but if `limbs = n*w/FLINT\_BITS` then if `limbs` exceeds `FFT_MULMOD_2EXPP1_CUTOFF` the function `fft_adjust_limbs` must be called to increase the number of limbs to an appropriate value.

4.17.8 Integer multiplication

```
void mul_truncate_sqrt2(mp_ptr r1, mp_srcptr i1, mp_size_t n1, mp_srcptr i2, mp_size_t n2,
                        flint_bitcnt_t depth, flint_bitcnt_t w)
```

Integer multiplication using the radix 2 truncated sqrt2 transforms.

Set $(r1, n1 + n2)$ to the product of $(i1, n1)$ by $(i2, n2)$. This is achieved through an FFT convolution of length at most $2^{(\text{depth} + 2)}$ with coefficients of size nw bits where $n = 2^{\text{depth}}$. We require $\text{depth} \geq 6$. The input data is broken into chunks of data not exceeding $(nw - (\text{depth} + 1))/2$ bits. If breaking the first integer into chunks of this size results in $j1$ coefficients and breaking the second integer results in $j2$ chunks then $j1 + j2 - 1 \leq 2^{(\text{depth} + 2)}$.

If $n = 2^{\text{depth}}$ then we require nw to be at least 64.

```
void mul_mfa_truncate_sqrt2(mp_ptr r1, mp_srcptr i1, mp_size_t n1, mp_srcptr i2, mp_size_t
                           n2, flint_bitcnt_t depth, flint_bitcnt_t w)
```

As for `mul_truncate_sqrt2` except that the cache friendly matrix Fourier algorithm is used.

If $n = 2^{\text{depth}}$ then we require nw to be at least 64. Here we also require w to be 2^i for some $i \geq 0$.

```
void flint_mpn_mul_fft_main(mp_ptr r1, mp_srcptr i1, mp_size_t n1, mp_srcptr i2, mp_size_t
                           n2)
```

The main integer multiplication routine. Sets $(r1, n1 + n2)$ to $(i1, n1)$ times $(i2, n2)$. We require $n1 \geq n2 > 0$.

4.17.9 Convolution

```
void fft_convolution(mp_limb_t **ii, mp_limb_t **jj, slong depth, slong limbs, slong trunc,
                    mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **s1, mp_limb_t **tt)
```

Perform an FFT convolution of ii with jj , both of length $4*n$ where $n = 2^{\text{depth}}$. Assume that all but the first `trunc` coefficients of the output (placed in ii) are zero. Each coefficient is taken modulo $B^{\text{limbs} + 1}$. The temporary spaces $t1$, $t2$ and $s1$ must have $\text{limbs} + 1$ limbs of space and tt must have $2*(\text{limbs} + 1)$ of free space.

4.17.10 FFT Precaching

```
void fft_precache(mp_limb_t **jj, slong depth, slong limbs, slong trunc, mp_limb_t **t1,
                  mp_limb_t **t2, mp_limb_t **s1)
```

Precompute the FFT of jj for use with precache functions. The parameters are as for `fft_convolution`.

```
void fft_convolution_precache(mp_limb_t **ii, mp_limb_t **jj, slong depth, slong limbs, slong
                              trunc, mp_limb_t **t1, mp_limb_t **t2, mp_limb_t **s1,
                              mp_limb_t **tt)
```

As per `fft_convolution` except that it is assumed `fft_precache` has been called on jj with the same parameters. This will then run faster than if `fft_convolution` had been run with the original jj .

4.18 fft_small.h – FFT modulo word-size primes

This module currently requires building FLINT with support for AVX2 or NEON instructions.

4.18.1 Integer multiplication

type `mpn_ctx_struct`

type `mpn_ctx_t`

Context object for multiplications allowing non-FFT moduli. The structure contains FFT context objects for multiple FFT primes (currently 8) together with tables for Chinese remaindering.

void `mpn_ctx_init`(*mpn_ctx_t* R, *ulong* p)

Initialize multiplication context object with initial prime p.

void `mpn_ctx_clear`(*mpn_ctx_t* R)

Free memory allocated by the context object.

mpn_ctx_struct *`get_default_mpn_ctx`(void)

Return a pointer to a cached thread-local context object used by default for multiplications. Calling `flint_cleanup()` or `flint_cleanup_master()` frees the cache.

void `mpn_ctx_mpn_mul`(*mpn_ctx_t* R, *ulong* *r1, const *ulong* *i1, *ulong* n1, const *ulong* *i2, *ulong* n2)

void `mpn_mul_default_mpn_ctx`(*nn_ptr* r1, *nn_srcptr* i1, *slong* n1, *nn_srcptr* i2, *slong* n2)

Writes to r1 the product of the integers (i1, n1) and (i2, n2). Assumes that $n_1 \geq n_2 \geq 1$, respectively using a given context object R or the default thread-local object.

4.18.2 Polynomial arithmetic

void `_nmod_poly_mul_mid_mpn_ctx`(*ulong* *z, *ulong* zl, *ulong* zh, const *ulong* *a, *ulong* an, const *ulong* *b, *ulong* bn, *nmod_t* mod, *mpn_ctx_t* R)

void `_nmod_poly_mul_mid_default_mpn_ctx`(*nn_ptr* res, *slong* zl, *slong* zh, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, *nmod_t* mod)

Writes to z the middle product containing coefficients in the range [zl, zh) of the product of the polynomials (a, an) and (b, bn), respectively using a given context object R or the default thread-local object. Assumes that $an \geq bn \geq 1$.

int `_fmpz_poly_mul_mid_mpn_ctx`(*fmpz* *z, *ulong* zl, *ulong* zh, const *fmpz* *a, *ulong* an, const *fmpz* *b, *ulong* bn, *mpn_ctx_t* R)

int `_fmpz_poly_mul_mid_default_mpn_ctx`(*fmpz* *z, *ulong* zl, *ulong* zh, const *fmpz* *a, *ulong* an, const *fmpz* *b, *ulong* bn)

Like the `nmod` functions. Performs the multiplication and returns 1 if there are sufficiently many primes R to compute the result; otherwise returns 0 without touching the output.

void `_nmod_poly_divrem_mpn_ctx`(*ulong* *q, *ulong* *r, const *ulong* *a, *ulong* an, const *ulong* *b, *ulong* bn, *nmod_t* mod, *mpn_ctx_t* R)

Polynomial division with remainder.

4.18.3 Preconditioned polynomial arithmetic

type `mul_precomp_struct`

void `_mul_precomp_init`(*mul_precomp_struct* *M, const *ulong* *b, *ulong* bn, *ulong* btrunc, *ulong* depth, *nmod_t* mod, *mpn_ctx_t* R)

void `_mul_precomp_clear`(*mul_precomp_struct* *M)

Represents (b, bn) in transformed form for preconditioned multiplication.

int `_nmod_poly_mul_mid_precomp`(*ulong* *z, *ulong* zl, *ulong* zh, const *ulong* *a, *ulong* an, *mul_precomp_struct* *M, *nmod_t* mod, *mpn_ctx_t* R)

Polynomial multiplication given a precomputed transform M. Returns 1 if successful, 0 if the pre-computed transform is too short.

type `nmod_poly_divrem_precomp_struct`

void `_nmod_poly_divrem_precomp_init`(*nmod_poly_divrem_precomp_struct* *M, const *ulong* *b, *ulong* bn, *ulong* Bn, *nmod_t* mod, *mpn_ctx_t* R)

void `_nmod_poly_divrem_precomp_clear`(*nmod_poly_divrem_precomp_struct* *M)

Represents (b, bn) and its inverse in transformed form for preconditioned multiplication.

int `_nmod_poly_divrem_precomp`(*ulong* *q, *ulong* *r, const *ulong* *a, *ulong* an, *nmod_poly_divrem_precomp_struct* *M, *nmod_t* mod, *mpn_ctx_t* R)

Polynomial multiplication given a precomputed transform M. Returns 1 if successful, 0 if the pre-computed transform is too short.

4.19 qsieve.h – Quadratic sieve

ulong `qsieve_knuth_schroeppel`(*qs_t* qs_inf)

Return the Knuth-Schroeppel multiplier for the n , integer to be factored based upon the Knuth-Schroeppel function.

ulong `qsieve_primes_init`(*qs_t* qs_inf)

Compute the factor base prime along with there inverse for kn , where k is Knuth-Schroeppel multiplier and n is the integer to be factored. It also computes the square root of kn modulo factor base primes.

ulong `qsieve_primes_increment`(*qs_t* qs_inf, *ulong* delta)

It increase the number of factor base primes by amount ‘delta’ and calculate inverse of those primes along with the square root of kn modulo those primes.

void `qsieve_init_A0`(*qs_t* qs_inf)

First it chooses the possible range of factor of A_0 , based on the number of bits in optimal value of A_0 . It tries to select range such that we have plenty of primes to choose from as well as number of factor in A_0 are sufficient. For input of size less than 130 bit, this selection method doesn’t work therefore we randomly generate 2 or 3-subset of all the factor base prime as the factor of A_0 . Otherwise, if we have to select s factor for A_0 , we generate $s - 1$ - subset from odd indices of the possible range of factor and then search last factor using binary search from the even indices of possible range of factor such that value of A_0 is close to it’s optimal value.

void `qsieve_next_A0`(*qs_t* qs_inf)

Find next candidate for A_0 as follows: generate next lexicographic $s - 1$ -subset from the odd indices of possible range of factor base and choose the last factor from even indices using binary search so that value A_0 is close to it’s optimal value.

void **qsieve_compute_pre_data**(qs_t qs_inf)

Precompute all the data associated with factor's of A_0 , since A_0 is going to be fixed for several A .

void **qsieve_init_poly_first**(qs_t qs_inf)

Initializes the value of $A = q_0 * A_0$, where q_0 is non-factor base prime. precompute the data necessary for generating different B value using grey code formula. Combine the data calculated for the factor of A_0 along with the parameter q_0 to obtain data as for factor of A . It also calculates the sieve offset for all the factor base prime, for first polynomial.

void **qsieve_init_poly_next**(qs_t qs_inf, *slong* i)

Generate next polynomial or next B value for particular A and also updates the sieve offsets for all the factor base prime, for this B value.

void **qsieve_compute_C**(*fmpz_t* C, qs_t qs_inf, qs_poly_t poly)

Given A and B , calculate $C = (B^2 - A)/N$.

void **qsieve_do_sieving**(qs_t qs_inf, unsigned char *sieve, qs_poly_t poly)

First initialize the sieve array to zero, then for each $p \in \text{factor base}$, add $\log_2(p)$ to the locations $\text{soln1}_p + i * p$ and $\text{soln2}_p + i * p$ for $i = 0, 1, 2, \dots$, where soln1_p and soln2_p are the sieve offsets calculated for p .

void **qsieve_do_sieving2**(qs_t qs_inf, unsigned char *sieve, qs_poly_t poly)

Perform the same task as above but instead of sieving over whole array at once divide the array in blocks and then sieve over each block for all the primes in factor base.

slong **qsieve_evaluate_candidate**(qs_t qs_inf, *ulong* i, unsigned char *sieve, qs_poly_t poly)

For location i in sieve array value at which, is greater than sieve threshold, check the value of $Q(x)$ at position i for smoothness. If value is found to be smooth then store it for later processing, else check the residue for the partial if it is found to be partial then store it for late processing.

slong **qsieve_evaluate_sieve**(qs_t qs_inf, unsigned char *sieve, qs_poly_t poly)

Scan the sieve array for location at, which accumulated value is greater than sieve threshold.

slong **qsieve_collect_relations**(qs_t qs_inf, unsigned char *sieve)

Call for initialization of polynomial, sieving, and scanning of sieve for all the possible polynomials for particular hypercube i.e. A .

void **qsieve_write_to_file**(qs_t qs_inf, *ulong* prime, const *fmpz_t* Y, const qs_poly_t poly)

Write a relation to the file in a binary format as follows. First, write large prime of size `sizeof(ulong)`, in case of full relation it is 1. After this, write the number of small primes with size `sizeof(slong)`. Then, write the small primes, with a total size of `number_of_small_primes * sizeof(slong)`. Then, write the number of factors with a size of `sizeof(slong)`. After that, write the factors and their exponents in the format `factor_1, exponent_1, factor_2, ...`, all with a total size of `2 * number_of_factors * sizeof(slong)`. Then write Y with the size of Y first (size `sizeof(slong)`, that may be negative), and then its limbs (size `Y_size * sizeof(ulong)`).

hash_t ***qsieve_get_table_entry**(qs_t qs_inf, *ulong* prime)

Return the pointer to the location of 'prime' is hash table if it exist, else create an entry for it in hash table and return pointer to that.

void **qsieve_add_to_hashtable**(qs_t qs_inf, *ulong* prime)

Add 'prime' to the hash table.

relation_t **qsieve_parse_relation**(qs_t qs_inf)

Read a relation from the file associated with *qs_inf* and parse it to obtain all the parameters of the relation.

relation_t **qsieve_merge_relation**(qs_t qs_inf, relation_t a, relation_t b)

Given two partial relation having same large prime, merge them to obtain a full relation.

```
int qsieve_compare_relation(const void *a, const void *b)
```

Compare two relation based on, first large prime, then number of factor and then offsets of factor in factor base.

```
int qsieve_remove_duplicates(relation_t *rel_list, slong num_relations)
```

Remove duplicate from given list of relations by sorting relations in the list.

```
void qsieve_insert_relation2(qs_t qs_inf, relation_t *rel_list, slong num_relations)
```

Given a list of relations, insert each relation from the list into the matrix for further processing.

```
int qsieve_process_relation(qs_t qs_inf)
```

After we have accumulated required number of relations, first process the file by reading all the relations, removes singleton. Then merge all the possible partial to obtain full relations.

```
void qsieve_factor(fmpz_factor_t factors, const fmpz_t n)
```

Factor n using the quadratic sieve method. It is required that n is not a prime and not a perfect power. There is no guarantee that the factors found will be prime, or distinct.

4.20 radix.h – multiprecision arithmetic in general radix

This module implements multiprecision arithmetic in a general radix $2 \leq b < \beta$ where $\beta = 2^{\text{FLINT\_BITS}}$ is the machine word radix.

The user selects both a *digit radix* b and a *limb radix* $B = b^e$ where $B < \beta$, allowing several digits to be packed into each machine word. A *limb* is a chunk of e consecutive digits stored in one word. Efficiency is typically maximized by choosing the largest limb radix that fits in a machine word. For example, for decimal arithmetic one should typically select $B = 10^{19}$ on a 64-bit machine and $B = 10^9$ on a 32-bit machine. A smaller limb radix may however be more efficient in some circumstances.

Arithmetic in radix B is generally slower than arithmetic in the machine word radix β , but has the advantage that base-related operations such as reduction modulo b^n and string input/output in base b can be done in time $O(n)$ rather than $O(M(n))$ or $O(M(n) \log n)$. The two principal applications are decimal arithmetic (with $b = 10$) and p -adic arithmetic (with $b = p$ a small odd prime number). This module can also be useful for experimenting with multi-limb algorithms where debugging and discovering corner cases is easier in a small base like 10 than with full-word limbs.

The functions in this module are not optimized for radix 2^e , where the machine word radix is applicable (see GMP's `mpn` and FLINT's `flint_mpn` routines).

4.20.1 Radix objects

```
type radix_struct
```

```
type radix_t
```

A context object defines the radix $B = b^e$ and stores additional data such as various precomputed powers b^k and their inverses used for fast division and modular reduction.

```
void radix_init(radix_t radix, ulong b, unsigned int e)
```

Initializes context object representing radix $B = b^e$ given $b \geq 2$ and an exponent $e \geq 1$ such that $B < \beta$. Optionally, the user can input $e = 0$, in which case e will automatically be set to the largest exponent such that $b^e < \beta$.

```
void radix_clear(radix_t radix)
```

Frees any memory allocated by the context object.

```
void radix_init_randtest(radix_t radix, flint_rand_t state)
```

Initializes to a randomly chosen radix.

```
ulong radix_digit_radix(const radix_t radix)
```

ulong **radix_limb_radix**(const *radix_t* radix)

ulong **radix_limb_exponent**(const *radix_t* radix)

Respectively returns b , B and e .

4.20.2 Low-level natural number arithmetic

A limb has type *ulong*. The type *nn_ptr* denotes a pointer to an array of writable limbs and *nn_srcptr* denotes a pointer to an array of readonly limbs. We use the notation (x, n) for the natural number in the range $[0, B^n - 1]$ represented by an array of n limbs starting at address x . Except where otherwise noted, the following rules apply:

- Input limbs must be normalised to the range $[0, B - 1]$, and output limbs will also be normalised to this range.
- Aliasing is permitted as long as aliased operands start at the same address.

ulong **_radix_size_digits_1**(*ulong* c, const *radix_t* radix)

ulong **radix_size_digits_1**(*ulong* c, const *radix_t* radix)

Return the size of the limb c in digits. The underscore method requires $c > 0$.

ulong **_radix_size_digits**(*nn_srcptr* x, *slong* n, const *radix_t* radix)

ulong **radix_size_digits**(*nn_srcptr* x, *slong* n, const *radix_t* radix)

Return the size of (x, n) in digits. The underscore requires $n > 0$ and $x_{n-1} \neq 0$. The version without an underscore allows $n = 0$ and $x_{n-1} = 0$.

ulong **_radix_valuation_digits_1**(*ulong* c, const *radix_t* radix)

Return the digit valuation of the limb c , i.e. the number of trailing zeros in base b , or the largest exponent k such that b^k divides c . Requires $c > 0$.

void **radix_rand_limbs**(*nn_ptr* res, *flint_rand_t* state, *slong* n, const *radix_t* radix)

Sets (res, n) to a uniformly random integer in $[0, B^n - 1]$.

void **radix_rand_digits**(*nn_ptr* res, *flint_rand_t* state, *slong* n, const *radix_t* radix)

Sets (res, rn) to a uniformly random integer in $[0, b^n - 1]$ where $rn = \lceil n/e \rceil$.

void **radix_randtest_limbs**(*nn_ptr* res, *flint_rand_t* state, *slong* n, const *radix_t* radix)

Sets (res, n) to a non-uniformly random integer in $[0, B^n - 1]$. This will produce long strings of limbs with the value 0 or $B - 1$ with high probability.

void **radix_randtest_digits**(*nn_ptr* res, *flint_rand_t* state, *slong* n, const *radix_t* radix)

Sets (res, rn) to a non-uniformly random integer in $[0, b^n - 1]$ where $rn = \lceil n/e \rceil$.

ulong **radix_neg**(*nn_ptr* res, *nn_srcptr* x, *slong* n, const *radix_t* radix)

Sets (res, n) to the B :s complement negation of (x, n) , i.e., $(-x) \bmod B^n$, returning borrow.

ulong **radix_add**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, const *radix_t* radix)

ulong **radix_sub**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, const *radix_t* radix)

Sets (res, xn) to the sum or difference of (x, xn) and (y, yn) , returning carry or borrow. Requires $xn \geq yn \geq 1$.

ulong **radix_lshift_digits**(*nn_ptr* res, *nn_srcptr* a, *slong* n, unsigned int k, const *radix_t* radix)

Sets (res, n) to (a, n) shifted left by k digits, i.e. multiplied by b^k , with the carry out returned in the k least significant digits of the return limb. This is equivalent to **radix_mul_1()** with scalar b^k , but more efficient. Requires $n > 1$ and $1 \leq k < e$. Note that this operation only makes sense in a limb radix with $e \geq 2$ digits per limb.

ulong **radix_rshift_digits**(*nn_ptr* res, *nn_srcptr* a, *slong* n, unsigned int k, const *radix_t* radix)

Sets (res, n) to (a, n) shifted right by k digits, i.e. divided by b^k , with the fractional part returned in the k most significant digits of the return limb. This is equivalent to **radix_divrem_1()** with scalar b^k followed by multiplying the remainder by b^{e-k} , but more efficient. Requires $n > 1$ and $1 \leq k < e$. Note that this operation only makes sense in a limb radix with $e \geq 2$ digits per limb.

ulong **radix_mul_1**(*nn_ptr* res, *nn_srcptr* a, *slong* n, *ulong* c, const *radix_t* radix)

Sets (res, xn) to (x, xn) multiplied by c , returning carry out. Requires $1 \leq d \leq c$.

ulong **radix_mul_two**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

Sets (res, xn) to (x, xn) multiplied by 2, returning carry out.

void **radix_mul**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, const *radix_t* radix)

Sets $(res, xn + yn)$ to the product of (x, xn) and (y, yn) . Requires $xn, yn \geq 1$. Does not allow res to be aliased with either x or y .

This function is currently a wrapper of *radix_mulmid()*.

void **radix_sqr**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, const *radix_t* radix)

Sets $(res, 2\,xn)$ to the square of (x, xn) . Requires $xn \geq 1$. Does not allow res to be aliased with x .

void **radix_mulmid**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, *slong* lo, *slong* hi, const *radix_t* radix)

void **radix_mulmid_classical**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, *slong* lo, *slong* hi, const *radix_t* radix)

void **radix_mulmid_fft_small**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, *slong* lo, *slong* hi, const *radix_t* radix)

void **radix_mulmid_KS**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, *slong* lo, *slong* hi, const *radix_t* radix)

void **radix_mulmid_naive**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *nn_srcptr* y, *slong* yn, *slong* lo, *slong* hi, const *radix_t* radix)

Short, truncated or middle product. Requires $xn, yn \geq 1$ and $0 \leq lo < hi \leq xn + yn$. Does not allow res to be aliased with either x or y .

Viewing x and y as polynomials $X, Y \in \mathbb{Z}[T]$ evaluated at $T = B$, we implicitly compute the polynomial product $Z = XY$ and extract the slice $Z_{lo:hi} = \sum_{k=lo}^{hi-1} ([T^k]Z)T^{k-lo}$, and write $Z_{lo:hi}(B) \bmod B^{hi-lo}$ to $(res, hi - lo)$. We omit carries from lower order terms and we discard carries that would go into higher order terms.

In the special case $lo = 0$, this sets (res, hi) to the low product $xy \bmod B^{hi}$.

In the special case $hi = xn + yn$, this sets $(res, hi - lo)$ to an approximate high product r satisfying $xy \geq rB^{lo} \geq xy - \min(xn, yn, lo)B^{lo+1}$.

In the special case $lo = 0, hi = xn + yn$, this computes the full product.

Various multiplication algorithms are implemented:

- *classical* is an efficient implementation of the schoolbook algorithm, good when either yn or $hi - lo$ is small.
- *fft_small* multiplies using FFT. This is only available if FLINT is built with support for the *fft_small* module.
- *KS* denotes Kronecker substitution, converting to a larger integer multiplication in the machine word radix.
- *naive* is an unoptimized reference implementation which does a polynomial multiplication using *fmpz_poly*.

ulong **radix_divrem_1**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *ulong* d, const *radix_t* radix)

Sets (res, xn) to the quotient of (x, xn) divided by d , returning the remainder. Requires $1 \leq d \leq B - 1$.

ulong **radix_divrem_two**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

Sets (res, xn) to the quotient of (x, xn) divided by 2, returning the remainder.

void **radix_divexact_1**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *ulong* d, const *radix_t* radix)

Sets (res, xn) to the quotient of (x, xn) divided by d assuming that (x, xn) is an exact multiple of d , with undefined behavior otherwise. Requires $1 \leq d \leq B - 1$.

void **radix_inv_approx_basecase**(*nn_ptr* q, *nn_srcptr* a, *slong* an, *slong* n, const *radix_t* radix)

void **radix_inv_approx**(*nn_ptr* q, *nn_srcptr* a, *slong* an, *slong* n, const *radix_t* radix)

Given (a, an) with $a_{an-1} \neq 0$ representing a fixed-point number $a \in [1/B, 1)$ with an fraction limbs, sets $(q, n + 2)$ to an approximation of $1/a \in (1, B]$ with n fraction limbs and two integral limbs (the highest limb may be zero). The relative error is bounded by $4B^{-n}$, i.e. the absolute error is bounded by $4B^{-n}/a$.

void **radix_div_approx_invmul**(*nn_ptr* q, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, *slong* n, const *radix_t* radix)

void **radix_div_approx**(*nn_ptr* q, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, *slong* n, const *radix_t* radix)

Given (b, bn) with $b_{bn-1} \neq 0$ representing a fixed-point number $b \in [1/B, 1)$ with $bn \geq 1$ fraction limbs, and (a, an) with $an \geq 1$ fraction limbs representing a fixed-point number $a \in [0, 1)$, sets $(q, n + 2)$ to an approximation of a/b with n fraction limbs and two integral limbs (the highest limbs may be zero). The absolute error is bounded by $4B^{-n}/b$.

void **radix_divrem_via_mpn**(*nn_ptr* q, *nn_ptr* r, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, const *radix_t* radix)

void **radix_divrem_newton**(*nn_ptr* q, *nn_ptr* r, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, const *radix_t* radix)

void **radix_divrem_newton_karp_markstein**(*nn_ptr* Q, *nn_ptr* R, *nn_srcptr* A, *slong* An, *nn_srcptr* B, *slong* Bn, const *radix_t* radix)

void **radix_divrem**(*nn_ptr* q, *nn_ptr* r, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, const *radix_t* radix)

Sets $(q, an - bn + 1)$ to the quotient and (r, bn) to the remainder of (a, an) divided by (b, bn) . Requires $an \geq bn \geq 1$ and $b_{bn-1} \neq 0$. The user can pass NULL for r to compute only the quotient.

void **radix_divrem_preinv**(*nn_ptr* q, *nn_ptr* r, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, *nn_srcptr* binv, *slong* binvn, const *radix_t* radix)

Similar to **radix_divrem()**, but accepts a precomputed inverse of b given as $(b, binvn + 2)$ with $binvn$ fraction limbs and two integral limbs, as computed by **radix_inv_approx()**. Currently requires that $binvn \geq an - bn + 1$. Passing an inverse with several extra limbs can improve performance. The user can pass NULL for r to compute only the quotient.

int **radix_div**(*nn_ptr* q, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, const *radix_t* radix)

Sets $(q, an - bn + 1)$ to the quotient of a divided by b and returns 1 if the division is exact. If the division is not exact, returns 0 leaving the data in q undefined.

int **radix_divexact**(*nn_ptr* q, *nn_srcptr* a, *slong* an, *nn_srcptr* b, *slong* bn, const *radix_t* radix)

Sets $(q, an - bn + 1)$ to the quotient of a divided by b assuming that the division is exact. The result is undefined if the division is inexact.

int **radix_invmod_bn**(*nn_ptr* res, *nn_srcptr* x, *slong* xn, *slong* n, const *radix_t* radix)

If (x, xn) is invertible modulo B^n , sets (res, n) to the inverse and returns 1. If (x, xn) is not invertible, returns 0.

int **radix_cmp_bn_half**(*nn_srcptr* x, *slong* n, const *radix_t* radix)

Returns -1, 0 or 1 according to whether (x, n) is less, equal or greater than $\lfloor B^n/2 \rfloor$. We require n to be positive.

void **radix_rsqrt_1_approx_basecase**(*nn_ptr* res, *ulong* a, *slong* n, const *radix_t* radix)

void **radix_rsqrt_1_approx**(*nn_ptr* res, *ulong* a, *slong* n, const *radix_t* radix)

Sets (res, n) to the fractional limbs of an approximation of $1/\sqrt{a}$. Assumes that $2 \leq a < B$. The error is bounded by $2B^{-n}$, i.e. by 2 fixed-point ulps.

4.20.3 Radix conversion

slong **radix_get_mpn_basecase**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

slong **radix_get_mpn_divconquer**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

slong **radix_get_mpn**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

Convert (a, an) to the machine word radix, writing the output limbs to *res* and returning the exact size of the result in the machine word radix. Leading zero limbs are omitted from the output size and may or may not be written. Requires that *res* has enough space for the largest representable integer with *an* limbs in the given radix, plus one extra limb.

slong **radix_set_mpn_basecase**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

slong **radix_set_mpn_divconquer**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

slong **radix_set_mpn**(*nn_ptr* res, *nn_srcptr* a, *slong* an, const *radix_t* radix)

Convert (a, an) from the machine word radix, writing the output limbs to *res* and returning the exact size of the result in the target radix. Leading zero limbs are omitted from the output size and may or may not be written. Requires that *res* has space for at least `radix_set_mpn_need_alloc(an, radix)` limbs.

slong **radix_set_mpn_need_alloc**(*slong* n, const *radix_t* radix)

Return a number of output limbs for which `radix_set_mpn()` is safe to call with input of length *n*.

4.20.4 String conversion

char ***radix_get_str_decimal**(char *res, *nn_srcptr* x, *slong* n, int negative, const *radix_t* radix)

Returns a null-terminated string representing the integer (x, n) in base 10 (e.g. -12345). If $n = 0$, the string 0 is returned. Otherwise, the most significant limb is assumed to be zero; a leading minus sign will be inserted if the flag *negative* is 1.

If the input radix is of the form 10^e , this function runs in $O(n)$. Otherwise, a radix conversion to decimal will be performed internally, which runs in $O(M(n) \log n)$.

If the given pointer *res* is *NULL*, a new string will be allocated with `flint_malloc()` and returned. Otherwise, *res* will be used, and is assumed to have sufficient space to store the result including null terminator.

char ***radix_get_str_sum**(char *res, *nn_srcptr* x, *slong* n, int negative, int ascending, const *radix_t* radix)

Returns (x, n) represented in expression form as a sum over its limbs. For example, 12345678 in radix 10^3 will be printed as $12 * 10^6 + 345 * 10^3 + 678$ if *negative* and *ascending* flags are both set to 0, and as $-(678 + 345 * 10^3 + 12 * 10^6)$ if the *negative* and *ascending* flags are both set to 1.

If the given pointer *res* is *NULL*, a new string will be allocated with `flint_malloc()` and returned. Otherwise, *res* will be used, and is assumed to have sufficient space to store the result including null terminator.

4.20.5 Memory-managed integers

type **radix_integer_struct**

type **radix_integer_t**

Represents a signed multi-limb integer (with respect to a separate radix object), in a format similar to `mpz`. There are three slots: `d` (pointer to limbs), `alloc`, and `size` whose absolute value stores the normalized limb count (negated to indicate a negative value).

void **gr_ctx_init_radix_integer**(*gr_ctx_t* ctx, *ulong* b, unsigned int e)

Initialize a generic ring for working with integers in radix b^e (where $e = 0$ can be provided to select the largest admissible limb radix automatically). Elements have type *radix_integer_t*.

void **radix_integer_init**(*radix_integer_t* res, const *radix_t* radix)

void **radix_integer_clear**(*radix_integer_t* res, const *radix_t* radix)

nn_ptr **radix_integer_fit_limbs**(*radix_integer_t* res, *slong* nlimbs, const *radix_t* radix)

void **radix_integer_zero**(*radix_integer_t* res, const *radix_t* radix)

void **radix_integer_rand_limbs**(*radix_integer_t* res, *flint_rand_t* state, *slong* n, const *radix_t* radix)

void **radix_integer_randtest_limbs**(*radix_integer_t* res, *flint_rand_t* state, *slong* max_limbs, const *radix_t* radix)

void **radix_integer_one**(*radix_integer_t* res, const *radix_t* radix)

void **radix_integer_neg_one**(*radix_integer_t* res, const *radix_t* radix)

int **radix_integer_is_zero**(const *radix_integer_t* x, const *radix_t* radix)

int **radix_integer_is_one**(const *radix_integer_t* x, const *radix_t* radix)

int **radix_integer_is_neg_one**(const *radix_integer_t* x, const *radix_t* radix)

int **radix_integer_equal**(const *radix_integer_t* x, const *radix_integer_t* y, const *radix_t* radix)

int **radix_integer_cmp**(const *radix_integer_t* x, const *radix_integer_t* y, const *radix_t* radix)

int **radix_integer_cmpabs**(const *radix_integer_t* x, const *radix_integer_t* y, const *radix_t* radix)

void **radix_integer_set**(*radix_integer_t* res, const *radix_integer_t* x, const *radix_t* radix)

void **radix_integer_set_ui**(*radix_integer_t* res, *ulong* x, const *radix_t* radix)

void **radix_integer_set_si**(*radix_integer_t* res, *slong* x, const *radix_t* radix)

void **radix_integer_set_fmpz**(*radix_integer_t* res, const *fmpz_t* x, const *radix_t* radix)

void **radix_integer_get_fmpz**(*fmpz_t* res, const *radix_integer_t* x, const *radix_t* radix)

void **radix_integer_neg**(*radix_integer_t* res, const *radix_integer_t* x, const *radix_t* radix)

void **radix_integer_abs**(*radix_integer_t* res, const *radix_integer_t* x, const *radix_t* radix)

int **radix_integer_sgn**(const *radix_integer_t* x, const *radix_t* radix)

void **radix_integer_add**(*radix_integer_t* res, const *radix_integer_t* x, const *radix_integer_t* y, const *radix_t* radix)

void **radix_integer_sub**(*radix_integer_t* res, const *radix_integer_t* x, const *radix_integer_t* y, const *radix_t* radix)

void **radix_integer_mul**(*radix_integer_t* res, const *radix_integer_t* x, const *radix_integer_t* y, const *radix_t* radix)

Direct interface for working with *radix_integer_t*. These can be used as an alternative to the generic interface.

int **radix_integer_is_normalised**(const *radix_integer_t* x, const *radix_t* radix)

Returns whether all limbs of x are reduced modulo the limb radix and the most significant limb is nonzero. If the result is 0, operations on x have undefined behavior.

slong **radix_integer_size_limbs**(const *radix_integer_t* x, const *radix_t* radix)

slong **radix_integer_size_digits**(const *radix_integer_t* x, const *radix_t* radix)

Number of limbs (respectively digits) in x , ignoring the sign.

slong **radix_integer_ssize_limbs**(const *radix_integer_t* x, const *radix_t* radix)

slong **radix_integer_ssize_digits**(const *radix_integer_t* x, const *radix_t* radix)

Number of limbs (respectively digits) in x , negated if x is negative.

ulong **radix_integer_get_limb**(const *radix_integer_t* x, *slong* index, const *radix_t* radix)

`ulong radix_integer_get_digit(const radix_integer_t x, slong index, const radix_t radix)`

Returns the limb (respectively digit) at position *index* in *x*. The index must be nonnegative. The sign of *x* is ignored.

`void radix_integer_set_limb(radix_integer_t res, const radix_integer_t x, slong index, ulong c, const radix_t radix)`

`void radix_integer_set_digit(radix_integer_t res, const radix_integer_t x, slong index, ulong c, const radix_t radix)`

Sets *res* to *x* with the limb (respectively digit) at position *index* replaced by the given limb (respectively digit) *c*. The index must be nonnegative and *c* must be smaller than the limb radix (respectively digit radix). The sign of *x* is preserved. If *x* is zero and *c* is nonzero, the sign of the result will be positive.

`void radix_integer_lshift_limbs(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

`void radix_integer_lshift_digits(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

Sets *res* to *x* multiplied by B^n (respectively b^n).

`void radix_integer_rshift_limbs(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

`void radix_integer_rshift_digits(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

Sets *res* to *x* divided by B^n (respectively b^n), truncating the result (rounding to zero).

`slong radix_integer_valuation_limbs(const radix_integer_t x, const radix_t radix)`

`slong radix_integer_valuation_digits(const radix_integer_t x, const radix_t radix)`

Returns the largest *n* such that B^n (respectively b^n) divides *x* exactly, i.e. the number of least significant zero limbs (respectively digits). In the special case $x = 0$, returns zero.

`void radix_integer_trunc_limbs(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

Sets *res* to *x* with the absolute value reduced modulo B^n , preserving the sign of *x*.

`void radix_integer_mod_limbs(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

Sets *res* to *x* reduced modulo B^n , returning the unique unsigned representative in $[0, B^n - 1]$.

`void radix_integer_smod_limbs(radix_integer_t res, const radix_integer_t x, slong n, const radix_t radix)`

Sets *res* to *x* reduced modulo B^n , returning the unique signed representative in $[-\lfloor B^n/2 \rfloor, \lfloor B^n/2 \rfloor]$ if *B* is odd and in $[-B^n/2, B^n/2 - 1]$ if *B* is even.

`void radix_integer_mullow_limbs(radix_integer_t res, const radix_integer_t x, const radix_integer_t y, slong n, const radix_t radix)`

Sets *res* to the product of *x* and *y* with the absolute value reduced modulo B^n .

`int radix_integer_div(radix_integer_t q, const radix_integer_t a, const radix_integer_t b, const radix_t radix)`

Sets *q* to the quotient of *a* divided by *b* and returns 1 if the division is exact, otherwise sets *q* to zero and returns 0.

`int radix_integer_divexact(radix_integer_t q, const radix_integer_t a, const radix_integer_t b, const radix_t radix)`

Sets *q* to the quotient of *a* divided by *b* assuming that the division is exact. The behavior is undefined if the division is inexact.

`void radix_integer_tdiv_qr(radix_integer_t q, radix_integer_t r, const radix_integer_t a, const radix_integer_t b, const radix_t radix)`

```
void radix_integer_fdiv_qr(radix_integer_t q, radix_integer_t r, const radix_integer_t a, const  
    radix_integer_t b, const radix_t radix)  
void radix_integer_cdiv_qr(radix_integer_t q, radix_integer_t r, const radix_integer_t a, const  
    radix_integer_t b, const radix_t radix)  
void radix_integer_tdiv_q(radix_integer_t q, const radix_integer_t a, const radix_integer_t b,  
    const radix_t radix)  
void radix_integer_fdiv_q(radix_integer_t q, const radix_integer_t a, const radix_integer_t b,  
    const radix_t radix)  
void radix_integer_cdiv_q(radix_integer_t q, const radix_integer_t a, const radix_integer_t b,  
    const radix_t radix)  
void radix_integer_tdiv_r(radix_integer_t r, const radix_integer_t a, const radix_integer_t b,  
    const radix_t radix)  
void radix_integer_fdiv_r(radix_integer_t r, const radix_integer_t a, const radix_integer_t b,  
    const radix_t radix)  
void radix_integer_cdiv_r(radix_integer_t r, const radix_integer_t a, const radix_integer_t b,  
    const radix_t radix)
```

Division with remainder: compute $a = qb + r$ where q is the truncation, floor, and ceiling quotient respectively.

```
int radix_integer_invmod_limbs(radix_integer_t res, const radix_integer_t x, slong n, const  
    radix_t radix)
```

If x is invertible modulo B^n , sets res to the inverse and returns 1. The sign of the input is preserved. If x is not invertible, returns 0.

RATIONAL NUMBERS

5.1 `fmprq.h` – rational numbers

The `fmprq_t` data type represents rational numbers as fractions of multiprecision integers.

An `fmprq_t` is an array of length 1 of type `fmprq`, with `fmprq` being implemented as a pair of `fmprz`'s representing numerator and denominator.

This format is designed to allow rational numbers with small numerators or denominators to be stored and manipulated efficiently. When components no longer fit in single machine words, the cost of `fmprq_t` arithmetic is roughly the same as that of `mpq_t` arithmetic, plus a small amount of overhead.

A fraction is said to be in canonical form if the numerator and denominator have no common factor and the denominator is positive. Except where otherwise noted, all functions in the `fmprq` module assume that inputs are in canonical form, and produce outputs in canonical form. The user can manipulate the numerator and denominator of an `fmprq_t` as arbitrary integers, but then becomes responsible for canonicalising the number (for example by calling `fmprq_canonicalise`) before passing it to any library function.

For most operations, both a function operating on `fmprq_t`'s and an underscore version operating on `fmprz_t` components are provided. The underscore functions may perform less error checking, and may impose limitations on aliasing between the input and output variables, but generally assume that the components are in canonical form just like the non-underscore functions.

5.1.1 Types, macros and constants

type `fmprq`

An `fmprq` is implemented as a struct containing two `fmprz`'s, one for the numerator, and one for the denominator.

type `fmprq_t`

An array of length 1 of `fmprq`'s. This is used to pass `fmprq`'s around by reference without fuss, similar to the way `mpq_t`'s work.

`fmprz *fmprq_numref`(const `fmprq_t` x)

`fmprz *fmprq_denref`(const `fmprq_t` x)

Returns respectively a pointer to the numerator and denominator of x.

5.1.2 Memory management

void **fmpq_init**(*fmpq_t* x)

Initialises the *fmpq_t* variable *x* for use. Its value is set to 0.

void **fmpq_clear**(*fmpq_t* x)

Clears the *fmpq_t* variable *x*. To use the variable again, it must be re-initialised with **fmpq_init**.

5.1.3 Canonicalisation

void **fmpq_canonicalise**(*fmpq_t* res)

Puts *res* in canonical form: the numerator and denominator are reduced to lowest terms, and the denominator is made positive. If the numerator is zero, the denominator is set to one.

If the denominator is zero, the outcome of calling this function is undefined, regardless of the value of the numerator.

void **_fmpq_canonicalise**(*fmpz_t* num, *fmpz_t* den)

Does the same thing as **fmpq_canonicalise**, but for numerator and denominator given explicitly as *fmpz_t* variables. Aliasing of *num* and *den* is not allowed.

int **fmpq_is_canonical**(const *fmpq_t* x)

Returns nonzero if *fmpq_t* *x* is in canonical form (as produced by **fmpq_canonicalise**), and zero otherwise.

int **_fmpq_is_canonical**(const *fmpz_t* num, const *fmpz_t* den)

Does the same thing as **fmpq_is_canonical**, but for numerator and denominator given explicitly as *fmpz_t* variables.

5.1.4 Basic assignment

void **fmpq_set**(*fmpq_t* dest, const *fmpq_t* src)

Sets *dest* to a copy of *src*. No canonicalisation is performed.

void **fmpq_swap**(*fmpq_t* op1, *fmpq_t* op2)

Swaps the two rational numbers *op1* and *op2*.

void **fmpq_neg**(*fmpq_t* dest, const *fmpq_t* src)

Sets *dest* to the additive inverse of *src*.

void **fmpq_abs**(*fmpq_t* dest, const *fmpq_t* src)

Sets *dest* to the absolute value of *src*.

void **fmpq_zero**(*fmpq_t* res)

Sets the value of *res* to 0.

void **fmpq_one**(*fmpq_t* res)

Sets the value of *res* to 1.

5.1.5 Comparison

int `fmq_is_zero`(const *fmq_t* res)

Returns nonzero if `res` has value 0, and returns zero otherwise.

int `fmq_is_one`(const *fmq_t* res)

Returns nonzero if `res` has value 1, and returns zero otherwise.

int `fmq_is_pm1`(const *fmq_t* res)

Returns nonzero if `res` has value ± 1 and zero otherwise.

int `fmq_equal`(const *fmq_t* x, const *fmq_t* y)

int `fmq_equal_fmpz`(const *fmq_t* x, const *fmpz_t* y)

int `fmq_equal_si`(*fmq_t* x, *slong* y)

int `fmq_equal_ui`(*fmq_t* x, *ulong* y)

Returns nonzero if `x` and `y` are equal, and zero otherwise.

int `fmq_sgn`(const *fmq_t* x)

Returns the sign of the rational number `x`. That is, returns -1 if $x < 0$, 1 if $x > 0$ and 0 if $x = 0$.

int `fmq_cmp`(const *fmq_t* x, const *fmq_t* y)

int `fmq_cmp_fmpz`(const *fmq_t* x, const *fmpz_t* y)

int `fmq_cmp_si`(const *fmq_t* x, *slong* y)

int `fmq_cmp_ui`(const *fmq_t* x, *ulong* y)

Returns negative if $x < y$, zero if $x = y$, and positive if $x > y$.

void `fmq_height`(*fmpz_t* height, const *fmq_t* x)

Sets `height` to the height of `x`, defined as the larger of the absolute values of the numerator and denominator of `x`.

flint_bitcnt_t `fmq_height_bits`(const *fmq_t* x)

Returns the number of bits in the height of `x`.

5.1.6 Conversion

void `fmq_set_fmpz_frac`(*fmq_t* res, const *fmpz_t* p, const *fmpz_t* q)

Sets `res` to the canonical form of the fraction `p / q`. This is equivalent to assigning the numerator and denominator separately and calling `fmq_canonicalise`.

void `fmq_get_fmpz_frac`(*fmpz_t* a, *fmpz_t* b, *fmq_t* c)

Sets `a`, `b` to the numerator and denominator of `c` respectively.

Note: Requires that `gmp.h` has been included before any FLINT header is included.

void `fmq_set_si`(*fmq_t* res, *slong* p, *ulong* q)

Sets `res` to the canonical form of the fraction `p / q`.

void `_fmq_set_si`(*fmpz_t* rnum, *fmpz_t* rden, *slong* p, *ulong* q)

Sets (`rnum`, `rden`) to the canonical form of the fraction `p / q`. `rnum` and `rden` may not be aliased.

void `fmq_set_ui`(*fmq_t* res, *ulong* p, *ulong* q)

Sets `res` to the canonical form of the fraction `p / q`.

void `_fmq_set_ui`(*fmpz_t* rnum, *fmpz_t* rden, *ulong* p, *ulong* q)

Sets (`rnum`, `rden`) to the canonical form of the fraction `p / q`. `rnum` and `rden` may not be aliased.

void `fmq_set_mfq`(*fmq_t* dest, const *mfq_t* src)

Sets the value of `dest` to that of the `mfq_t` variable `src`.

Note: Requires that `gmp.h` has been included before any FLINT header is included.

int **fmpq_set_str**(*fmpq_t* dest, const char *s, int base)

Sets the value of **dest** to the value represented in the string **s** in base **base**.

Returns 0 if no error occurs. Otherwise returns -1 and **dest** is set to zero.

double **fmpq_get_d**(const *fmpq_t* f)

Returns *f* as a **double**, rounding towards zero if *f* cannot be represented exactly. The return is system dependent if *f* is too large or too small to fit in a **double**.

void **fmpq_get_mpq**(*mpq_t* dest, const *fmpq_t* src)

Sets the value of **dest**

Note: Requires that **gmp.h** has been included before any FLINT header is included.

int **fmpq_get_mpfr**(*mpfr_t* dest, const *fmpq_t* src, *mpfr_rnd_t* rnd)

Sets the MPFR variable **dest** to the value of **src**, rounded to the nearest representable binary floating-point value in direction **rnd**. Returns the sign of the rounding, according to MPFR conventions.

Note: Requires that **mpfr.h** has been included before any FLINT header is included.

char *_**fmpq_get_str**(char *str, int b, const *fmpz_t* num, const *fmpz_t* den)

char *_**fmpq_get_str**(char *str, int b, const *fmpq_t* x)

Prints the string representation of *x* in base $b \in [2, 36]$ to a suitable buffer.

If **str** is not NULL, this is used as the buffer and also the return value. If **str** is NULL, allocates sufficient space and returns a pointer to the string.

void **flint_mpq_init_set_readonly**(*mpq_t* z, const *fmpq_t* f)

Sets the uninitialised *mpq_t* *z* to the value of the readonly *fmpq_t* *f*.

Note that it is assumed that *f* does not change during the lifetime of *z*.

The rational *z* has to be cleared by a call to *flint_mpq_clear_readonly()*.

The suggested use of the two functions is as follows:

```
fmpq_t f;
...
{
    mpq_t z;

    flint_mpq_init_set_readonly(z, f);
    foo(..., z);
    flint_mpq_clear_readonly(z);
}
```

This provides a convenient function for user code, only requiring to work with the types *fmpq_t* and *mpq_t*.

Note: Requires that **gmp.h** has been included before any FLINT header is included.

void **flint_mpq_clear_readonly**(*mpq_t* z)

Clears the readonly *mpq_t* *z*.

Note: Requires that **gmp.h** has been included before any FLINT header is included.

void **fmpq_init_set_readonly**(*fmpq_t* f, const *mpq_t* z)

Sets the uninitialised *fmpq_t* *f* to a readonly version of the rational *z*.

Note that the value of *z* is assumed to remain constant throughout the lifetime of *f*.

The *fmpq_t* *f* has to be cleared by calling the function *fmpq_clear_readonly()*.

The suggested use of the two functions is as follows:

```

mpq_t z;
...
{
    fmpq_t f;

    fmpq_init_set_readonly(f, z);
    foo(..., f);
    fmpq_clear_readonly(f);
}

```

void **fmpq_clear_readonly**(*fmpq_t* f)
 Clears the readonly *fmpq_t* *f*.

5.1.7 Input and output

int **fmpq_fprint**(FILE *file, const *fmpq_t* x)
 Prints *x* as a fraction to the stream *file*. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

int **_fmpq_fprint**(FILE *file, const *fmpz_t* num, const *fmpz_t* den)
 Does the same thing as **fmpq_fprint**, but for numerator and denominator given explicitly as *fmpz_t* variables.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

int **fmpq_print**(const *fmpq_t* x)
 Prints *x* as a fraction. The numerator and denominator are printed verbatim as integers, with a forward slash (/) printed in between.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

int **_fmpq_print**(const *fmpz_t* num, const *fmpz_t* den)
 Does the same thing as **fmpq_print**, but for numerator and denominator given explicitly as *fmpz_t* variables.

In case of success, returns a positive number. In case of failure, returns a non-positive number.

5.1.8 Random number generation

void **fmpq_randtest**(*fmpq_t* res, *flint_rand_t* state, *flint_bitcnt_t* bits)
 Sets *res* to a random value, with numerator and denominator having up to *bits* bits. The resulting fraction will be in canonical form. This function has an increased probability of generating special values which are likely to trigger corner cases.

void **_fmpq_randtest**(*fmpz_t* num, *fmpz_t* den, *flint_rand_t* state, *flint_bitcnt_t* bits)
 Does the same thing as **fmpq_randtest**, but for numerator and denominator given explicitly as *fmpz_t* variables. Aliasing of *num* and *den* is not allowed.

void **fmpq_randtest_not_zero**(*fmpq_t* res, *flint_rand_t* state, *flint_bitcnt_t* bits)
 As per **fmpq_randtest**, but the result will not be 0. If *bits* is set to 0, an exception will result.

void **fmpq_randbits**(*fmpq_t* res, *flint_rand_t* state, *flint_bitcnt_t* bits)
 Sets *res* to a random value, with numerator and denominator both having exactly *bits* bits before canonicalisation, and then puts *res* in canonical form. Note that as a result of the canonicalisation, the resulting numerator and denominator can be slightly smaller than *bits* bits.

void `_fmpq_randbits`(*fmpz_t* num, *fmpz_t* den, *flint_rand_t* state, *flint_bitcnt_t* bits)

Does the same thing as `fmpq_randbits`, but for numerator and denominator given explicitly as *fmpz_t* variables. Aliasing of *num* and *den* is not allowed.

5.1.9 Arithmetic

void `fmpq_add`(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)

void `fmpq_sub`(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)

void `fmpq_mul`(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)

void `fmpq_div`(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)

Sets *res* respectively to $op1 + op2$, $op1 - op2$, $op1 * op2$, or $op1 / op2$. Division by zero results in an error. Aliasing between any combination of the variables is allowed.

void `_fmpq_add`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* op1num, const *fmpz_t* op1den, const *fmpz_t* op2num, const *fmpz_t* op2den)

void `_fmpq_sub`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* op1num, const *fmpz_t* op1den, const *fmpz_t* op2num, const *fmpz_t* op2den)

void `_fmpq_mul`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* op1num, const *fmpz_t* op1den, const *fmpz_t* op2num, const *fmpz_t* op2den)

void `_fmpq_div`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* op1num, const *fmpz_t* op1den, const *fmpz_t* op2num, const *fmpz_t* op2den)

Sets (*rnum*, *rden*) to the canonical form of the sum, difference, product or quotient respectively of the fractions represented by (*op1num*, *op1den*) and (*op2num*, *op2den*). Aliasing between any combination of the variables is allowed, whilst no numerator is aliased with a denominator.

void `_fmpq_add_si`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, *slong* r)

void `_fmpq_sub_si`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, *slong* r)

void `_fmpq_add_ui`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, *ulong* r)

void `_fmpq_sub_ui`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, *ulong* r)

void `_fmpq_add_fmpz`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, const *fmpz_t* r)

void `_fmpq_sub_fmpz`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, const *fmpz_t* r)

Sets (*rnum*, *rden*) to the canonical form of the sum or difference respectively of the fractions represented by (*p*, *q*) and (*r*, 1). Numerators may not be aliased with denominators.

void `fmpq_add_si`(*fmpq_t* res, const *fmpq_t* op1, *slong* c)

void `fmpq_sub_si`(*fmpq_t* res, const *fmpq_t* op1, *slong* c)

void `fmpq_add_ui`(*fmpq_t* res, const *fmpq_t* op1, *ulong* c)

void `fmpq_sub_ui`(*fmpq_t* res, const *fmpq_t* op1, *ulong* c)

void `fmpq_add_fmpz`(*fmpq_t* res, const *fmpq_t* op1, const *fmpz_t* c)

void `fmpq_sub_fmpz`(*fmpq_t* res, const *fmpq_t* op1, const *fmpz_t* c)

Sets *res* to the sum or difference respectively of the fraction *op1* and the integer *c*.

void `_fmpq_mul_si`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, *slong* r)

Sets (*rnum*, *rden*) to the product of (*p*, *q*) and the integer *r*.

void `fmpq_mul_si`(*fmpq_t* res, const *fmpq_t* op1, *slong* c)

Sets *res* to the product of *op1* and the integer *c*.

void `_fmpq_mul_ui`(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, *ulong* r)

Sets (*rnum*, *rden*) to the product of (*p*, *q*) and the integer *r*.

void `fmpq_mul_ui`(*fmpq_t* res, const *fmpq_t* op1, *ulong* c)

Sets *res* to the product of *op1* and the integer *c*.

void `fmpq_addmul`(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)

void **fmpq_submul**(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)
 Sets *res* to $\text{res} + \text{op1} * \text{op2}$ or $\text{res} - \text{op1} * \text{op2}$, respectively. Aliasing between any combination of the variables is allowed.

void **_fmpq_addmul**(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* op1num, const *fmpz_t* op1den, const *fmpz_t* op2num, const *fmpz_t* op2den)
 void **_fmpq_submul**(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* op1num, const *fmpz_t* op1den, const *fmpz_t* op2num, const *fmpz_t* op2den)
 Sets (rnum, rden) to the canonical form of the fraction (rnum, rden) + (op1num, op1den) * (op2num, op2den) or (rnum, rden) - (op1num, op1den) * (op2num, op2den) respectively. Aliasing between any combination of the variables is allowed, whilst no numerator is aliased with a denominator.

void **fmpq_inv**(*fmpq_t* dest, const *fmpq_t* src)
 Sets *dest* to $1 / \text{src}$.

void **_fmpq_pow_si**(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* opnum, const *fmpz_t* opden, *slong* e)
 void **fmpq_pow_si**(*fmpq_t* res, const *fmpq_t* op, *slong* e)
 Sets *res* to *op* raised to the power *e*, where *e* is a *slong*. If *e* is 0 and *op* is 0, then *res* will be set to 1.

int **fmpq_pow_fmpz**(*fmpq_t* a, const *fmpq_t* b, const *fmpz_t* e)
 Set *res* to *op* raised to the power *e*. Return 1 for success and 0 for failure.

void **fmpq_mul_fmpz**(*fmpq_t* res, const *fmpq_t* op, const *fmpz_t* x)
 Sets *res* to the product of the rational number *op* and the integer *x*.

void **fmpq_div_fmpz**(*fmpq_t* res, const *fmpq_t* op, const *fmpz_t* x)
 Sets *res* to the quotient of the rational number *op* and the integer *x*.

void **fmpq_mul_2exp**(*fmpq_t* res, const *fmpq_t* x, *flint_bitcnt_t* exp)
 Sets *res* to *x* multiplied by 2^{exp} .

void **fmpq_div_2exp**(*fmpq_t* res, const *fmpq_t* x, *flint_bitcnt_t* exp)
 Sets *res* to *x* divided by 2^{exp} .

void **_fmpq_gcd**(*fmpz_t* rnum, *fmpz_t* rden, const *fmpz_t* p, const *fmpz_t* q, const *fmpz_t* r, const *fmpz_t* s)
 Set (rnum, rden) to the gcd of (p, q) and (r, s) which we define to be the canonicalisation of $\text{gcd}(ps, qr) / (qs)$. Does not assume that (rnum, rden), (p, q) or (r, s) are canonical. (This is apparently Euclid's original definition and is stable under scaling of numerator and denominator. It also agrees with the gcd on the integers. Note that it does not agree with gcd as defined in *fmpq_poly*.) This definition agrees with the result as output by Sage and Pari/GP.

void **fmpq_gcd**(*fmpq_t* res, const *fmpq_t* op1, const *fmpq_t* op2)
 Set *res* to the gcd of *op1* and *op2*. See the low level function **_fmpq_gcd** for our definition of gcd.

void **_fmpq_gcd_cofactors**(*fmpz_t* gnum, *fmpz_t* gden, *fmpz_t* abar, *fmpz_t* bbar, const *fmpz_t* anum, const *fmpz_t* aden, const *fmpz_t* bnum, const *fmpz_t* bden)
 void **fmpq_gcd_cofactors**(*fmpq_t* g, *fmpz_t* abar, *fmpz_t* bbar, const *fmpq_t* a, const *fmpq_t* b)
 Set *g* to $\text{gcd}(a, b)$ as per **fmpq_gcd()** and also compute $\bar{a} = a/g$ and $\bar{b} = b/g$. Unlike **_fmpq_gcd()**, **_fmpq_gcd_cofactors()** requires canonical inputs.

void **_fmpq_add_small**(*fmpz_t* rnum, *fmpz_t* rden, *slong* p1, *ulong* q1, *slong* p2, *ulong* q2)
 Sets (rnum, rden) to the sum of (p1, q1) and (p2, q2). Assumes that (p1, q1) and (p2, q2) are in canonical form and that all inputs are between COEFF_MIN and COEFF_MAX.

void **_fmpq_mul_small**(*fmpz_t* rnum, *fmpz_t* rden, *slong* p1, *ulong* q1, *slong* p2, *ulong* q2)
 Sets (rnum, rden) to the product of (p1, q1) and (p2, q2). Assumes that (p1, q1) and (p2, q2) are in canonical form and that all inputs are between COEFF_MIN and COEFF_MAX.

5.1.10 Modular reduction and rational reconstruction

```
int _fmpz_mod_fmpz(fmpz_t res, const fmpz_t num, const fmpz_t den, const fmpz_t mod)
```

```
int fmpz_mod_fmpz(fmpz_t res, const fmpz_t x, const fmpz_t mod)
```

Sets the integer `res` to the residue a of $x = n/d = (\text{num}, \text{den})$ modulo the positive integer $m = \text{mod}$, defined as the $0 \leq a < m$ satisfying $n \equiv ad \pmod{m}$. If such an a exists, 1 will be returned, otherwise 0 will be returned.

```
int _fmpz_reconstruct_fmpz_2_naive(fmpz_t n, fmpz_t d, const fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)
```

```
int _fmpz_reconstruct_fmpz_2(fmpz_t n, fmpz_t d, const fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)
```

```
int fmpz_reconstruct_fmpz_2(fmpz_t res, const fmpz_t a, const fmpz_t m, const fmpz_t N, const fmpz_t D)
```

Reconstructs a rational number from its residue a modulo m .

Given a modulus $m > 2$, a residue $0 \leq a < m$, and positive N, D satisfying $2ND < m$, this function attempts to find a fraction n/d with $0 \leq |n| \leq N$ and $0 < d \leq D$ such that $\gcd(n, d) = 1$ and $n \equiv ad \pmod{m}$. If a solution exists, then it is also unique. The function returns 1 if successful, and 0 to indicate that no solution exists.

```
int _fmpz_reconstruct_fmpz(fmpz_t n, fmpz_t d, const fmpz_t a, const fmpz_t m)
```

```
int fmpz_reconstruct_fmpz(fmpz_t res, const fmpz_t a, const fmpz_t m)
```

Reconstructs a rational number from its residue a modulo m , returning 1 if successful and 0 if no solution exists. Uses the balanced bounds $N = D = \lfloor \sqrt{\frac{m-1}{2}} \rfloor$.

5.1.11 Rational enumeration

```
void _fmpz_next_minimal(fmpz_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
```

```
void fmpz_next_minimal(fmpz_t res, const fmpz_t x)
```

Given $x = \text{num}/\text{den}$, assumed to be nonnegative and in canonical form, sets `res` to the next rational number in the sequence obtained by enumerating all positive denominators q , for each q enumerating the numerators $1 \leq p < q$ in order and generating both p/q and q/p , but skipping all $\gcd(p, q) \neq 1$. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

0, 1, 1/2, 2, 1/3, 3, 2/3, 3/2, 1/4, 4, 3/4, 4/3, 1/5, 5, 2/5, ...

This enumeration produces the rational numbers in order of minimal height. It has the disadvantage of being somewhat slower to compute than the Calkin-Wilf enumeration.

```
void _fmpz_next_signed_minimal(fmpz_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
```

```
void fmpz_next_signed_minimal(fmpz_t res, const fmpz_t x)
```

Given a signed rational number $x = \text{num}/\text{den}$, assumed to be in canonical form, sets `res` to the next element in the minimal-height sequence generated by `fmpz_next_minimal` but with negative numbers interleaved:

0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, ...

Starting with zero, this generates every rational number once and only once, in order of minimal height.

```
void _fmpz_next_calkin_wilf(fmpz_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
```

```
void fmpz_next_calkin_wilf(fmpz_t res, const fmpz_t x)
```

Given $x = \text{num}/\text{den}$, which is assumed to be nonnegative and in canonical form, sets `res` to the next number in the breadth-first traversal of the Calkin-Wilf tree. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

$0, 1, 1/2, 2, 1/3, 3/2, 2/3, 3, 1/4, 4/3, 3/5, 5/2, 2/5, \dots$

Despite the appearance of the initial entries, the Calkin-Wilf enumeration does not produce the rational numbers in order of height: some small fractions will appear late in the sequence. This order has the advantage of being faster to produce than the minimal-height order.

```
void _fmpq_next_signed_calkin_wilf(fmpz_t rnum, fmpz_t rden, const fmpz_t num, const fmpz_t den)
```

```
void fmpq_next_signed_calkin_wilf(fmpq_t res, const fmpq_t x)
```

Given a signed rational number $x = \text{num}/\text{den}$, assumed to be in canonical form, sets **res** to the next element in the Calkin-Wilf sequence with negative numbers interleaved:

$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \dots$

Starting with zero, this generates every rational number once and only once, but not in order of minimal height.

```
void fmpq_farey_neighbors(fmpq_t l, fmpq_t r, const fmpq_t x, const fmpz_t Q)
```

Set l and r to the fractions directly below and above x in the Farey sequence of order Q . This function will throw if Q is less than the denominator of x .

```
void _fmpq_simplest_between(fmpz_t x_num, fmpz_t x_den, const fmpz_t l_num, const fmpz_t l_den, const fmpz_t r_num, const fmpz_t r_den)
```

```
void fmpq_simplest_between(fmpq_t x, const fmpq_t l, const fmpq_t r)
```

Set x to the simplest fraction in the closed interval $[l, r]$. The underscore version makes the additional assumption that $l \leq r$. The endpoints l and r do not need to be canonical, but their denominators do need to be positive. x will always be returned in canonical form. A canonical fraction a_1/b_1 is defined to be simpler than a_2/b_2 iff $b_1 < b_2$ or $b_1 = b_2$ and $a_1 < a_2$.

5.1.12 Continued fractions

```
slong fmpq_get_cfrac(fmpz *c, fmpq_t rem, const fmpq_t x, slong n)
```

```
slong fmpq_get_cfrac_naive(fmpz *c, fmpq_t rem, const fmpq_t x, slong n)
```

Generates up to n terms of the (simple) continued fraction expansion of x , writing the coefficients to the vector c and the remainder r to the **rem** variable. The return value is the number k of generated terms. The output satisfies

$$x = c_0 + \frac{1}{c_1 + \frac{1}{c_2 + \frac{1}{\ddots + \frac{1}{c_{k-1} + r}}}}$$

If r is zero, the continued fraction expansion is complete. If r is nonzero, $1/r$ can be passed back as input to generate $c_k, c_{k+1}, \dots$. Calls to `fmpq_get_cfrac` can therefore be chained to generate the continued fraction incrementally, extracting any desired number of coefficients at a time.

In general, a rational number has exactly two continued fraction expansions. By convention, we generate the shorter one. The longer expansion can be obtained by replacing the last coefficient a_{k-1} by the pair of coefficients $a_{k-1} - 1, 1$.

The behavior of this function in corner cases is as follows:

- if x is infinite (anything over 0), **rem** will be zero and the return is $k = 0$ regardless of n .
- else (if x is finite),
 - if $n \leq 0$, **rem** will be $1/x$ (allowing for infinite in the case $x = 0$) and the return is $k = 0$
 - else (if $n > 0$), **rem** will finite and the return is $0 < k \leq n$.

Essentially, if this function is called with canonical x and $n > 0$, then `rem` will be canonical. Therefore, applications relying on canonical `fmpr_t`'s should not call this function with $n \leq 0$.

void `fmpr_set_cfrac`(*fmpr_t* x, const *fmpr* *c, *slong* n)

Sets x to the value of the continued fraction

$$x = c_0 + \frac{1}{c_1 + \frac{1}{c_2 + \frac{1}{\ddots + \frac{1}{c_{n-1}}}}}$$

where all c_i except c_0 should be nonnegative. It is assumed that $n > 0$.

For large n , this function implements a subquadratic algorithm. The convergents are given by a chain product of 2 by 2 matrices. This product is split in half recursively to balance the size of the coefficients.

slong `fmpr_cfrac_bound`(const *fmpr_t* x)

Returns an upper bound for the number of terms in the continued fraction expansion of x . The computed bound is not necessarily sharp.

We use the fact that the smallest denominator that can give a continued fraction of length n is the Fibonacci number F_{n+1} .

5.1.13 Special functions

void `_fmpr_harmonic_ui`(*fmpr_t* num, *fmpr_t* den, *ulong* n)

void `fmpr_harmonic_ui`(*fmpr_t* x, *ulong* n)

Computes the harmonic number $H_n = 1 + 1/2 + 1/3 + \dots + 1/n$. Table lookup is used for H_n whose numerator and denominator fit in single limb. For larger n , a divide and conquer strategy is used.

5.1.14 Dedekind sums

Most of the definitions and relations used in the following section are given by Apostol [Apostol1997]. The Dedekind sum $s(h, k)$ is defined for all integers h and k as

$$s(h, k) = \sum_{i=1}^{k-1} \left(\left(\frac{i}{k} \right) \right) \left(\left(\frac{hi}{k} \right) \right)$$

where

$$\left(\left(x \right) \right) = \begin{cases} x - [x] - 1/2 & \text{if } x \in \mathbf{Q} \setminus \mathbf{Z} \\ 0 & \text{if } x \in \mathbf{Z}. \end{cases}$$

If $0 < h < k$ and $(h, k) = 1$, this reduces to

$$s(h, k) = \sum_{i=1}^{k-1} \frac{i}{k} \left(\frac{hi}{k} - \left\lfloor \frac{hi}{k} \right\rfloor - \frac{1}{2} \right).$$

The main formula for evaluating the series above is the following. Letting $r_0 = k$, $r_1 = h$, $r_2, r_3, \dots, r_n, r_{n+1} = 1$ be the remainder sequence in the Euclidean algorithm for computing GCD of h and k ,

$$s(h, k) = \frac{1 - (-1)^n}{8} - \frac{1}{12} \sum_{i=1}^{n+1} (-1)^i \left(\frac{1 + r_i^2 + r_{i-1}^2}{r_i r_{i-1}} \right).$$

Writing $s(h, k) = p/q$, some useful properties employed are $|s| < k/12$, $q \mid 6k$ and $2|p| < k^2$.

void **fmprq_dedekind_sum**(*fmprq_t* s, const *fmprz_t* h, const *fmprz_t* k)

void **fmprq_dedekind_sum_naive**(*fmprq_t* s, const *fmprz_t* h, const *fmprz_t* k)

Computes $s(h, k)$ for arbitrary h and k . The naive version uses a straightforward implementation of the defining sum using **fmprz** arithmetic and is slow for large k .

5.2 fmprq_vec.h – vectors over rational numbers

5.2.1 Memory management

fmprq *_**fmprq_vec_init**(*slong* n)

Initialises a vector of **fmprq** values of length n and sets all values to 0. This is equivalent to generating a **fmprz** vector of length $2n$ with **_fmprz_vec_init** and setting all denominators to 1.

void **_fmprq_vec_clear**(*fmprq* *vec, *slong* n)

Frees an **fmprq** vector.

5.2.2 Randomisation

void **_fmprq_vec_randtest**(*fmprq* *f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets the entries of a vector of the given length to random rationals with numerator and denominator having up to the given number of bits per entry.

void **_fmprq_vec_randtest_uniq_sorted**(*fmprq* *vec, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets the entries of a vector of the given length to random distinct rationals with numerator and denominator having up to the given number of bits per entry. The entries in the vector are sorted.

5.2.3 Bit sizes and heights

void **_fmprq_vec_max_height**(*fmprz_t* height, const *fmprq* *vec, *slong* len);

Computes the maximum of the height of any coefficient of (vec, len), each height being computed by **fmprq_height()**.

flint_bitcnt_t **_fmprq_vec_max_height_bits**(const *fmprq* *vec, *slong* len);

Computes the maximum number of bits of the height of any coefficient of (vec, len), each one being computed by **fmprq_height_bits()**.

5.2.4 Comparison

int **_fmprq_vec_equal**(const *fmprq* *vec1, const *fmprq* *vec2, *slong* len)

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

5.2.5 Sorting

void **_fmprq_vec_sort**(*fmprq* *vec, *slong* len)

Sorts the entries of (vec, len).

5.2.6 Conversions

`void _fmpq_vec_set_fmpz_vec(fmpz *res, const fmpz *vec, slong len)`

Sets (res, len) to (vec, len).

`void _fmpq_vec_get_fmpz_vec_fmpz(fmpz *num, fmpz_t den, const fmpq *a, slong len)`

Find a common denominator den of the entries of a and set (num, len) to the corresponding numerators.

5.2.7 Dot product

`void _fmpq_vec_dot(fmpz_t res, const fmpq *vec1, const fmpq *vec2, slong len)`

Sets res to the dot product of the vectors (vec1, len) and (vec2, len).

5.2.8 Input and output

`int _fmpq_vec_fprint(FILE *file, const fmpq *vec, slong len)`

Prints the vector of given length to the stream file. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int _fmpq_vec_print(const fmpq *vec, slong len)`

Prints the vector of given length to stdout.

For further details, see `_fmpq_vec_fprint()`.

5.3 fmpq_mat.h – matrices over the rational numbers

The `fmpq_mat_t` data type represents matrices over $\mathbb{Q}$.

A rational matrix is stored as an array of `fmpq` elements in order to allow convenient and efficient manipulation of individual entries. In general, `fmpq_mat` functions assume that input entries are in canonical form, and produce output with entries in canonical form.

Since rational arithmetic is expensive, computations are typically performed by clearing denominators, performing the heavy work over the integers, and converting the final result back to a rational matrix. The `fmpq_mat` functions take care of such conversions transparently. For users who need fine-grained control, various functions for conversion between rational and integer matrices are provided.

5.3.1 Types, macros and constants

`type fmpq_mat_struct`

`type fmpq_mat_t`

5.3.2 Memory management

void **fmpq_mat_init**(*fmpq_mat_t* mat, *slong* rows, *slong* cols)
Initialises a matrix with the given number of rows and columns for use.

void **fmpq_mat_init_set**(*fmpq_mat_t* mat1, const *fmpq_mat_t* mat2)
Initialises mat1 and sets it equal to mat2.

void **fmpq_mat_clear**(*fmpq_mat_t* mat)
Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

void **fmpq_mat_swap**(*fmpq_mat_t* mat1, *fmpq_mat_t* mat2)
Swaps two matrices. The dimensions of mat1 and mat2 are allowed to be different.

void **fmpq_mat_swap_entrywise**(*fmpq_mat_t* mat1, *fmpq_mat_t* mat2)
Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

5.3.3 Entry access

fmpq ***fmpq_mat_entry**(const *fmpq_mat_t* mat, *slong* i, *slong* j)
Gives a reference to the entry at row i and column j. The reference can be passed as an input or output variable to any **fmpq** function for direct manipulation of the matrix element. No bounds checking is performed.

fmpz ***fmpq_mat_entry_num**(const *fmpq_mat_t* mat, *slong* i, *slong* j)
Gives a reference to the numerator of the entry at row i and column j. The reference can be passed as an input or output variable to any **fmpz** function for direct manipulation of the matrix element. No bounds checking is performed.

fmpz ***fmpq_mat_entry_den**(const *fmpq_mat_t* mat, *slong* i, *slong* j)
Gives a reference to the denominator of the entry at row i and column j. The reference can be passed as an input or output variable to any **fmpz** function for direct manipulation of the matrix element. No bounds checking is performed.

slong **fmpq_mat_nrows**(const *fmpq_mat_t* mat)
Return the number of rows of the matrix mat.

slong **fmpq_mat_ncols**(const *fmpq_mat_t* mat)
Return the number of columns of the matrix mat.

5.3.4 Basic assignment

void **fmpq_mat_set**(*fmpq_mat_t* dest, const *fmpq_mat_t* src)
Sets the entries in **dest** to the same values as in **src**, assuming the two matrices have the same dimensions.

void **fmpq_mat_zero**(*fmpq_mat_t* mat)
Sets mat to the zero matrix.

void **fmpq_mat_one**(*fmpq_mat_t* mat)
Let m be the minimum of the number of rows and columns in the matrix mat. This function sets the first $m \times m$ block to the identity matrix, and the remaining block to zero.

void **fmpq_mat_transpose**(*fmpq_mat_t* rop, const *fmpq_mat_t* op)
Sets the matrix **rop** to the transpose of the matrix **op**, assuming that their dimensions are compatible. Aliasing is allowed for square matrices.

void **fmpq_mat_swap_rows**(*fmpq_mat_t* mat, *slong* *perm, *slong* r, *slong* s)
Swaps rows *r* and *s* of *mat*. If *perm* is non-NULL, the permutation of the rows will also be applied to *perm*.

void **fmpq_mat_swap_cols**(*fmpq_mat_t* mat, *slong* *perm, *slong* r, *slong* s)
Swaps columns *r* and *s* of *mat*. If *perm* is non-NULL, the permutation of the columns will also be applied to *perm*.

void **fmpq_mat_invert_rows**(*fmpq_mat_t* mat, *slong* *perm)
Swaps rows *i* and *r - i* of *mat* for $0 \leq i < r/2$, where *r* is the number of rows of *mat*. If *perm* is non-NULL, the permutation of the rows will also be applied to *perm*.

void **fmpq_mat_invert_cols**(*fmpq_mat_t* mat, *slong* *perm)
Swaps columns *i* and *c - i* of *mat* for $0 \leq i < c/2$, where *c* is the number of columns of *mat*. If *perm* is non-NULL, the permutation of the columns will also be applied to *perm*.

5.3.5 Addition, scalar multiplication

void **fmpq_mat_add**(*fmpq_mat_t* mat, const *fmpq_mat_t* mat1, const *fmpq_mat_t* mat2)
Sets *mat* to the sum of *mat1* and *mat2*, assuming that all three matrices have the same dimensions.

void **fmpq_mat_sub**(*fmpq_mat_t* mat, const *fmpq_mat_t* mat1, const *fmpq_mat_t* mat2)
Sets *mat* to the difference of *mat1* and *mat2*, assuming that all three matrices have the same dimensions.

void **fmpq_mat_neg**(*fmpq_mat_t* rop, const *fmpq_mat_t* op)
Sets *rop* to the negative of *op*, assuming that the two matrices have the same dimensions.

void **fmpq_mat_scalar_mul_fmpq**(*fmpq_mat_t* rop, const *fmpq_mat_t* op, const *fmpq_t* x)
Sets *rop* to *op* multiplied by the rational *x*, assuming that the two matrices have the same dimensions.

Note that the rational *x* may not be aliased with any part of the entries of *rop*.

void **fmpq_mat_scalar_mul_fmpz**(*fmpq_mat_t* rop, const *fmpq_mat_t* op, const *fmpz_t* x)
Sets *rop* to *op* multiplied by the integer *x*, assuming that the two matrices have the same dimensions.

Note that the integer *x* may not be aliased with any part of the entries of *rop*.

void **fmpq_mat_scalar_div_fmpz**(*fmpq_mat_t* rop, const *fmpq_mat_t* op, const *fmpz_t* x)
Sets *rop* to *op* divided by the integer *x*, assuming that the two matrices have the same dimensions and that *x* is non-zero.

Note that the integer *x* may not be aliased with any part of the entries of *rop*.

5.3.6 Input and output

void **fmpq_mat_print**(const *fmpq_mat_t* mat)
Prints the matrix *mat* to standard output.

5.3.7 Random matrix generation

void **fmprq_mat_randbits**(*fmprq_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits)

This is equivalent to applying **fmprq_randbits** to all entries in the matrix.

void **fmprq_mat_randtest**(*fmprq_mat_t* mat, *flint_rand_t* state, *flint_bitcnt_t* bits)

This is equivalent to applying **fmprq_randtest** to all entries in the matrix.

5.3.8 Window

void **fmprq_mat_window_init**(*fmprq_mat_t* window, const *fmprq_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2)

Initializes the matrix **window** to be an $r2 - r1$ by $c2 - c1$ submatrix of **mat** whose (0,0) entry is the (r1, c1) entry of **mat**. The memory for the elements of **window** is shared with **mat**.

void **fmprq_mat_window_clear**(*fmprq_mat_t* window)

Clears the matrix **window** and releases any memory that it uses. Note that the memory to the underlying matrix that **window** points to is not freed.

5.3.9 Concatenate

void **fmprq_mat_concat_vertical**(*fmprq_mat_t* res, const *fmprq_mat_t* mat1, const *fmprq_mat_t* mat2)

Sets **res** to vertical concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions: **mat1**: $m \times n$, **mat2**: $k \times n$, **res**: $(m + k) \times n$.

void **fmprq_mat_concat_horizontal**(*fmprq_mat_t* res, const *fmprq_mat_t* mat1, const *fmprq_mat_t* mat2)

Sets **res** to horizontal concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions: **mat1**: $m \times n$, **mat2**: $m \times k$, **res**: $m \times (n + k)$.

5.3.10 Special matrices

void **fmprq_mat_hilbert_matrix**(*fmprq_mat_t* mat)

Sets **mat** to a Hilbert matrix of the given size. That is, the entry at row i and column j is set to $1/(i + j + 1)$.

5.3.11 Basic comparison and properties

int **fmprq_mat_equal**(const *fmprq_mat_t* mat1, const *fmprq_mat_t* mat2)

Returns nonzero if **mat1** and **mat2** have the same shape and all their entries agree, and returns zero otherwise. Assumes the entries in both **mat1** and **mat2** are in canonical form.

int **fmprq_mat_is_integral**(const *fmprq_mat_t* mat)

Returns nonzero if all entries in **mat** are integer-valued, and returns zero otherwise. Assumes that the entries in **mat** are in canonical form.

int **fmprq_mat_is_zero**(const *fmprq_mat_t* mat)

Returns nonzero if all entries in **mat** are zero, and returns zero otherwise.

int **fmprq_mat_is_one**(const *fmprq_mat_t* mat)

Returns nonzero if **mat** ones along the diagonal and zeros elsewhere, and returns zero otherwise.

int **fmpq_mat_is_empty**(const *fmpq_mat_t* mat)

Returns a non-zero value if the number of rows or the number of columns in **mat** is zero, and otherwise returns zero.

int **fmpq_mat_is_square**(const *fmpq_mat_t* mat)

Returns a non-zero value if the number of rows is equal to the number of columns in **mat**, and otherwise returns zero.

5.3.12 Integer matrix conversion

int **fmpq_mat_get_fmpz_mat**(*fmpz_mat_t* dest, const *fmpq_mat_t* mat)

Sets **dest** to **mat** and returns nonzero if all entries in **mat** are integer-valued. If not all entries in **mat** are integer-valued, sets **dest** to an undefined matrix and returns zero. Assumes that the entries in **mat** are in canonical form.

void **fmpq_mat_get_fmpz_mat_entrywise**(*fmpz_mat_t* num, *fmpz_mat_t* den, const *fmpq_mat_t* mat)

Sets the integer matrices **num** and **den** respectively to the numerators and denominators of the entries in **mat**.

void **fmpq_mat_get_fmpz_mat_matwise**(*fmpz_mat_t* num, *fmpz_t* den, const *fmpq_mat_t* mat)

Converts all entries in **mat** to a common denominator, storing the rescaled numerators in **num** and the denominator in **den**. The denominator will be minimal if the entries in **mat** are in canonical form.

void **fmpq_mat_get_fmpz_mat_rowwise**(*fmpz_mat_t* num, *fmpz_t* *den, const *fmpq_mat_t* mat)

Clears denominators in **mat** row by row. The rescaled numerators are written to **num**, and the denominator of row **i** is written to position **i** in **den** which can be a preinitialised **fmpz** vector. Alternatively, NULL can be passed as the **den** variable, in which case the denominators will not be stored.

void **fmpq_mat_get_fmpz_mat_rowwise_2**(*fmpz_mat_t* num, *fmpz_mat_t* num2, *fmpz_t* *den, const *fmpq_mat_t* mat, const *fmpq_mat_t* mat2)

Clears denominators row by row of both **mat** and **mat2**, writing the respective numerators to **num** and **num2**. This is equivalent to concatenating **mat** and **mat2** horizontally, calling **fmpq_mat_get_fmpz_mat_rowwise**, and extracting the two submatrices in the result.

void **fmpq_mat_get_fmpz_mat_colwise**(*fmpz_mat_t* num, *fmpz_t* *den, const *fmpq_mat_t* mat)

Clears denominators in **mat** column by column. The rescaled numerators are written to **num**, and the denominator of column **i** is written to position **i** in **den** which can be a preinitialised **fmpz** vector. Alternatively, NULL can be passed as the **den** variable, in which case the denominators will not be stored.

void **fmpq_mat_set_fmpz_mat**(*fmpq_mat_t* dest, const *fmpz_mat_t* src)

Sets **dest** to **src**.

void **fmpq_mat_set_fmpz_mat_div_fmpz**(*fmpq_mat_t* mat, const *fmpz_mat_t* num, const *fmpz_t* den)

Sets **mat** to the integer matrix **num** divided by the common denominator **den**.

5.3.13 Modular reduction and rational reconstruction

void `fmprq_mat_get_fmpz_mat_mod_fmpz`(*fmprz_mat_t* dest, const *fmprq_mat_t* mat, const *fmpz_t* mod)

Sets each entry in `dest` to the corresponding entry in `mat`, reduced modulo `mod`.

int `fmprq_mat_set_fmpz_mat_mod_fmpz`(*fmprq_mat_t* X, const *fmpz_mat_t* Xmod, const *fmpz_t* mod)

Sets `X` to the entrywise rational reconstruction integer matrix `Xmod` modulo `mod`, and returns nonzero if the reconstruction is successful. If rational reconstruction fails for any element, returns zero and sets the entries in `X` to undefined values.

5.3.14 Matrix multiplication

void `fmprq_mat_mul_direct`(*fmprq_mat_t* C, const *fmprq_mat_t* A, const *fmprq_mat_t* B)

Sets `C` to the matrix product `AB`, computed naively using rational arithmetic. This is typically very slow and should only be used in circumstances where clearing denominators would consume too much memory.

void `fmprq_mat_mul_cleared`(*fmprq_mat_t* C, const *fmprq_mat_t* A, const *fmprq_mat_t* B)

Sets `C` to the matrix product `AB`, computed by clearing denominators and multiplying over the integers.

void `fmprq_mat_mul`(*fmprq_mat_t* C, const *fmprq_mat_t* A, const *fmprq_mat_t* B)

Sets `C` to the matrix product `AB`. This simply calls `fmprq_mat_mul_cleared`.

void `fmprq_mat_mul_fmpz_mat`(*fmprq_mat_t* C, const *fmprq_mat_t* A, const *fmpz_mat_t* B)

Sets `C` to the matrix product `AB`, with `B` an integer matrix. This function works efficiently by clearing denominators of `A`.

void `fmprq_mat_mul_r_fmpz_mat`(*fmprq_mat_t* C, const *fmpz_mat_t* A, const *fmprq_mat_t* B)

Sets `C` to the matrix product `AB`, with `A` an integer matrix. This function works efficiently by clearing denominators of `B`.

void `fmprq_mat_mul_fmpq_vec`(*fmpq* *c, const *fmprq_mat_t* A, const *fmpq* *b, *slong* blen)

void `fmprq_mat_mul_fmpz_vec`(*fmpq* *c, const *fmprq_mat_t* A, const *fmpz* *b, *slong* blen)

void `fmprq_mat_mul_fmpq_vec_ptr`(*fmpq* *const *c, const *fmprq_mat_t* A, const *fmpq* *const *b, *slong* blen)

void `fmprq_mat_mul_fmpz_vec_ptr`(*fmpq* *const *c, const *fmprq_mat_t* A, const *fmpz* *const *b, *slong* blen)

Compute a matrix-vector product of `A` and `(b, blen)` and store the result in `c`. The vector `(b, blen)` is either truncated or zero-extended to the number of columns of `A`. The number entries written to `c` is always equal to the number of rows of `A`.

void `fmprq_mat_fmpq_vec_mul`(*fmpq* *c, const *fmpq* *a, *slong* alen, const *fmprq_mat_t* B)

void `fmprq_mat_fmpz_vec_mul`(*fmpq* *c, const *fmpz* *a, *slong* alen, const *fmprq_mat_t* B)

void `fmprq_mat_fmpq_vec_mul_ptr`(*fmpq* *const *c, const *fmpq* *const *a, *slong* alen, const *fmprq_mat_t* B)

void `fmprq_mat_fmpz_vec_mul_ptr`(*fmpq* *const *c, const *fmpz* *const *a, *slong* alen, const *fmprq_mat_t* B)

Compute a vector-matrix product of `(a, alen)` and `B` and store the result in `c`. The vector `(a, alen)` is either truncated or zero-extended to the number of rows of `B`. The number entries written to `c` is always equal to the number of columns of `B`.

5.3.15 Kronecker product

void **fmprq_mat_kronecker_product**(*fmprq_mat_t* C, const *fmprq_mat_t* A, const *fmprq_mat_t* B)
Sets C to the Kronecker product of A and B.

5.3.16 Trace

void **fmprq_mat_trace**(*fmprq_t* trace, const *fmprq_mat_t* mat)
Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

5.3.17 Determinant

void **fmprq_mat_det**(*fmprq_t* det, const *fmprq_mat_t* mat)
Sets *det* to the determinant of *mat*. In the general case, the determinant is computed by clearing denominators and computing a determinant over the integers. Matrices of size 0, 1 or 2 are handled directly.

5.3.18 Permanent

int **fmprq_mat_permanent**(*fmprq_t* res, const *fmprq_mat_t* A)
Sets *res* to the permanent of the square matrix *A*, returning 1 on success. If the matrix is too large, returns 0.

5.3.19 Nonsingular solving

int **fmprq_mat_solve_fraction_free**(*fmprq_mat_t* X, const *fmprq_mat_t* A, const *fmprq_mat_t* B)
int **fmprq_mat_solve_dixon**(*fmprq_mat_t* X, const *fmprq_mat_t* A, const *fmprq_mat_t* B)
int **fmprq_mat_solve_multi_mod**(*fmprq_mat_t* X, const *fmprq_mat_t* A, const *fmprq_mat_t* B)
int **fmprq_mat_solve**(*fmprq_mat_t* X, const *fmprq_mat_t* A, const *fmprq_mat_t* B)
Solves $AX = B$ for nonsingular *A*. Returns nonzero if *A* is nonsingular or if the right hand side is empty, and zero otherwise.

All algorithms clear denominators to obtain a rescaled system over the integers. The *fraction_free* algorithm uses FFLU solving over the integers. The *dixon* and *multi_mod* algorithms use Dixon p-adic lifting or multimodular solving, followed by rational reconstruction with an adaptive stopping test. The *dixon* and *multi_mod* algorithms are generally the best choice for large systems.

The default method chooses an algorithm automatically.

int **fmprq_mat_solve_fmpz_mat_fraction_free**(*fmprq_mat_t* X, const *fmpz_mat_t* A, const *fmpz_mat_t* B)
int **fmprq_mat_solve_fmpz_mat_dixon**(*fmprq_mat_t* X, const *fmpz_mat_t* A, const *fmpz_mat_t* B)
int **fmprq_mat_solve_fmpz_mat_multi_mod**(*fmprq_mat_t* X, const *fmpz_mat_t* A, const *fmpz_mat_t* B)
int **fmprq_mat_solve_fmpz_mat**(*fmprq_mat_t* X, const *fmpz_mat_t* A, const *fmpz_mat_t* B)
Solves $AX = B$ for nonsingular *A*, where *A* and *B* are integer matrices. Returns nonzero if *A* is nonsingular or if the right hand side is empty, and zero otherwise.
int **fmprq_mat_can_solve_multi_mod**(*fmprq_mat_t* X, const *fmprq_mat_t* A, const *fmprq_mat_t* B)
Returns 1 if $AX = B$ has a solution and if so, sets *X* to one such solution. The matrices can have any shape but must have the same number of rows.

```
int fmpq_mat_can_solve_fraction_free(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets X to one such solution. The matrices can have any shape but must have the same number of rows.

```
int fmpq_mat_can_solve_fmpz_mat_dixon(fmpq_mat_t X, const fmpz_mat_t A, const fmpz_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets X to one such solution. The matrices can have any shape but must have the same number of rows. The input matrices must have integer entries and A cannot be an empty matrix.

```
int fmpq_mat_can_solve_dixon(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets X to one such solution. The matrices can have any shape but must have the same number of rows.

```
int fmpq_mat_can_solve(fmpq_mat_t X, const fmpq_mat_t A, const fmpq_mat_t B)
```

Returns 1 if $AX = B$ has a solution and if so, sets X to one such solution. The matrices can have any shape but must have the same number of rows.

5.3.20 Inverse

```
int fmpq_mat_inv(fmpq_mat_t B, const fmpq_mat_t A)
```

Sets B to the inverse matrix of A and returns nonzero. Returns zero if A is singular. A must be a square matrix.

5.3.21 Echelon form

```
int fmpq_mat_pivot(slong *perm, fmpq_mat_t mat, slong r, slong c)
```

Helper function for row reduction. Returns 1 if the entry of mat at row r and column c is nonzero. Otherwise searches for a nonzero entry in the same column among rows $r+1, r+2, \dots$. If a nonzero entry is found at row s , swaps rows r and s and the corresponding entries in $perm$ (unless `NULL`) and returns -1. If no nonzero pivot entry is found, leaves the inputs unchanged and returns 0.

```
slong fmpq_mat_rref_classical(fmpq_mat_t B, const fmpq_mat_t A)
```

Sets B to the reduced row echelon form of A and returns the rank. Performs Gauss-Jordan elimination directly over the rational numbers. This algorithm is usually inefficient and is mainly intended to be used for testing purposes.

```
slong fmpq_mat_rref_fraction_free(fmpq_mat_t B, const fmpq_mat_t A)
```

Sets B to the reduced row echelon form of A and returns the rank. Clears denominators and performs fraction-free Gauss-Jordan elimination using `fmpz_mat` functions.

```
slong fmpq_mat_rref(fmpq_mat_t B, const fmpq_mat_t A)
```

Sets B to the reduced row echelon form of A and returns the rank. This function automatically chooses between the classical and fraction-free algorithms depending on the size of the matrix.

5.3.22 Gram-Schmidt Orthogonalisation

void **fmprq_mat_gso**(*fmprq_mat_t* B, const *fmprq_mat_t* A)

Takes a subset of $\mathbb{Q}^m$ $S = \{a_1, a_2, \dots, a_n\}$ (as the columns of a $m \times n$ matrix A) and generates an orthogonal set $S' = \{b_1, b_2, \dots, b_n\}$ (as the columns of the $m \times n$ matrix B) that spans the same subspace of $\mathbb{Q}^m$ as S .

5.3.23 Transforms

void **fmprq_mat_similarity**(*fmprq_mat_t* A, *slong* r, *fmprq_t* d)

Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

5.3.24 Characteristic polynomial

void **_fmprq_mat_charpoly**(*fmprz_t* coeffs, *fmprz_t* den, const *fmprq_mat_t* mat)

Set (coeffs, den) to the characteristic polynomial of the given $n \times n$ matrix.

void **fmprq_mat_charpoly**(*fmprq_poly_t* pol, const *fmprq_mat_t* mat)

Set pol to the characteristic polynomial of the given $n \times n$ matrix. If mat is not square, an exception is raised.

5.3.25 Minimal polynomial

slong **_fmprq_mat_minpoly**(*fmprz_t* coeffs, *fmprz_t* den, const *fmprq_mat_t* mat)

Set (coeffs, den) to the minimal polynomial of the given $n \times n$ matrix and return the length of the polynomial.

void **fmprq_mat_minpoly**(*fmprq_poly_t* pol, const *fmprq_mat_t* mat)

Set pol to the minimal polynomial of the given $n \times n$ matrix. If mat is not square, an exception is raised.

5.4 fmprq_poly.h – univariate polynomials over the rational numbers

The *fmprq_poly_t* data type represents elements of $\mathbb{Q}[x]$. The **fmprq_poly** module provides routines for memory management, basic arithmetic, and conversions from or to other types.

A rational polynomial is stored as the quotient of an integer polynomial and an integer denominator. To be more precise, the coefficient vector of the numerator can be accessed with the function **fmprq_poly_numref()** and the denominator with **fmprq_poly_denref()**. Although one can construct use cases in which a representation as a list of rational coefficients would be beneficial, the choice made here is typically more efficient.

We can obtain a unique representation based on this choice by enforcing, for non-zero polynomials, that the numerator and denominator are coprime and that the denominator is positive. The unique representation of the zero polynomial is chosen as $0/1$.

Similar to the situation in the *fmprz_poly_t* case, an *fmprq_poly_t* object also has a **length** parameter, which denotes the length of the vector of coefficients of the numerator. We say a polynomial is *normalised* either if this length is zero or if the leading coefficient is non-zero.

We say a polynomial is in *canonical* form if it is given in the unique representation discussed above and normalised.

The functions provided in this module roughly fall into two categories:

On the one hand, there are functions mainly provided for the user, whose names do not begin with an underscore. These typically operate on polynomials of type `fmpq_poly_t` in canonical form and, unless specified otherwise, permit aliasing between their input arguments and between their output arguments.

On the other hand, there are versions of these functions whose names are prefixed with a single underscore. These typically operate on polynomials given in the form of a triple of object of types `fmpz *`, `fmpz_t`, and `slong`, containing the numerator, denominator and length, respectively. In general, these functions expect their input to be normalised, i.e. they do not allow zero padding, and to be in lowest terms, and they do not allow their input and output arguments to be aliased.

5.4.1 Types, macros and constants

type `fmpq_poly_struct`

type `fmpq_poly_t`

5.4.2 Memory management

void `fmpq_poly_init(fmpq_poly_t poly)`

Initialises the polynomial for use. The length is set to zero.

void `fmpq_poly_init2(fmpq_poly_t poly, slong alloc)`

Initialises the polynomial with space for at least `alloc` coefficients and sets the length to zero. The `alloc` coefficients are all set to zero.

void `fmpq_poly_realloc(fmpq_poly_t poly, slong alloc)`

Reallocates the given polynomial to have space for `alloc` coefficients. If `alloc` is zero then the polynomial is cleared and then reinitialised. If the current length is greater than `alloc` then `poly` is first truncated to length `alloc`. Note that this might leave the rational polynomial in non-canonical form.

void `fmpq_poly_fit_length(fmpq_poly_t poly, slong len)`

If `len` is greater than the number of coefficients currently allocated, then the polynomial is re-allocated to have space for at least `len` coefficients. No data is lost when calling this function. The function efficiently deals with the case where `fit_length()` is called many times in small increments by at least doubling the number of allocated coefficients when `len` is larger than the number of coefficients currently allocated.

void `_fmpq_poly_set_length(fmpq_poly_t poly, slong len)`

Sets the length of the numerator polynomial to `len`, demoting coefficients beyond the new length. Note that this method does not guarantee that the rational polynomial is in canonical form.

void `fmpq_poly_clear(fmpq_poly_t poly)`

Clears the given polynomial, releasing any memory used. The polynomial must be reinitialised in order to be used again.

void `_fmpq_poly_normalise(fmpq_poly_t poly)`

Sets the length of `poly` so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. Note that this function does not guarantee the coprimality of the numerator polynomial and the integer denominator.

void **_fmpz_poly_canonicalise**(*fmpz* *poly, *fmpz_t* den, *slong* len)

Puts (poly, den) of length len into canonical form.

It is assumed that the array poly contains a non-zero entry in position len - 1 whenever len > 0. Assumes that den is non-zero.

void **fmpz_poly_canonicalise**(*fmpz_poly_t* poly)

Puts the polynomial poly into canonical form. Firstly, the length is set to the actual length of the numerator polynomial. For non-zero polynomials, it is then ensured that the numerator and denominator are coprime and that the denominator is positive. The canonical form of the zero polynomial is a zero numerator polynomial and a one denominator.

int **_fmpz_poly_is_canonical**(const *fmpz* *poly, const *fmpz_t* den, *slong* len)

Returns whether the polynomial is in canonical form.

int **fmpz_poly_is_canonical**(const *fmpz_poly_t* poly)

Returns whether the polynomial is in canonical form.

5.4.3 Polynomial parameters

slong **fmpz_poly_degree**(const *fmpz_poly_t* poly)

Returns the degree of poly, which is one less than its length, as a *slong*.

slong **fmpz_poly_length**(const *fmpz_poly_t* poly)

Returns the length of poly.

5.4.4 Accessing the numerator and denominator

fmpz ***fmpz_poly_numref**(*fmpz_poly_t* poly)

Returns a reference to the numerator polynomial as an array.

Note that, because of a delayed initialisation approach, this might be NULL for zero polynomials. This situation can be salvaged by calling either *fmpz_poly_fit_length()* or *fmpz_poly_realloc()*.

This function is implemented as a macro returning (poly)->coeffs.

fmpz_t **fmpz_poly_denref**(*fmpz_poly_t* poly)

Returns a reference to the denominator as a *fmpz_t*. The integer is guaranteed to be properly initialised.

This function is implemented as a macro returning (poly)->den.

void **fmpz_poly_get_numerator**(*fmpz_poly_t* res, const *fmpz_poly_t* poly)

Sets res to the numerator of poly, e.g. the primitive part as an *fmpz_poly_t* if it is in canonical form.

void **fmpz_poly_get_denominator**(*fmpz_t* den, const *fmpz_poly_t* poly)

Sets res to the denominator of poly.

5.4.5 Random testing

The functions `fmpq_poly_randtest_foo()` provide random polynomials suitable for testing. On an integer level, this means that long strings of zeros and ones in the binary representation are favoured as well as the special absolute values 0, 1, `COEFF_MAX`, and `WORD_MAX`. On a polynomial level, the integer numerator has a reasonable chance to have a non-trivial content.

void `fmpq_poly_randtest`(*fmpq_poly_t* f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets *f* to a random polynomial with coefficients up to the given length and where each coefficient has up to the given number of bits. The coefficients are signed randomly. One must call `flint_rand_init()` before calling this function.

void `fmpq_poly_randtest_unsigned`(*fmpq_poly_t* f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

Sets *f* to a random polynomial with coefficients up to the given length and where each coefficient has up to the given number of bits. One must call `flint_rand_init()` before calling this function.

void `fmpq_poly_randtest_not_zero`(*fmpq_poly_t* f, *flint_rand_t* state, *slong* len, *flint_bitcnt_t* bits)

As for `fmpq_poly_randtest()` except that *len* and *bits* may not be zero and the polynomial generated is guaranteed not to be the zero polynomial. One must call `flint_rand_init()` before calling this function.

5.4.6 Assignment, swap, negation

void `fmpq_poly_set`(*fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Sets *poly1* to equal *poly2*.

void `fmpq_poly_set_si`(*fmpq_poly_t* poly, *slong* x)

Sets *poly* to the integer *x*.

void `fmpq_poly_set_ui`(*fmpq_poly_t* poly, *ulong* x)

Sets *poly* to the integer *x*.

void `fmpq_poly_set_fmpz`(*fmpq_poly_t* poly, const *fmpz_t* x)

Sets *poly* to the integer *x*.

void `fmpq_poly_set_fmpq`(*fmpq_poly_t* poly, const *fmpq_t* x)

Sets *poly* to the rational *x*, which is assumed to be given in lowest terms.

void `fmpq_poly_set_fmpz_poly`(*fmpq_poly_t* rop, const *fmpz_poly_t* op)

Sets the rational polynomial *rop* to the same value as the integer polynomial *op*.

void `fmpq_poly_set_nmod_poly`(*fmpq_poly_t* rop, const *nmod_poly_t* op)

Sets the coefficients of *rop* to the residues in *op*, normalised to the interval $-m/2 \leq r < m/2$ where *m* is the modulus.

void `fmpq_poly_get_nmod_poly`(*nmod_poly_t* rop, const *fmpq_poly_t* op)

Sets the coefficients of *rop* to the coefficients in the denominator of *op*, reduced by the modulus of *rop*. The result is multiplied by the inverse of the denominator of *op*. It is assumed that the reduction of the denominator of *op* is invertible.

void `fmpq_poly_get_nmod_poly_den`(*nmod_poly_t* rop, const *fmpq_poly_t* op, int den)

Sets the coefficients of *rop* to the coefficients in the denominator of *op*, reduced by the modulus of *rop*. If *den* == 1, the result is multiplied by the inverse of the denominator of *op*. In this case it is assumed that the reduction of the denominator of *op* is invertible.

int `_fmpq_poly_set_str`(*fmpz_t* *poly, *fmpz_t* den, const char *str, *slong* len)

Sets (*poly*, *den*) to the polynomial specified by the null-terminated string *str* of *len* coefficients. The input format is a sequence of coefficients separated by one space.

The result is only guaranteed to be in lowest terms if all coefficients in the input string are in lowest terms.

Returns 0 if no error occurred. Otherwise, returns -1 in which case the resulting value of (`poly`, `den`) is undefined. If `str` is not null-terminated, calling this method might result in a segmentation fault.

int `fmpq_poly_set_str`(*fmpq_poly_t* poly, const char *str)

Sets `poly` to the polynomial specified by the null-terminated string `str`. The input format is the same as the output format of `fmpq_poly_get_str`: the length given as a decimal integer, then two spaces, then the list of coefficients separated by one space.

The result is only guaranteed to be in canonical form if all coefficients in the input string are in lowest terms.

Returns 0 if no error occurred. Otherwise, returns -1 in which case the resulting value of `poly` is set to zero. If `str` is not null-terminated, calling this method might result in a segmentation fault.

char *`fmpq_poly_get_str`(const *fmpq_poly_t* poly)

Returns the string representation of `poly`.

char *`fmpq_poly_get_str_pretty`(const *fmpq_poly_t* poly, const char *var)

Returns the pretty representation of `poly`, using the null-terminated string `var` not equal to `"\0"` as the variable name.

void `fmpq_poly_zero`(*fmpq_poly_t* poly)

Sets `poly` to zero.

void `fmpq_poly_one`(*fmpq_poly_t* poly)

Sets `poly` to the constant polynomial 1.

void `fmpq_poly_neg`(*fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Sets `poly1` to the additive inverse of `poly2`.

void `fmpq_poly_inv`(*fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Sets `poly1` to the multiplicative inverse of `poly2` if possible. Otherwise, if `poly2` is not a unit, leaves `poly1` unmodified and calls `abort()`.

void `fmpq_poly_swap`(*fmpq_poly_t* poly1, *fmpq_poly_t* poly2)

Efficiently swaps the polynomials `poly1` and `poly2`.

void `fmpq_poly_truncate`(*fmpq_poly_t* poly, *slong* n)

If the current length of `poly` is greater than `n`, it is truncated to the given length. Discarded coefficients are demoted, but they are not necessarily set to zero.

void `fmpq_poly_set_trunc`(*fmpq_poly_t* res, const *fmpq_poly_t* poly, *slong* n)

Sets `res` to a copy of `poly`, truncated to length `n`.

void `fmpq_poly_get_slice`(*fmpq_poly_t* rop, const *fmpq_poly_t* op, *slong* i, *slong* j)

Returns the slice with coefficients from x^i (including) to x^j (excluding).

void `fmpq_poly_reverse`(*fmpq_poly_t* res, const *fmpq_poly_t* poly, *slong* n)

This function considers the polynomial `poly` to be of length `n`, notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result `res` may be of length less than `n`.

5.4.7 Getting and setting coefficients

void **fmpq_poly_get_coeff_fmpz**(*fmpz_t* x, const *fmpq_poly_t* poly, *slong* n)

Retrieves the *n*th coefficient of the numerator of *poly*.

void **fmpq_poly_get_coeff_fmpq**(*fmpq_t* x, const *fmpq_poly_t* poly, *slong* n)

Retrieves the *n*th coefficient of *poly*, in lowest terms.

void **fmpq_poly_set_coeff_si**(*fmpq_poly_t* poly, *slong* n, *slong* x)

Sets the *n*th coefficient in *poly* to the integer *x*.

void **fmpq_poly_set_coeff_ui**(*fmpq_poly_t* poly, *slong* n, *ulong* x)

Sets the *n*th coefficient in *poly* to the integer *x*.

void **fmpq_poly_set_coeff_fmpz**(*fmpq_poly_t* poly, *slong* n, const *fmpz_t* x)

Sets the *n*th coefficient in *poly* to the integer *x*.

void **fmpq_poly_set_coeff_fmpq**(*fmpq_poly_t* poly, *slong* n, const *fmpq_t* x)

Sets the *n*th coefficient in *poly* to the rational *x*.

5.4.8 Comparison

int **fmpq_poly_equal**(const *fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Returns 1 if *poly1* is equal to *poly2*, otherwise returns 0.

int **_fmpq_poly_equal_trunc**(const *fmpz_t* *poly1, const *fmpz_t* den1, *slong* len1, const *fmpz_t* *poly2, const *fmpz_t* den2, *slong* len2, *slong* n)

Returns 1 if *poly1* and *poly2* notionally truncated to length *n* are equal, otherwise returns 0.

int **fmpq_poly_equal_trunc**(const *fmpq_poly_t* poly1, const *fmpq_poly_t* poly2, *slong* n)

Returns 1 if *poly1* and *poly2* notionally truncated to length *n* are equal, otherwise returns 0.

int **_fmpq_poly_cmp**(const *fmpz_t* *lpoly, const *fmpz_t* lden, const *fmpz_t* *rpoly, const *fmpz_t* rden, *slong* len)

Compares two non-zero polynomials, assuming they have the same length *len* > 0.

The polynomials are expected to be provided in canonical form.

int **fmpq_poly_cmp**(const *fmpq_poly_t* left, const *fmpq_poly_t* right)

Compares the two polynomials *left* and *right*.

Compares the two polynomials *left* and *right*, returning -1, 0, or 1 as *left* is less than, equal to, or greater than *right*. The comparison is first done by the degree, and then, in case of a tie, by the individual coefficients from highest to lowest.

int **fmpq_poly_is_one**(const *fmpq_poly_t* poly)

Returns 1 if *poly* is the constant polynomial 1, otherwise returns 0.

int **fmpq_poly_is_zero**(const *fmpq_poly_t* poly)

Returns 1 if *poly* is the zero polynomial, otherwise returns 0.

int **fmpq_poly_is_gen**(const *fmpq_poly_t* poly)

Returns 1 if *poly* is the degree 1 polynomial *x*, otherwise returns 0.

5.4.9 Addition and subtraction

```
void _fmpq_poly_add(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1,
                  const fmpz *poly2, const fmpz_t den2, slong len2)
```

Forms the sum (rpoly, rden) of (poly1, den1, len1) and (poly2, den2, len2), placing the result into canonical form.

Assumes that rpoly is an array of length the maximum of len1 and len2. The input operands are assumed to be in canonical form and are also allowed to be of length 0.

(rpoly, rden) and (poly1, den1) may be aliased, but (rpoly, rden) and (poly2, den2) may *not* be aliased.

```
void _fmpq_poly_add_can(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong
                      len1, const fmpz *poly2, const fmpz_t den2, slong len2, int can)
```

As per _fmpq_poly_add except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void fmpq_poly_add(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
```

Sets res to the sum of poly1 and poly2, using Henriici's algorithm.

```
void fmpq_poly_add_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, int
                      can)
```

As per mpq_poly_add except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void _fmpq_poly_add_series(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong
                          len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)
```

As per _fmpq_poly_add but the inputs are first notionally truncated to length n . If n is less than len1 or len2 then the output only needs space for n coefficients. We require $n \geq 0$.

```
void _fmpq_poly_add_series_can(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1,
                              slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong
                              n, int can)
```

As per _fmpq_poly_add_can but the inputs are first notionally truncated to length n . If n is less than len1 or len2 then the output only needs space for n coefficients. We require $n \geq 0$.

```
void fmpq_poly_add_series(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2,
                          slong n)
```

As per fmpq_poly_add but the inputs are first notionally truncated to length n .

```
void fmpq_poly_add_series_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t
                             poly2, slong n, int can)
```

As per fmpq_poly_add_can but the inputs are first notionally truncated to length n .

```
void _fmpq_poly_sub(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong len1,
                  const fmpz *poly2, const fmpz_t den2, slong len2)
```

Forms the difference (rpoly, rden) of (poly1, den1, len1) and (poly2, den2, len2), placing the result into canonical form.

Assumes that rpoly is an array of length the maximum of len1 and len2. The input operands are assumed to be in canonical form and are also allowed to be of length 0.

(rpoly, rden) and (poly1, den1, len1) may be aliased, but (rpoly, rden) and (poly2, den2, len2) may *not* be aliased.

```
void _fmpq_poly_sub_can(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong
                      len1, const fmpz *poly2, const fmpz_t den2, slong len2, int can)
```

As per _fmpq_poly_sub except that one can specify whether to canonicalise the output or not.

This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void fmpq_poly_sub(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
```

Sets `res` to the difference of `poly1` and `poly2`, using Henrici's algorithm.

```
void fmpq_poly_sub_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2, int
                      can)
```

As per `_fmpq_poly_sub` except that one can specify whether to canonicalise the output or not. This function is intended to be used with weak canonicalisation to prevent explosion in memory usage. It exists for performance reasons.

```
void _fmpq_poly_sub_series(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong
                          len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong n)
```

As per `_fmpq_poly_sub` but the inputs are first notionally truncated to length n . If n is less than `len1` or `len2` then the output only needs space for n coefficients. We require $n \geq 0$.

```
void _fmpq_poly_sub_series_can(fmpz *rpoly, fmpz_t rden, const fmpz *poly1, const fmpz_t den1,
                              slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong
                              n, int can)
```

As per `_fmpq_poly_sub_can` but the inputs are first notionally truncated to length n . If n is less than `len1` or `len2` then the output only needs space for n coefficients. We require $n \geq 0$.

```
void fmpq_poly_sub_series(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2,
                          slong n)
```

As per `fmpq_poly_sub` but the inputs are first notionally truncated to length n .

```
void fmpq_poly_sub_series_can(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t
                             poly2, slong n, int can)
```

As per `fmpq_poly_sub_can` but the inputs are first notionally truncated to length n .

5.4.10 Scalar multiplication and division

```
void _fmpq_poly_scalar_mul_si(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                              slong len, slong c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, len)`.

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_ui(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                              slong len, ulong c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, len)`.

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between `(rpoly, den)` and `(poly, den)`.

```
void _fmpq_poly_scalar_mul_fmpz(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                                slong len, const fmpz_t c)
```

Sets `(rpoly, rden, len)` to the product of c of `(poly, den, len)`.

If the input is normalised, then so is the output, provided it is non-zero. If the input is in lowest terms, then so is the output. However, even if neither of these conditions are met, the result will be (mathematically) correct.

Supports exact aliasing between `(rpoly, den)` and `(poly, den)`.

```
void _fmpz_poly_scalar_mul_fmpz(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                               slong len, const fmpz_t r, const fmpz_t s)
```

Sets (rpoly, rden) to the product of r/s and (poly, den, len), in lowest terms.

Assumes that (poly, den, len) and r/s are provided in lowest terms. Assumes that rpoly is an array of length len. Supports aliasing of (rpoly, den) and (poly, den). The fmpz_t's r and s may not be part of (rpoly, rden).

```
void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t rop, const fmpz_poly_t op, const fmpz_t c)
```

```
void fmpz_poly_scalar_mul_si(fmpz_poly_t rop, const fmpz_poly_t op, slong c)
```

```
void fmpz_poly_scalar_mul_ui(fmpz_poly_t rop, const fmpz_poly_t op, ulong c)
```

Sets rop to c times op.

```
void fmpz_poly_scalar_mul_fmpz(fmpz_poly_t rop, const fmpz_poly_t op, const fmpz_t c)
```

Sets rop to c times op. Assumes that the fmpz_t c is not part of rop.

```
void _fmpz_poly_scalar_div_fmpz(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                               slong len, const fmpz_t c)
```

Sets (rpoly, rden, len) to (poly, den, len) divided by c , in lowest terms.

Assumes that len is positive. Assumes that c is non-zero. Supports aliasing between (rpoly, rden) and (poly, den). Assumes that c is not part of (rpoly, rden).

```
void _fmpz_poly_scalar_div_si(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                             slong len, slong c)
```

Sets (rpoly, rden, len) to (poly, den, len) divided by c , in lowest terms.

Assumes that len is positive. Assumes that c is non-zero. Supports aliasing between (rpoly, rden) and (poly, den).

```
void _fmpz_poly_scalar_div_ui(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                             slong len, ulong c)
```

Sets (rpoly, rden, len) to (poly, den, len) divided by c , in lowest terms.

Assumes that len is positive. Assumes that c is non-zero. Supports aliasing between (rpoly, rden) and (poly, den).

```
void _fmpz_poly_scalar_div_fmpz(fmpz *rpoly, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                               slong len, const fmpz_t r, const fmpz_t s)
```

Sets (rpoly, rden, len) to (poly, den, len) divided by r/s , in lowest terms.

Assumes that len is positive. Assumes that r/s is non-zero and in lowest terms. Supports aliasing between (rpoly, rden) and (poly, den). The fmpz_t's r and s may not be part of (rpoly, poly).

```
void fmpz_poly_scalar_div_si(fmpz_poly_t rop, const fmpz_poly_t op, slong c)
```

```
void fmpz_poly_scalar_div_ui(fmpz_poly_t rop, const fmpz_poly_t op, ulong c)
```

```
void fmpz_poly_scalar_div_fmpz(fmpz_poly_t rop, const fmpz_poly_t op, const fmpz_t c)
```

```
void fmpz_poly_scalar_div_fmpz(fmpz_poly_t rop, const fmpz_poly_t op, const fmpz_t c)
```

Sets rop to op divided by the scalar c .

5.4.11 Multiplication

void `_fmpz_poly_mul`(*fmpz* *rpoly, *fmpz_t* rden, const *fmpz* *poly1, const *fmpz_t* den1, *slong* len1, const *fmpz* *poly2, const *fmpz_t* den2, *slong* len2)

Sets (rpoly, rden, len1 + len2 - 1) to the product of (poly1, den1, len1) and (poly2, den2, len2). If the input is provided in canonical form, then so is the output.

Assumes len1 >= len2 > 0. Allows zero-padding in the input. Does not allow aliasing between the inputs and outputs.

void `fmpz_poly_mul`(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2)

Sets res to the product of poly1 and poly2.

void `_fmpz_poly_mullo`(*fmpz* *rpoly, *fmpz_t* rden, const *fmpz* *poly1, const *fmpz_t* den1, *slong* len1, const *fmpz* *poly2, const *fmpz_t* den2, *slong* len2, *slong* n)

Sets (rpoly, rden, n) to the low *n* coefficients of (poly1, den1) and (poly2, den2). The output is not guaranteed to be in canonical form.

Assumes len1 >= len2 > 0 and 0 < n <= len1 + len2 - 1. Allows for zero-padding in the inputs. Does not allow aliasing between the inputs and outputs.

void `fmpz_poly_mullo`(*fmpz_poly_t* res, const *fmpz_poly_t* poly1, const *fmpz_poly_t* poly2, *slong* n)

Sets res to the product of poly1 and poly2, truncated to length *n*.

void `fmpz_poly_addmul`(*fmpz_poly_t* rop, const *fmpz_poly_t* op1, const *fmpz_poly_t* op2)

Adds the product of op1 and op2 to rop.

void `fmpz_poly_submul`(*fmpz_poly_t* rop, const *fmpz_poly_t* op1, const *fmpz_poly_t* op2)

Subtracts the product of op1 and op2 from rop.

5.4.12 Powering

void `_fmpz_poly_pow`(*fmpz* *rpoly, *fmpz_t* rden, const *fmpz* *poly, const *fmpz_t* den, *slong* len, *ulong* e)

Sets (rpoly, rden) to (poly, den)^e, assuming e, len > 0. Assumes that rpoly is an array of length at least e * (len - 1) + 1. Supports aliasing of (rpoly, den) and (poly, den).

void `fmpz_poly_pow`(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *ulong* e)

Sets res to poly^e, where the only special case 0⁰ is defined as 1.

void `_fmpz_poly_pow_trunc`(*fmpz* *res, *fmpz_t* rden, const *fmpz* *f, const *fmpz_t* fden, *slong* flen, *ulong* exp, *slong* len)

Sets (rpoly, rden, len) to (poly, den)^e truncated to length len, where len is at most e * (flen - 1) + 1.

void `fmpz_poly_pow_trunc`(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *ulong* e, *slong* n)

Sets res to poly^e truncated to length n.

5.4.13 Shifting

void **fmpq_poly_shift_left**(*fmpq_poly_t* res, const *fmpq_poly_t* poly, *slong* n)

Set *res* to *poly* shifted left by *n* coefficients. Zero coefficients are inserted.

void **fmpq_poly_shift_right**(*fmpq_poly_t* res, const *fmpq_poly_t* poly, *slong* n)

Set *res* to *poly* shifted right by *n* coefficients. If *n* is equal to or greater than the current length of *poly*, *res* is set to the zero polynomial.

5.4.14 Euclidean division

void **_fmpq_poly_divrem**(*fmpz* *Q, *fmpz_t* q, *fmpz* *R, *fmpz_t* r, const *fmpz* *A, const *fmpz_t* a, *slong* lenA, const *fmpz* *B, const *fmpz_t* b, *slong* lenB, const *fmpz_preinvn_t* inv)

Finds the quotient (*Q*, *q*) and remainder (*R*, *r*) of the Euclidean division of (*A*, *a*) by (*B*, *b*).

Assumes that `lenA >= lenB > 0`. Assumes that *R* has space for `lenA` coefficients, although only the bottom `lenB - 1` will carry meaningful data on exit. Supports no aliasing between the two outputs, or between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of *B* from `fmpz_preinvn_init` can be supplied. Otherwise *inv* should be `NULL`.

Note: `fmpz.h` has to be included before `fmpq_poly.h` in order for the latter to declare this function.

void **fmpq_poly_divrem**(*fmpq_poly_t* Q, *fmpq_poly_t* R, const *fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Finds the quotient *Q* and remainder *R* of the Euclidean division of *poly1* by *poly2*.

void **_fmpq_poly_div**(*fmpz* *Q, *fmpz_t* q, const *fmpz* *A, const *fmpz_t* a, *slong* lenA, const *fmpz* *B, const *fmpz_t* b, *slong* lenB, const *fmpz_preinvn_t* inv)

Finds the quotient (*Q*, *q*) of the Euclidean division of (*A*, *a*) by (*B*, *b*).

Assumes that `lenA >= lenB > 0`. Supports no aliasing between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of *B* from `fmpz_preinvn_init` can be supplied. Otherwise *inv* should be `NULL`.

Note: `fmpz.h` has to be included before `fmpq_poly.h` in order for the latter to declare this function.

void **fmpq_poly_div**(*fmpq_poly_t* Q, const *fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Finds the quotient *Q* and remainder *R* of the Euclidean division of *poly1* by *poly2*.

void **_fmpq_poly_rem**(*fmpz* *R, *fmpz_t* r, const *fmpz* *A, const *fmpz_t* a, *slong* lenA, const *fmpz* *B, const *fmpz_t* b, *slong* lenB, const *fmpz_preinvn_t* inv)

Finds the remainder (*R*, *r*) of the Euclidean division of (*A*, *a*) by (*B*, *b*).

Assumes that `lenA >= lenB > 0`. Supports no aliasing between the inputs and the outputs.

An optional precomputed inverse of the leading coefficient of *B* from `fmpz_preinvn_init` can be supplied. Otherwise *inv* should be `NULL`.

Note: `fmpz.h` has to be included before `fmpq_poly.h` in order for the latter to declare this function.

void **fmpq_poly_rem**(*fmpq_poly_t* R, const *fmpq_poly_t* poly1, const *fmpq_poly_t* poly2)

Finds the remainder *R* of the Euclidean division of *poly1* by *poly2*.

5.4.15 Powering

fmprz *poly_struct* **_fmprz_poly_powers_precompute*(const *fmprz* *B, const *fmprz_t* denB, *slong* len)

Computes $2 \cdot \text{len} - 1$ powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

void *fmprz_poly_powers_precompute*(*fmprz_poly_powers_precomp_t* pinv, *fmprz_poly_t* poly)

Computes $2 \cdot \text{len} - 1$ powers of x modulo the polynomial B of the given length. This is used as a kind of precomputed inverse in the remainder routine below.

void *_fmprz_poly_powers_clear*(*fmprz_poly_struct* *powers, *slong* len)

Clean up resources used by precomputed powers which have been computed by *_fmprz_poly_powers_precompute*.

void *fmprz_poly_powers_clear*(*fmprz_poly_powers_precomp_t* pinv)

Clean up resources used by precomputed powers which have been computed by *fmprz_poly_powers_precompute*.

void *_fmprz_poly_rem_powers_precomp*(*fmprz* *A, *fmprz_t* denA, *slong* m, const *fmprz* *B, const *fmprz_t* denB, *slong* n, *fmprz_poly_struct* *const powers)

Set A to the remainder of A divide B given precomputed powers mod B provided by *_fmprz_poly_powers_precompute*. No aliasing is allowed.

This function is only faster if $m \leq 2 \cdot n - 1$.

The output of this function is *not* canonicalised.

void *fmprz_poly_rem_powers_precomp*(*fmprz_poly_t* R, const *fmprz_poly_t* A, const *fmprz_poly_t* B, const *fmprz_poly_powers_precomp_t* B_inv)

Set R to the remainder of A divide B given precomputed powers mod B provided by *fmprz_poly_powers_precompute*.

This function is only faster if $A \rightarrow \text{length} \leq 2 \cdot B \rightarrow \text{length} - 1$.

The output of this function is *not* canonicalised.

5.4.16 Divisibility testing

int *_fmprz_poly_divides*(*fmprz* *qpoly, *fmprz_t* qden, const *fmprz* *poly1, const *fmprz_t* den1, *slong* len1, const *fmprz* *poly2, const *fmprz_t* den2, *slong* len2)

Return 1 if (poly2, den2, len2) divides (poly1, den1, len1) and set (qpoly, qden, len1 - len2 + 1) to the quotient. Otherwise return 0. Requires that qpoly has space for $\text{len1} - \text{len2} + 1$ coefficients and that $\text{len1} \geq \text{len2} > 0$.

int *fmprz_poly_divides*(*fmprz_poly_t* q, const *fmprz_poly_t* poly1, const *fmprz_poly_t* poly2)

Return 1 if poly2 divides poly1 and set q to the quotient. Otherwise return 0.

slong *fmprz_poly_remove*(*fmprz_poly_t* q, const *fmprz_poly_t* poly1, const *fmprz_poly_t* poly2)

Sets q to the quotient of poly1 by the highest power of poly2 which divides it, and returns the power. The divisor poly2 must not be constant or an exception is raised.

5.4.17 Power series division

```
void _fmpz_poly_inv_series_newton(fmpz *rpol, fmpz_t rden, const fmpz *poly, const fmpz_t den,
                                slong len, slong n)
```

Computes the first n terms of the inverse power series of $(poly, den, len)$ using Newton iteration. The result is produced in canonical form.

Assumes that $n \geq 1$ and that $poly$ has non-zero constant term. Does not support aliasing.

```
void fmpz_poly_inv_series_newton(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
```

Computes the first n terms of the inverse power series of $poly$ using Newton iteration, assuming that $poly$ has non-zero constant term and $n \geq 1$.

```
void _fmpz_poly_inv_series(fmpz *rpol, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong
                           den_len, slong n)
```

Computes the first n terms of the inverse power series of $(poly, den, len)$.

The result is produced in canonical form.

Assumes that $n \geq 1$ and that $poly$ has non-zero constant term. Does not support aliasing.

```
void fmpz_poly_inv_series(fmpz_poly_t res, const fmpz_poly_t poly, slong n)
```

Computes the first n terms of the inverse power series of $poly$, assuming that $poly$ has non-zero constant term and $n \geq 1$.

```
void _fmpz_poly_div_series(fmpz *Q, fmpz_t denQ, const fmpz *A, const fmpz_t denA, slong lenA,
                           const fmpz *B, const fmpz_t denB, slong lenB, slong n)
```

Divides $(A, denA, lenA)$ by $(B, denB, lenB)$ as power series over $\mathbb{Q}$, assuming B has non-zero constant term and that all lengths are positive.

Aliasing is not supported.

This function ensures that the numerator and denominator are coprime on exit.

```
void fmpz_poly_div_series(fmpz_poly_t Q, const fmpz_poly_t A, const fmpz_poly_t B, slong n)
```

Performs power series division in $\mathbb{Q}[[x]]/(x^n)$. The function considers the polynomials A and B as power series of length n starting with the constant terms. The function assumes that B has non-zero constant term and $n \geq 1$.

5.4.18 Greatest common divisor

```
void _fmpz_poly_gcd(fmpz *G, fmpz_t denG, const fmpz *A, slong lenA, const fmpz *B, slong lenB)
```

Computes the monic greatest common divisor G of A and B .

Assumes that G has space for $\text{len}(B)$ coefficients, where $\text{len}(A) \geq \text{len}(B) > 0$.

Aliasing between the output and input arguments is not supported.

Does not support zero-padding.

```
void fmpz_poly_gcd(fmpz_poly_t G, const fmpz_poly_t A, const fmpz_poly_t B)
```

Computes the monic greatest common divisor G of A and B .

In the special case when $A = B = 0$, sets $G = 0$.

```
void _fmpz_poly_xgcd(fmpz *G, fmpz_t denG, fmpz *S, fmpz_t denS, fmpz *T, fmpz_t denT, const
                    fmpz *A, const fmpz_t denA, slong lenA, const fmpz *B, const fmpz_t denB,
                    slong lenB)
```

Computes polynomials G , S , and T such that $G = \text{gcd}(A, B) = SA + TB$, where G is the monic greatest common divisor of A and B .

Assumes that G , S , and T have space for $\text{len}(B)$, $\text{len}(B)$, and $\text{len}(A)$ coefficients, respectively, where it is also assumed that $\text{len}(A) \geq \text{len}(B) > 0$.

Does not support zero padding of the input arguments.

```
void fmpq_poly_xgcd(fmpq_poly_t G, fmpq_poly_t S, fmpq_poly_t T, const fmpq_poly_t A, const
                  fmpq_poly_t B)
```

Computes polynomials G , S , and T such that $G = \gcd(A, B) = SA + TB$, where G is the monic greatest common divisor of A and B .

Corner cases are handled as follows. If $A = B = 0$, returns $G = S = T = 0$. If $A \neq 0$, $B = 0$, returns the suitable scalar multiple of $G = A$, $S = 1$, and $T = 0$. The case when $A = 0$, $B \neq 0$ is handled similarly.

```
void _fmpq_poly_lcm(fmpz *L, fmpz_t denL, const fmpz *A, slong lenA, const fmpz *B, slong lenB)
```

Computes the monic least common multiple L of A and B .

Assumes that L has space for $\text{len}(A) + \text{len}(B) - 1$ coefficients, where $\text{len}(A) \geq \text{len}(B) > 0$.

Aliasing between the output and input arguments is not supported.

Does not support zero-padding.

```
void fmpq_poly_lcm(fmpq_poly_t L, const fmpq_poly_t A, const fmpq_poly_t B)
```

Computes the monic least common multiple L of A and B .

In the special case when $A = B = 0$, sets $L = 0$.

```
void _fmpq_poly_resultant(fmpz_t rnum, fmpz_t rden, const fmpz *poly1, const fmpz_t den1, slong
                        len1, const fmpz *poly2, const fmpz_t den2, slong len2)
```

Sets $(\text{rnum}, \text{rden})$ to the resultant of the two input polynomials.

Assumes that $\text{len1} \geq \text{len2} > 0$. Does not support zero-padding of the input polynomials. Does not support aliasing of the input and output arguments.

```
void fmpq_poly_resultant(fmpq_t r, const fmpq_poly_t f, const fmpq_poly_t g)
```

Returns the resultant of f and g .

Enumerating the roots of f and g over $\bar{\mathbf{Q}}$ as $r_1, \dots, r_m$ and $s_1, \dots, s_n$, respectively, and letting x and y denote the leading coefficients, the resultant is defined as

$$x^{\deg(f)} y^{\deg(g)} \prod_{1 \leq i, j \leq n} (r_i - s_j).$$

We handle special cases as follows: if one of the polynomials is zero, the resultant is zero. Note that otherwise if one of the polynomials is constant, the last term in the above expression is the empty product.

```
void fmpq_poly_resultant_div(fmpq_t r, const fmpq_poly_t f, const fmpq_poly_t g, const fmpz_t
                          div, slong nbits)
```

Returns the resultant of f and g divided by div under the assumption that the result has at most nbits bits. The result must be an integer.

5.4.19 Discriminant

```
void _fmpq_poly_discriminant(fmpq_t res, const fmpz *poly, const fmpz_t den, slong len)
```

Set res to the discriminant of $(\text{poly}, \text{den}, \text{len})$. Assumes $\text{len} > 1$.

```
void fmpq_poly_discriminant(fmpq_t res, const fmpq_poly_t poly)
```

Set res to the discriminant of poly . We normalise the discriminant so that $\text{disc}(f) = (-1)^{(n(n-1)/2)} \text{res}(f, f') / \text{lc}(f)$, thus $\text{disc}(f) = \text{lc}(f)^{(2n-2)} \prod_{i < j} (r_i - r_j)^2$, where $\text{lc}(f)$ is the leading coefficient of f , n is the degree of f and r_i are the roots of f .

5.4.20 Derivative and integral

void **_fmpz_poly_derivative**(*fmpz* *rpoly, *fmpz_t* rden, const *fmpz* *poly, const *fmpz_t* den, *slong* len)

Sets (rpoly, rden, len - 1) to the derivative of (poly, den, len). Does nothing if len <= 1. Supports aliasing between the two polynomials.

void **fmpz_poly_derivative**(*fmpz_poly_t* res, const *fmpz_poly_t* poly)

Sets res to the derivative of poly.

void **_fmpz_poly_nth_derivative**(*fmpz* *rpoly, *fmpz_t* rden, const *fmpz* *poly, const *fmpz_t* den, *ulong* n, *slong* len)

Sets (rpoly, rden, len - n) to the nth derivative of (poly, den, len). Does nothing if len <= n. Supports aliasing between the two polynomials.

void **fmpz_poly_nth_derivative**(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *ulong* n)

Sets res to the nth derivative of poly.

void **_fmpz_poly_integral**(*fmpz* *rpoly, *fmpz_t* rden, const *fmpz* *poly, const *fmpz_t* den, *slong* len)

Sets (rpoly, rden, len) to the integral of (poly, den, len - 1). Assumes len >= 0. Supports aliasing between the two polynomials. The output will be in canonical form if the input is in canonical form.

void **fmpz_poly_integral**(*fmpz_poly_t* res, const *fmpz_poly_t* poly)

Sets res to the integral of poly. The constant term is set to zero. In particular, the integral of the zero polynomial is the zero polynomial.

5.4.21 Square roots

void **_fmpz_poly_sqrt_series**(*fmpz* *g, *fmpz_t* gden, const *fmpz* *f, const *fmpz_t* fden, *slong* flen, *slong* n)

Sets (g, gden, n) to the series expansion of the square root of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 1. Does not support aliasing between the input and output polynomials.

void **fmpz_poly_sqrt_series**(*fmpz_poly_t* res, const *fmpz_poly_t* f, *slong* n)

Sets res to the series expansion of the square root of f to order n > 1. Requires f to have constant term 1.

void **_fmpz_poly_invsqrt_series**(*fmpz* *g, *fmpz_t* gden, const *fmpz* *f, const *fmpz_t* fden, *slong* flen, *slong* n)

Sets (g, gden, n) to the series expansion of the inverse square root of (f, fden, flen). Assumes n > 0 and that (f, fden, flen) has constant term 1. Does not support aliasing between the input and output polynomials.

void **fmpz_poly_invsqrt_series**(*fmpz_poly_t* res, const *fmpz_poly_t* f, *slong* n)

Sets res to the series expansion of the inverse square root of f to order n > 0. Requires f to have constant term 1.

5.4.22 Power sums

void `_fmpz_poly_power_sums`(*fmpz* *res, *fmpz_t* rden, const *fmpz* *poly, *slong* len, *slong* n)
 Compute the (truncated) power sums series of the polynomial (poly,len) up to length *n* using Newton identities.

void `fmpz_poly_power_sums`(*fmpz_poly_t* res, const *fmpz_poly_t* poly, *slong* n)
 Compute the (truncated) power sum series of the monic polynomial poly up to length *n* using Newton identities. That is the power series whose coefficient of degree *i* is the sum of the *i*-th power of all (complex) roots of the polynomial poly.

void `_fmpz_poly_power_sums_to_poly`(*fmpz* *res, const *fmpz* *poly, const *fmpz_t* den, *slong* len)
 Compute an integer polynomial given by its power sums series (poly,den,len).

void `fmpz_poly_power_sums_to_fmpz_poly`(*fmpz_poly_t* res, const *fmpz_poly_t* Q)
 Compute the integer polynomial with content one and positive leading coefficient given by its power sums series Q.

void `fmpz_poly_power_sums_to_poly`(*fmpz_poly_t* res, const *fmpz_poly_t* Q)
 Compute the monic polynomial from its power sums series Q.

5.4.23 Transcendental functions

void `_fmpz_poly_log_series`(*fmpz* *g, *fmpz_t* gden, const *fmpz* *f, const *fmpz_t* fden, *slong* flen, *slong* n)
 Sets (g, gden, n) to the series expansion of the logarithm of (f, fden, flen). Assumes *n* > 0 and that (f, fden, flen) has constant term 1. Supports aliasing between the input and output polynomials.

void `fmpz_poly_log_series`(*fmpz_poly_t* res, const *fmpz_poly_t* f, *slong* n)
 Sets res to the series expansion of the logarithm of f to order *n* > 0. Requires f to have constant term 1.

void `_fmpz_poly_exp_series`(*fmpz* *g, *fmpz_t* gden, const *fmpz* *h, const *fmpz_t* hden, *slong* hlen, *slong* n)
 Sets (g, gden, n) to the series expansion of the exponential function of (h, hden, hlen). Assumes *n* > 0, hlen > 0 and that (h, hden, hlen) has constant term 0. Supports aliasing between the input and output polynomials.

void `fmpz_poly_exp_series`(*fmpz_poly_t* res, const *fmpz_poly_t* h, *slong* n)
 Sets res to the series expansion of the exponential function of h to order *n* > 0. Requires f to have constant term 0.

void `_fmpz_poly_exp_expinv_series`(*fmpz* *res1, *fmpz_t* res1den, *fmpz* *res2, *fmpz_t* res2den, const *fmpz* *h, const *fmpz_t* hden, *slong* hlen, *slong* n)
 The same as `fmpz_poly_exp_series`, but simultaneously computes the exponential (in res1, res1den) and its multiplicative inverse (in res2, res2den). Supports aliasing between the input and output polynomials.

void `fmpz_poly_exp_expinv_series`(*fmpz_poly_t* res1, *fmpz_poly_t* res2, const *fmpz_poly_t* h, *slong* n)
 The same as `fmpz_poly_exp_series`, but simultaneously computes the exponential (in res1) and its multiplicative inverse (in res2).

void `_fmpz_poly_atan_series`(*fmpz* *g, *fmpz_t* gden, const *fmpz* *f, const *fmpz_t* fden, *slong* flen, *slong* n)
 Sets (g, gden, n) to the series expansion of the inverse tangent of (f, fden, flen). Assumes *n* > 0 and that (f, fden, flen) has constant term 0. Supports aliasing between the input and output polynomials.

void **fmq_poly_atan_series**(*fmq_poly_t* res, const *fmq_poly_t* f, *slong* n)
Sets **res** to the series expansion of the inverse tangent of **f** to order $n > 0$. Requires **f** to have constant term 0.

void **_fmq_poly_atanh_series**(*fmqz *g*, *fmqz_t* gden, const *fmqz *f*, const *fmqz_t* fden, *slong* flen, *slong* n)
Sets (**g**, **gden**, **n**) to the series expansion of the inverse hyperbolic tangent of (**f**, **fden**, **flen**). Assumes $n > 0$ and that (**f**, **fden**, **flen**) has constant term 0. Supports aliasing between the input and output polynomials.

void **fmq_poly_atanh_series**(*fmq_poly_t* res, const *fmq_poly_t* f, *slong* n)
Sets **res** to the series expansion of the inverse hyperbolic tangent of **f** to order $n > 0$. Requires **f** to have constant term 0.

void **_fmq_poly_asin_series**(*fmqz *g*, *fmqz_t* gden, const *fmqz *f*, const *fmqz_t* fden, *slong* flen, *slong* n)
Sets (**g**, **gden**, **n**) to the series expansion of the inverse sine of (**f**, **fden**, **flen**). Assumes $n > 0$ and that (**f**, **fden**, **flen**) has constant term 0. Supports aliasing between the input and output polynomials.

void **fmq_poly_asin_series**(*fmq_poly_t* res, const *fmq_poly_t* f, *slong* n)
Sets **res** to the series expansion of the inverse sine of **f** to order $n > 0$. Requires **f** to have constant term 0.

void **_fmq_poly_asinh_series**(*fmqz *g*, *fmqz_t* gden, const *fmqz *f*, const *fmqz_t* fden, *slong* flen, *slong* n)
Sets (**g**, **gden**, **n**) to the series expansion of the inverse hyperbolic sine of (**f**, **fden**, **flen**). Assumes $n > 0$ and that (**f**, **fden**, **flen**) has constant term 0. Supports aliasing between the input and output polynomials.

void **fmq_poly_asinh_series**(*fmq_poly_t* res, const *fmq_poly_t* f, *slong* n)
Sets **res** to the series expansion of the inverse hyperbolic sine of **f** to order $n > 0$. Requires **f** to have constant term 0.

void **_fmq_poly_tan_series**(*fmqz *g*, *fmqz_t* gden, const *fmqz *f*, const *fmqz_t* fden, *slong* flen, *slong* n)
Sets (**g**, **gden**, **n**) to the series expansion of the tangent function of (**f**, **fden**, **flen**). Assumes $n > 0$ and that (**f**, **fden**, **flen**) has constant term 0. Does not support aliasing between the input and output polynomials.

void **fmq_poly_tan_series**(*fmq_poly_t* res, const *fmq_poly_t* f, *slong* n)
Sets **res** to the series expansion of the tangent function of **f** to order $n > 0$. Requires **f** to have constant term 0.

void **_fmq_poly_sin_series**(*fmqz *g*, *fmqz_t* gden, const *fmqz *f*, const *fmqz_t* fden, *slong* flen, *slong* n)
Sets (**g**, **gden**, **n**) to the series expansion of the sine of (**f**, **fden**, **flen**). Assumes $n > 0$ and that (**f**, **fden**, **flen**) has constant term 0. Supports aliasing between the input and output polynomials.

void **fmq_poly_sin_series**(*fmq_poly_t* res, const *fmq_poly_t* f, *slong* n)
Sets **res** to the series expansion of the sine of **f** to order $n > 0$. Requires **f** to have constant term 0.

void **_fmq_poly_cos_series**(*fmqz *g*, *fmqz_t* gden, const *fmqz *f*, const *fmqz_t* fden, *slong* flen, *slong* n)
Sets (**g**, **gden**, **n**) to the series expansion of the cosine of (**f**, **fden**, **flen**). Assumes $n > 0$ and that (**f**, **fden**, **flen**) has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_cos_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the cosine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_sin_cos_series(fmpz *s, fmpz_t sden, fmpz *c, fmpz_t cden, const fmpz *f, const
                             fmpz_t fden, slong flen, slong n)
```

Sets `(s, sden, n)` to the series expansion of the sine of `(f, fden, flen)`, and `(c, cden, n)` to the series expansion of the cosine. Assumes `n > 0` and that `(f, fden, flen)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_sin_cos_series(fmpq_poly_t res1, fmpq_poly_t res2, const fmpq_poly_t f, slong n)
```

Sets `res1` to the series expansion of the sine of `f` to order `n > 0`, and `res2` to the series expansion of the cosine. Requires `f` to have constant term 0.

```
void _fmpq_poly_sinh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen,
                           slong n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic sine of `(f, fden, flen)`. Assumes `n > 0` and that `(f, fden, flen)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_sinh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the hyperbolic sine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_cosh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen,
                           slong n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic cosine of `(f, fden, flen)`. Assumes `n > 0` and that `(f, fden, flen)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_cosh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the hyperbolic cosine of `f` to order `n > 0`. Requires `f` to have constant term 0.

```
void _fmpq_poly_sinh_cosh_series(fmpz *s, fmpz_t sden, fmpz *c, fmpz_t cden, const fmpz *f,
                                const fmpz_t fden, slong flen, slong n)
```

Sets `(s, sden, n)` to the series expansion of the hyperbolic sine of `(f, fden, flen)`, and `(c, cden, n)` to the series expansion of the hyperbolic cosine. Assumes `n > 0` and that `(f, fden, flen)` has constant term 0. Supports aliasing between the input and output polynomials.

```
void fmpq_poly_sinh_cosh_series(fmpq_poly_t res1, fmpq_poly_t res2, const fmpq_poly_t f, slong
                               n)
```

Sets `res1` to the series expansion of the hyperbolic sine of `f` to order `n > 0`, and `res2` to the series expansion of the hyperbolic cosine. Requires `f` to have constant term 0.

```
void _fmpq_poly_tanh_series(fmpz *g, fmpz_t gden, const fmpz *f, const fmpz_t fden, slong flen,
                           slong n)
```

Sets `(g, gden, n)` to the series expansion of the hyperbolic tangent of `(f, fden, flen)`. Assumes `n > 0` and that `(f, fden, flen)` has constant term 0. Does not support aliasing between the input and output polynomials.

```
void fmpq_poly_tanh_series(fmpq_poly_t res, const fmpq_poly_t f, slong n)
```

Sets `res` to the series expansion of the hyperbolic tangent of `f` to order `n > 0`. Requires `f` to have constant term 0.

5.4.24 Orthogonal polynomials

void **_fmpz_poly_legendre_p**(fmpz *coeffs, fmpz_t den, ulong n)

Sets **coeffs** to the coefficient array of the Legendre polynomial $P_n(x)$, defined by $(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$, for $n \geq 0$. Sets **den** to the overall denominator. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be **n+1**. To improve performance, the common denominator is computed in one step and the coefficients are evaluated using integer arithmetic. The denominator is given by $\gcd(n!, 2^n) = 2^{\lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \dots}$. See **fmpz_poly** for the shifted Legendre polynomials.

void **fmpz_poly_legendre_p**(fmpz_poly_t poly, ulong n)

Sets **poly** to the Legendre polynomial $P_n(x)$, defined by $(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence. To improve performance, the common denominator is computed in one step and the coefficients are evaluated using integer arithmetic. The denominator is given by $\gcd(n!, 2^n) = 2^{\lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \dots}$. See **fmpz_poly** for the shifted Legendre polynomials.

void **_fmpz_poly_laguerre_l**(fmpz *coeffs, fmpz_t den, ulong n)

Sets **coeffs** to the coefficient array of the Laguerre polynomial $L_n(x)$, defined by $(n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x)$, for $n \geq 0$. Sets **den** to the overall denominator. The coefficients are calculated using a hypergeometric recurrence. The length of the array will be **n+1**.

void **fmpz_poly_laguerre_l**(fmpz_poly_t poly, ulong n)

Sets **poly** to the Laguerre polynomial $L_n(x)$, defined by $(n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x)$, for $n \geq 0$. The coefficients are calculated using a hypergeometric recurrence.

void **_fmpz_poly_gegenbauer_c**(fmpz *coeffs, fmpz_t den, ulong n, const fmpz_t a)

Sets **coeffs** to the coefficient array of the Gegenbauer (ultraspherical) polynomial $C_n^{(\alpha)}(x) = \frac{(2\alpha)_n}{n!} {}_2F_1(-n, 2\alpha+n; \alpha+\frac{1}{2}; \frac{1-x}{2})$, for integer $n \geq 0$ and rational $\alpha > 0$. Sets **den** to the overall denominator. The coefficients are calculated using a hypergeometric recurrence.

void **fmpz_poly_gegenbauer_c**(fmpz_poly_t poly, ulong n, const fmpz_t a)

Sets **poly** to the Gegenbauer (ultraspherical) polynomial $C_n^{(\alpha)}(x) = \frac{(2\alpha)_n}{n!} {}_2F_1(-n, 2\alpha+n; \alpha+\frac{1}{2}; \frac{1-x}{2})$, for integer $n \geq 0$ and rational $\alpha > 0$. The coefficients are calculated using a hypergeometric recurrence.

5.4.25 Evaluation

void **_fmpz_poly_evaluate_fmpz**(fmpz_t rnum, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, const fmpz_t a)

Evaluates the polynomial (**poly**, **den**, **len**) at the integer **a** and sets (**rnum**, **rden**) to the result in lowest terms.

void **fmpz_poly_evaluate_fmpz**(fmpz_t res, const fmpz_poly_t poly, const fmpz_t a)

Evaluates the polynomial **poly** at the integer **a** and sets **res** to the result.

void **_fmpz_poly_evaluate_fmpzq**(fmpz_t rnum, fmpz_t rden, const fmpz *poly, const fmpz_t den, slong len, const fmpz_t anum, const fmpz_t aden)

Evaluates the polynomial (**poly**, **den**, **len**) at the rational (**anum**, **aden**) and sets (**rnum**, **rden**) to the result in lowest terms. Aliasing between (**rnum**, **rden**) and (**anum**, **aden**) is not supported.

void **fmpz_poly_evaluate_fmpzq**(fmpz_t res, const fmpz_poly_t poly, const fmpz_t a)

Evaluates the polynomial **poly** at the rational **a** and sets **res** to the result.

5.4.26 Interpolation

```
void _fmpq_poly_interpolation_weights(fmpz *w, fmpz_t wden, const fmpq *xs, slong len)
void _fmpq_poly_interpolate_fast_precomp(fmpz *poly, fmpz_t den, const fmpq *ys, fmpz *const
                                         *tree, const fmpz *weights, slong len)
void _fmpq_poly_interpolate_fast(fmpz *poly, fmpz_t den, const fmpq *xs, const fmpq *ys, slong
                                len)
void fmpq_poly_interpolate_fast(fmpq_poly_t poly, const fmpq *xs, const fmpq *ys, slong n)
    Fast polynomial interpolation using a subproduct tree assuming the evaluation points  $x_i$ 
    are distinct. The precomp version requires a precomputed subproduct tree generated using
    _fmpz_poly_tree_build_fmpq_vec() and precomputed interpolation weights generated using
    _fmpq_poly_interpolation_weights().
void _fmpq_poly_interpolate_barycentric(fmpz *poly, fmpz_t den, const fmpq *xs, const fmpq
                                       *ys, slong n)
void fmpq_poly_interpolate_barycentric(fmpq_poly_t poly, const fmpq *xs, const fmpq *ys, slong
                                       n)
void _fmpq_poly_interpolate_multi_mod(fmpz *poly, fmpz_t den, const fmpq *xs, const fmpq *ys,
                                      slong n)
void fmpq_poly_interpolate_multi_mod(fmpq_poly_t poly, const fmpq *xs, const fmpq *ys, slong n)
int _fmpq_poly_interpolate_fmpq_vec(fmpz *poly, fmpz_t den, const fmpq *xs, const fmpq *ys,
                                    slong n)
int fmpq_poly_interpolate_fmpz_vec(fmpq_poly_t poly, const fmpz *xs, const fmpz *ys, slong n)
int fmpq_poly_interpolate_fmpz_fmpq_vec(fmpq_poly_t poly, const fmpz *xs, const fmpq *ys, slong
                                         n)
int fmpq_poly_interpolate_fmpq_vec(fmpq_poly_t poly, const fmpq *xs, const fmpq *ys, slong n)
    Sets poly (or poly / den) to the unique interpolating polynomial of degree at most  $n - 1$  satisfying
     $f(x_i) = y_i$  for every pair  $x_i, y_i$  in xs and ys.

    The barycentric uses a naive implementation of Lagrange interpolation in its barycentric form,
    clearing denominators to avoid working with fractions. It assumes that the evaluation points  $x_i$ 
    are distinct.

    The multi_mod algorithm performs interpolation modulo one or more prime numbers and com-
    bines the results using the Chinese Remainder Theorem (CRT) with adaptive termination. This
    function is particularly efficient when the interpolated polynomial is expected to have small-bitsize
    coefficients. It assumes that the evaluation points  $x_i$  are distinct.

    The default implementation first checks that the  $x_i$  are distinct and, if so, automatically chooses
    between the barycentric, multi_mod and fast algorithms. It returns 1 if the  $x_i$  are distinct, otherwise
    0.
```

5.4.27 Composition

```
void _fmpq_poly_compose(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1, slong len1,
                       const fmpz *poly2, const fmpz_t den2, slong len2)
    Sets (res, den) to the composition of (poly1, den1, len1) and (poly2, den2, len2), assum-
    ing len1, len2 > 0.

    Assumes that res has space for  $(len1 - 1) * (len2 - 1) + 1$  coefficients. Does not support
    aliasing.
void fmpq_poly_compose(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t poly2)
    Sets res to the composition of poly1 and poly2.
```

```
void _fmpz_poly_rescale(fmpz *res, fmpz_t denr, const fmpz *poly, const fmpz_t den, slong len,
                      const fmpz_t anum, const fmpz_t aden)
```

Sets (res, denr, len) to (poly, den, len) with the indeterminate rescaled by (anum, aden).

Assumes that len > 0 and that (anum, aden) is non-zero and in lowest terms. Supports aliasing between (res, denr, len) and (poly, den, len).

```
void fmpz_poly_rescale(fmpz_poly_t res, const fmpz_poly_t poly, const fmpz_t a)
```

Sets res to poly with the indeterminate rescaled by a.

5.4.28 Power series composition

```
void _fmpz_poly_compose_series_horner(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t
                                     den1, slong len1, const fmpz *poly2, const fmpz_t den2,
                                     slong len2, slong n)
```

Sets (res, den, n) to the composition of (poly1, den1, len1) and (poly2, den2, len2) modulo x^n , where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 ≤ n, that (len1-1) * (len2-1) + 1 ≤ n, and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses the Horner scheme. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.

```
void fmpz_poly_compose_series_horner(fmpz_poly_t res, const fmpz_poly_t poly1, const
                                     fmpz_poly_t poly2, slong n)
```

Sets res to the composition of poly1 and poly2 modulo x^n , where the constant term of poly2 is required to be zero.

This implementation uses the Horner scheme. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.

```
void _fmpz_poly_compose_series_brent_kung(fmpz *res, fmpz_t den, const fmpz *poly1, const
                                          fmpz_t den1, slong len1, const fmpz *poly2, const
                                          fmpz_t den2, slong len2, slong n)
```

Sets (res, den, n) to the composition of (poly1, den1, len1) and (poly2, den2, len2) modulo x^n , where the constant term of poly2 is required to be zero.

Assumes that len1, len2, n > 0, that len1, len2 ≤ n, that (len1-1) * (len2-1) + 1 ≤ n, and that res has space for n coefficients. Does not support aliasing between any of the inputs and the output.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978]. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.

```
void fmpz_poly_compose_series_brent_kung(fmpz_poly_t res, const fmpz_poly_t poly1, const
                                          fmpz_poly_t poly2, slong n)
```

Sets res to the composition of poly1 and poly2 modulo x^n , where the constant term of poly2 is required to be zero.

This implementation uses Brent-Kung algorithm 2.1 [BrentKung1978]. The default fmpz_poly composition algorithm is automatically used when the composition can be performed over the integers.

```
void _fmpz_poly_compose_series(fmpz *res, fmpz_t den, const fmpz *poly1, const fmpz_t den1,
                              slong len1, const fmpz *poly2, const fmpz_t den2, slong len2, slong
                              n)
```

Sets (res, den, n) to the composition of (poly1, den1, len1) and (poly2, den2, len2) modulo x^n , where the constant term of poly2 is required to be zero.

Assumes that `len1`, `len2`, `n` > 0 , that `len1`, `len2` $\leq n$, that $(len1-1) * (len2-1) + 1 \leq n$, and that `res` has space for `n` coefficients. Does not support aliasing between any of the inputs and the output.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

```
void fmpq_poly_compose_series(fmpq_poly_t res, const fmpq_poly_t poly1, const fmpq_poly_t
                             poly2, slong n)
```

Sets `res` to the composition of `poly1` and `poly2` modulo x^n , where the constant term of `poly2` is required to be zero.

This implementation automatically switches between the Horner scheme and Brent-Kung algorithm 2.1 depending on the size of the inputs. The default `fmpz_poly` composition algorithm is automatically used when the composition can be performed over the integers.

5.4.29 Power series reversion

```
void _fmpq_poly_revert_series_lagrange(fmpz_t res, fmpz_t den, const fmpz_t poly1, const fmpz_t
                                       den1, slong len1, slong n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1, len1)` modulo x^n .

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation uses the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void fmpq_poly_revert_series_lagrange(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Sets `res` to the power series reversion of `poly1` modulo x^n . The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation uses the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void _fmpq_poly_revert_series_lagrange_fast(fmpz_t res, fmpz_t den, const fmpz_t poly1, const
                                             fmpz_t den1, slong len1, slong n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1, len1)` modulo x^n .

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void fmpq_poly_revert_series_lagrange_fast(fmpq_poly_t res, const fmpq_poly_t poly, slong n)
```

Sets `res` to the power series reversion of `poly1` modulo x^n . The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation uses a reduced-complexity implementation of the Lagrange inversion formula. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

```
void _fmpq_poly_revert_series_newton(fmpz_t res, fmpz_t den, const fmpz_t poly1, const fmpz_t
                                     den1, slong len1, slong n)
```

Sets `(res, den)` to the power series reversion of `(poly1, den1, len1)` modulo x^n .

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation uses Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

void `fmpq_poly_revert_series_newton`(*fmpq_poly_t* res, const *fmpq_poly_t* poly, *slong* n)

Sets `res` to the power series reversion of `poly1` modulo x^n . The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation uses Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

void `_fmpq_poly_revert_series`(*fmpz_t* *res, *fmpz_t* den, const *fmpz_t* *poly1, const *fmpz_t* den1, *slong* len1, *slong* n)

Sets `(res, den)` to the power series reversion of `(poly1, den1, len1)` modulo x^n .

The constant term of `poly2` is required to be zero and the linear term is required to be nonzero. Assumes that $n > 0$. Does not support aliasing between any of the inputs and the output.

This implementation defaults to using Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

void `fmpq_poly_revert_series`(*fmpq_poly_t* res, const *fmpq_poly_t* poly, *slong* n)

Sets `res` to the power series reversion of `poly1` modulo x^n . The constant term of `poly2` is required to be zero and the linear term is required to be nonzero.

This implementation defaults to using Newton iteration. The default `fmpz_poly` reversion algorithm is automatically used when the reversion can be performed over the integers.

5.4.30 Gaussian content

void `_fmpq_poly_content`(*fmpq_t* res, const *fmpz_t* *poly, const *fmpz_t* den, *slong* len)

Sets `res` to the content of `(poly, den, len)`. If `len == 0`, sets `res` to zero.

void `fmpq_poly_content`(*fmpq_t* res, const *fmpq_poly_t* poly)

Sets `res` to the content of `poly`. The content of the zero polynomial is defined to be zero.

void `_fmpq_poly_primitive_part`(*fmpz_t* *rpoly, *fmpz_t* rden, const *fmpz_t* *poly, const *fmpz_t* den, *slong* len)

Sets `(rpoly, rden, len)` to the primitive part, with non-negative leading coefficient, of `(poly, den, len)`. Assumes that `len > 0`. Supports aliasing between the two polynomials.

void `fmpq_poly_primitive_part`(*fmpq_poly_t* res, const *fmpq_poly_t* poly)

Sets `res` to the primitive part, with non-negative leading coefficient, of `poly`.

int `_fmpq_poly_is_monic`(const *fmpz_t* *poly, const *fmpz_t* den, *slong* len)

Returns whether the polynomial `(poly, den, len)` is monic. The zero polynomial is not monic by definition.

int `fmpq_poly_is_monic`(const *fmpq_poly_t* poly)

Returns whether the polynomial `poly` is monic. The zero polynomial is not monic by definition.

void `_fmpq_poly_make_monic`(*fmpz_t* *rpoly, *fmpz_t* rden, const *fmpz_t* *poly, const *fmpz_t* den, *slong* len)

Sets `(rpoly, rden, len)` to the monic scalar multiple of `(poly, den, len)`. Assumes that `len > 0`. Supports aliasing between the two polynomials.

void `fmpq_poly_make_monic`(*fmpq_poly_t* res, const *fmpq_poly_t* poly)

Sets `res` to the monic scalar multiple of `poly` whenever `poly` is non-zero. If `poly` is the zero polynomial, sets `res` to zero.

5.4.31 Square-free

int **fmpq_poly_is_squarefree**(const *fmpq_poly_t* poly)

Returns whether the polynomial **poly** is square-free. A non-zero polynomial is defined to be square-free if it has no non-unit square factors. We also define the zero polynomial to be square-free.

5.4.32 Input and output

int **_fmpq_poly_print**(const *fmpz_t* *poly, const *fmpz_t* den, *slong* len)

Prints the polynomial (**poly**, **den**, **len**) to **stdout**.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fmpq_poly_print**(const *fmpq_poly_t* poly)

Prints the polynomial to **stdout**.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fmpq_poly_print_pretty**(const *fmpz_t* *poly, const *fmpz_t* den, *slong* len, const char *x)

int **fmpq_poly_print_pretty**(const *fmpq_poly_t* poly, const char *var)

Prints the pretty representation of **poly** to **stdout**, using the null-terminated string **var** not equal to "\0" as the variable name.

In the current implementation always returns 1.

int **_fmpq_poly_fprint**(FILE *file, const *fmpz_t* *poly, const *fmpz_t* den, *slong* len)

Prints the polynomial (**poly**, **den**, **len**) to the stream **file**.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fmpq_poly_fprint**(FILE *file, const *fmpq_poly_t* poly)

Prints the polynomial to the stream **file**.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fmpq_poly_fprint_pretty**(FILE *file, const *fmpz_t* *poly, const *fmpz_t* den, *slong* len, const char *x)

int **fmpq_poly_fprint_pretty**(FILE *file, const *fmpq_poly_t* poly, const char *var)

Prints the pretty representation of **poly** to **stdout**, using the null-terminated string **var** not equal to "\0" as the variable name.

In the current implementation, always returns 1.

int **fmpq_poly_read**(*fmpq_poly_t* poly)

Reads a polynomial from **stdin**, storing the result in **poly**.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

int **fmpq_poly_fread**(FILE *file, *fmpq_poly_t* poly)

Reads a polynomial from the stream **file**, storing the result in **poly**.

In case of success, returns a positive number. In case of failure, returns a non-positive value.

5.5 fmpq_mpoly_factor.h – factorisation of multivariate polynomials over the rational numbers

5.5.1 Types, macros and constants

type **fmpq_mpoly_factor_struct**

A struct for holding a factored rational polynomial. There is a single constant and a product of bases to corresponding exponents.

type **fmpq_mpoly_factor_t**

An array of length 1 of **fmpq_mpoly_factor_struct**.

5.5.2 Memory management

void **fmpq_mpoly_factor_init**(*fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

Initialise *f*.

void **fmpq_mpoly_factor_clear**(*fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

Clear *f*.

5.5.3 Basic manipulation

slong **fmpq_mpoly_factor_length**(const *fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

Return the length of the product in *f*.

void **fmpq_mpoly_factor_get_constant_fmpq**(*fmpq_t* c, const *fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

Set *c* to the constant of *f*.

void **fmpq_mpoly_factor_get_base**(*fmpq_mpoly_t* B, const *fmpq_mpoly_factor_t* f, *slong* i, const *fmpq_mpoly_ctx_t* ctx)

void **fmpq_mpoly_factor_swap_base**(*fmpq_mpoly_t* B, *fmpq_mpoly_factor_t* f, *slong* i, const *fmpq_mpoly_ctx_t* ctx)

Set (resp. swap) *B* to (resp. with) the base of the term of index *i* in *A*.

slong **fmpq_mpoly_factor_get_exp_si**(*fmpq_mpoly_factor_t* f, *slong* i, const *fmpq_mpoly_ctx_t* ctx)

Return the exponent of the term of index *i* in *A*. It is assumed to fit an **slong**.

void **fmpq_mpoly_factor_sort**(*fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

Sort the product of *f* first by exponent and then by base.

int **fmpq_mpoly_factor_make_monic**(*fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

int **fmpq_mpoly_factor_make_integral**(*fmpq_mpoly_factor_t* f, const *fmpq_mpoly_ctx_t* ctx)

Make the bases in *f* monic (resp. integral and primitive with positive leading coefficient). Return 1 for success, 0 for failure.

5.5.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A : f is simply set to a factorisation of A , and thus these functions should not depend on the initial value of the output f . The normalization of the factors is not yet specified: use `fmpq_mpoly_factor_make_monic()` or `fmpq_mpoly_factor_make_integral()` for common normalizations.

```
int fmpq_mpoly_factor_squarefree(fmpq_mpoly_factor_t f, const fmpq_mpoly_t A, const
                                fmpq_mpoly_ctx_t ctx)
```

Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call `fmpq_mpoly_factor_sort()` and then multiply the bases with the desired exponent.

```
int fmpq_mpoly_factor(fmpq_mpoly_factor_t f, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t
                     ctx)
```

Set f to a factorization of A where the bases are irreducible.

5.6 fmpq_mpoly.h – multivariate polynomials over the rational numbers

The exponents follow the `mpoly` interface. No references to the coefficients are available.

5.6.1 Types, macros and constants

```
type fmpq_mpoly_struct
```

A structure holding a multivariate rational polynomial. It is implemented as a `fmpq_t` holding the content of the polynomial and a primitive integer polynomial.

```
type fmpq_mpoly_t
```

An array of length 1 of `fmpq_mpoly_struct`.

```
type fmpq_mpoly_ctx_struct
```

Context structure representing the parent ring of an `fmpq_mpoly`.

```
type fmpq_mpoly_ctx_t
```

An array of length 1 of `fmpq_mpoly_ctx_struct`.

5.6.2 Context object

```
void fmpq_mpoly_ctx_init(fmpq_mpoly_ctx_t ctx, slong nvars, const ordering_t ord)
```

Initialise a context object for a polynomial ring with the given number of variables and the given ordering. The possibilities for the ordering are `ORD_LEX`, `ORD_DEGLEX` and `ORD_DEGREVLEX`.

```
slong fmpq_mpoly_ctx_nvars(const fmpq_mpoly_ctx_t ctx)
```

Return the number of variables used to initialize the context.

```
ordering_t fmpq_mpoly_ctx_ord(const fmpq_mpoly_ctx_t ctx)
```

Return the ordering used to initialize the context.

```
void fmpq_mpoly_ctx_clear(fmpq_mpoly_ctx_t ctx)
```

Release up any space allocated by `ctx`.

5.6.3 Memory management

void **fmpq_mpoly_init**(*fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Initialise *A* for use with the given and initialised context object. Its value is set to zero.

void **fmpq_mpoly_init2**(*fmpq_mpoly_t* A, *slong* alloc, const *fmpq_mpoly_ctx_t* ctx)

Initialise *A* for use with the given and initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and at least MPOLY_MIN_BITS bits for the exponents.

void **fmpq_mpoly_init3**(*fmpq_mpoly_t* A, *slong* alloc, *flint_bitcnt_t* bits, const *fmpq_mpoly_ctx_t* ctx)

Initialise *A* for use with the given and initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and *bits* bits for the exponents.

void **fmpq_mpoly_fit_length**(*fmpq_mpoly_t* A, *slong* len, const *fmpq_mpoly_ctx_t* ctx)

Ensure that *A* has space for at least *len* terms.

void **fmpq_mpoly_fit_bits**(*fmpq_mpoly_t* A, *flint_bitcnt_t* bits, const *fmpq_mpoly_ctx_t* ctx)

Ensure that the exponent fields of *A* have at least *bits* bits.

void **fmpq_mpoly_realloc**(*fmpq_mpoly_t* A, *slong* alloc, const *fmpq_mpoly_ctx_t* ctx)

Reallocate *A* to have space for *alloc* terms. Assumes the current length of the polynomial is not greater than *alloc*.

void **fmpq_mpoly_clear**(*fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Release any space allocated for *A*.

5.6.4 Input/Output

The variable strings in *x* start with the variable of most significance at index 0. If *x* is NULL, the variables are named *x1*, *x2*, etc.

char ***fmpq_mpoly_get_str_pretty**(const *fmpq_mpoly_t* A, const char ***x*, const *fmpq_mpoly_ctx_t* ctx)

Return a string, which the user is responsible for cleaning up, representing *A*, given an array of variable strings *x*.

int **fmpq_mpoly_fprint_pretty**(FILE **file*, const *fmpq_mpoly_t* A, const char ***x*, const *fmpq_mpoly_ctx_t* ctx)

Print a string representing *A* to *file*.

int **fmpq_mpoly_print_pretty**(const *fmpq_mpoly_t* A, const char ***x*, const *fmpq_mpoly_ctx_t* ctx)

Print a string representing *A* to *stdout*.

int **fmpq_mpoly_set_str_pretty**(*fmpq_mpoly_t* A, const char **str*, const char ***x*, const *fmpq_mpoly_ctx_t* ctx)

Set *A* to the polynomial in the null-terminates string *str* given an array *x* of variable strings. If parsing *str* fails, *A* is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in *x*. The character $\wedge$ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

5.6.5 Basic manipulation

void **fmpq_mpoly_gen**(*fmpq_mpoly_t* A, *slong* var, const *fmpq_mpoly_ctx_t* ctx)
 Set *A* to the variable of index *var*, where *var* = 0 corresponds to the variable with the most significance with respect to the ordering.

int **fmpq_mpoly_is_gen**(const *fmpq_mpoly_t* A, *slong* var, const *fmpq_mpoly_ctx_t* ctx)
 If *var* ≥ 0 , return 1 if *A* is equal to the *var*-th generator, otherwise return 0. If *var* < 0 , return 1 if the polynomial is equal to any generator, otherwise return 0.

void **fmpq_mpoly_set**(*fmpq_mpoly_t* A, const *fmpq_mpoly_t* B, const *fmpq_mpoly_ctx_t* ctx)
 Set *A* to *B*.

int **fmpq_mpoly_equal**(const *fmpq_mpoly_t* A, const *fmpq_mpoly_t* B, const *fmpq_mpoly_ctx_t* ctx)
 Return 1 if *A* is equal to *B*, else return 0.

void **fmpq_mpoly_swap**(*fmpq_mpoly_t* A, *fmpq_mpoly_t* B, const *fmpq_mpoly_ctx_t* ctx)
 Efficiently swap *A* and *B*.

5.6.6 Constants

int **fmpq_mpoly_is_fmpq**(const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)
 Return 1 if *A* is a constant, else return 0.

void **fmpq_mpoly_get_fmpq**(*fmpq_t* c, const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)
 Assuming that *A* is a constant, set *c* to this constant. This function throws if *A* is not a constant.

void **fmpq_mpoly_set_fmpq**(*fmpq_mpoly_t* A, const *fmpq_t* c, const *fmpq_mpoly_ctx_t* ctx)
 void **fmpq_mpoly_set_fmpz**(*fmpq_mpoly_t* A, const *fmpz_t* c, const *fmpq_mpoly_ctx_t* ctx)
 void **fmpq_mpoly_set_ui**(*fmpq_mpoly_t* A, *ulong* c, const *fmpq_mpoly_ctx_t* ctx)
 void **fmpq_mpoly_set_si**(*fmpq_mpoly_t* A, *slong* c, const *fmpq_mpoly_ctx_t* ctx)
 Set *A* to the constant *c*.

void **fmpq_mpoly_zero**(*fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)
 Set *A* to the constant 0.

void **fmpq_mpoly_one**(*fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)
 Set *A* to the constant 1.

int **fmpq_mpoly_equal_fmpq**(const *fmpq_mpoly_t* A, const *fmpq_t* c, const *fmpq_mpoly_ctx_t* ctx)
 int **fmpq_mpoly_equal_fmpz**(const *fmpq_mpoly_t* A, const *fmpz_t* c, const *fmpq_mpoly_ctx_t* ctx)
 int **fmpq_mpoly_equal_ui**(const *fmpq_mpoly_t* A, *ulong* c, const *fmpq_mpoly_ctx_t* ctx)
 int **fmpq_mpoly_equal_si**(const *fmpq_mpoly_t* A, *slong* c, const *fmpq_mpoly_ctx_t* ctx)
 Return 1 if *A* is equal to the constant *c*, else return 0.

int **fmpq_mpoly_is_zero**(const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)
 Return 1 if *A* is equal to the constant 0, else return 0.

int **fmpq_mpoly_is_one**(const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)
 Return 1 if *A* is equal to the constant 1, else return 0.

5.6.7 Degrees

int `fmpq_mpoly_degrees_fit_si`(const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Return 1 if the degrees of A with respect to each variable fit into an `slong`, otherwise return 0.

void `fmpq_mpoly_degrees_fmpz`(*fmpz* **deg, const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

void `fmpq_mpoly_degrees_si`(*slong* *deg, const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Set *deg*s to the degrees of A with respect to each variable. If A is zero, all degrees are set to -1 .

void `fmpq_mpoly_degree_fmpz`(*fmpz_t* deg, const *fmpq_mpoly_t* A, *slong* var, const *fmpq_mpoly_ctx_t* ctx)

slong `fmpq_mpoly_degree_si`(const *fmpq_mpoly_t* A, *slong* var, const *fmpq_mpoly_ctx_t* ctx)

Either return or set *deg* to the degree of A with respect to the variable of index *var*. If A is zero, the degree is defined to be -1 .

int `fmpq_mpoly_total_degree_fits_si`(const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Return 1 if the total degree of A fits into an `slong`, otherwise return 0.

void `fmpq_mpoly_total_degree_fmpz`(*fmpz_t* tdeg, const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

slong `fmpq_mpoly_total_degree_si`(const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Either return or set *tdeg* to the total degree of A . If A is zero, the total degree is defined to be -1 .

void `fmpq_mpoly_used_vars`(int *used, const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

For each variable index i , set `used[i]` to nonzero if the variable of index i appears in A and to zero otherwise.

5.6.8 Coefficients

void `fmpq_mpoly_get_denominator`(*fmpz_t* d, const *fmpq_mpoly_t* A, const *fmpq_mpoly_ctx_t* ctx)

Set d to the denominator of A , the smallest positive integer d such that $d \times A$ has integer coefficients.

void `fmpq_mpoly_get_coeff_fmpz_monomial`(*fmpz_t* c, const *fmpq_mpoly_t* A, const *fmpq_mpoly_t* M, const *fmpq_mpoly_ctx_t* ctx)

Assuming that M is a monomial, set c to the coefficient of the corresponding monomial in A . This function throws if M is not a monomial.

void `fmpq_mpoly_set_coeff_fmpz_monomial`(*fmpq_mpoly_t* A, const *fmpz_t* c, const *fmpq_mpoly_t* M, const *fmpq_mpoly_ctx_t* ctx)

Assuming that M is a monomial, set the coefficient of the corresponding monomial in A to c . This function throws if M is not a monomial.

void `fmpq_mpoly_get_coeff_fmpz_fmpz`(*fmpz_t* c, const *fmpq_mpoly_t* A, *fmpz* *const *exp, const *fmpq_mpoly_ctx_t* ctx)

void `fmpq_mpoly_get_coeff_fmpz_ui`(*fmpz_t* c, const *fmpq_mpoly_t* A, const *ulong* *exp, const *fmpq_mpoly_ctx_t* ctx)

Set c to the coefficient of the monomial with exponent *exp*.

void `fmpq_mpoly_set_coeff_fmpz_fmpz`(*fmpq_mpoly_t* A, const *fmpz_t* c, *fmpz* *const *exp, const *fmpq_mpoly_ctx_t* ctx)

void `fmpq_mpoly_set_coeff_fmpz_ui`(*fmpq_mpoly_t* A, const *fmpz_t* c, const *ulong* *exp, const *fmpq_mpoly_ctx_t* ctx)

Set the coefficient of the monomial with exponent *exp* to c .

```
void fmpq_mpoly_get_coeff_vars_ui(fmpq_mpoly_t C, const fmpq_mpoly_t A, const slong *vars,
                                const ulong *exps, slong length, const fmpq_mpoly_ctx_t ctx)
```

Set C to the coefficient of A with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \leq nvars(A)$ and that the variables in $vars$ are distinct.

5.6.9 Comparison

```
int fmpq_mpoly_cmp(const fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
```

Return 1 (resp. -1, or 0) if A is after (resp. before, same as) B in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when A and B are both monomials.

5.6.10 Container operations

These functions try to deal efficiently with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw. The mutating functions here are not guaranteed to leave the polynomial in reduced form (see `fmpq_mpoly_is_canonical()` for a definition of reduced). This means that even if nonzero terms with distinct exponents have been constructed in the correct order, a call to `fmpq_mpoly_reduce()` is necessary to ensure that the polynomial is in canonical form. As with the `fmpz_mpoly` module, a call to `fmpq_mpoly_sort_terms()` followed by a call to `fmpq_mpoly_combine_like_terms()` should leave the polynomial in canonical form.

```
fmpq *fmpq_mpoly_content_ref(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Return a reference to the content of A .

```
fmpz_mpoly_struct *fmpq_mpoly_zpoly_ref(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Return a reference to the integer polynomial of A .

```
fmpz *fmpq_mpoly_zpoly_term_coeff_ref(fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)
```

Return a reference to the coefficient of index i of the integer polynomial of A .

```
int fmpq_mpoly_is_canonical(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Return 1 if A is in canonical form. Otherwise, return 0. An `fmpq_mpoly_t` is represented as the product of an `fmpq_t` `content` and an `fmpz_mpoly_t` `zpoly`. The representation is considered canonical when either (1) both `content` and `zpoly` are zero, or (2) both `content` and `zpoly` are nonzero and canonical and `zpoly` is reduced. A nonzero `zpoly` is considered reduced when the coefficients have GCD one and the leading coefficient is positive.

```
slong fmpq_mpoly_length(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Return the number of terms stored in A . If the polynomial is in canonical form, this will be the number of nonzero coefficients.

```
void fmpq_mpoly_resize(fmpq_mpoly_t A, slong new_length, const fmpq_mpoly_ctx_t ctx)
```

Set the length of A to `new_length`. Terms are either deleted from the end, or new zero terms are appended.

```
void fmpq_mpoly_get_term_coeff_fmpq(fmpq_t c, const fmpq_mpoly_t A, slong i, const
                                    fmpq_mpoly_ctx_t ctx)
```

Set c to coefficient of index i

```
void fmpq_mpoly_set_term_coeff_fmpq(fmpq_mpoly_t A, slong i, const fmpq_t c, const
                                    fmpq_mpoly_ctx_t ctx)
```

Set the coefficient of index i to c .

```
int fmpq_mpoly_term_exp_fits_si(const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)
```

`int fmpq_mpoly_term_exp_fits_ui(const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)`
Return 1 if all entries of the exponent vector of the term of index i fit into an `slong` (resp. a `ulong`). Otherwise, return 0.

`void fmpq_mpoly_get_term_exp_fmpz(fmpz **exps, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_get_term_exp_ui(ulong *exps, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_get_term_exp_si(slong *exps, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)`
Set exp to the exponent vector of the term of index i . The `_ui` (resp. `_si`) version throws if any entry does not fit into a `ulong` (resp. `slong`).

`ulong fmpq_mpoly_get_term_var_exp_ui(const fmpq_mpoly_t A, slong i, slong var, const fmpq_mpoly_ctx_t ctx)`

`slong fmpq_mpoly_get_term_var_exp_si(const fmpq_mpoly_t A, slong i, slong var, const fmpq_mpoly_ctx_t ctx)`
Return the exponent of the variable var of the term of index i . This function throws if the exponent does not fit into a `ulong` (resp. `slong`).

`void fmpq_mpoly_set_term_exp_fmpz(fmpq_mpoly_t A, slong i, fmpz *const *exps, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_set_term_exp_ui(fmpq_mpoly_t A, slong i, const ulong *exps, const fmpq_mpoly_ctx_t ctx)`
Set the exponent vector of the term of index i to exp .

`void fmpq_mpoly_get_term(fmpq_mpoly_t M, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)`
Set M to the term of index i in A .

`void fmpq_mpoly_get_term_monomial(fmpq_mpoly_t M, const fmpq_mpoly_t A, slong i, const fmpq_mpoly_ctx_t ctx)`
Set M to the monomial of the term of index i in A . The coefficient of M will be one.

`void fmpq_mpoly_push_term_fmpz_fmpz(fmpq_mpoly_t A, const fmpq_t c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_fmpz_ffmpz(fmpq_mpoly_t A, const fmpq_t c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_fmpz_fmpz(fmpq_mpoly_t A, const fmpz_t c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_fmpz_ffmpz(fmpq_mpoly_t A, const fmpz_t c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_ui_fmpz(fmpq_mpoly_t A, ulong c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_ui_ffmpz(fmpq_mpoly_t A, ulong c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_si_fmpz(fmpq_mpoly_t A, slong c, fmpz *const *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_si_ffmpz(fmpq_mpoly_t A, slong c, const fmpz *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_fmpz_ui(fmpq_mpoly_t A, const fmpq_t c, const ulong *exp, const fmpq_mpoly_ctx_t ctx)`

`void fmpq_mpoly_push_term_fmpz_ui(fmpq_mpoly_t A, const fmpz_t c, const ulong *exp, const fmpq_mpoly_ctx_t ctx)`

```
void fmpq_mpoly_push_term_ui_ui(fmpq_mpoly_t A, ulong c, const ulong *exp, const
                                fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_push_term_si_ui(fmpq_mpoly_t A, slong c, const ulong *exp, const
                                fmpq_mpoly_ctx_t ctx)
```

Append a term to A with coefficient c and exponent vector exp . This function should run in constant average time if the terms pushed have bounded denominator.

```
void fmpq_mpoly_reduce(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Factor out necessary content from $A \rightarrow \text{zpoly}$ so that it is reduced. If the terms of A were nonzero and sorted with distinct exponents to begin with, the result will be in canonical form.

```
void fmpq_mpoly_sort_terms(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Sort the internal $A \rightarrow \text{zpoly}$ into the canonical ordering dictated by the ordering in ctx . This function does not combine like terms, nor does it delete terms with coefficient zero, nor does it reduce.

```
void fmpq_mpoly_combine_like_terms(fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Combine adjacent like terms in the internal $A \rightarrow \text{zpoly}$ and then factor out content via a call to `fmpq_mpoly_reduce()`. If the terms of A were sorted to begin with, the result will be in canonical form.

```
void fmpq_mpoly_reverse(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
```

Set A to the reversal of B .

5.6.11 Random generation

```
void fmpq_mpoly_randtest_bound(fmpq_mpoly_t A, flint_rand_t state, slong length, ulong
                                coeff_bits, ulong exp_bound, const fmpq_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents in the range $[0, \text{exp\_bound} - 1]$. The exponents of each variable are generated by calls to `n_randint(state, exp_bound)`.

```
void fmpq_mpoly_randtest_bounds(fmpq_mpoly_t A, flint_rand_t state, slong length, ulong
                                coeff_bits, ulong *exp_bounds, const fmpq_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents in the range $[0, \text{exp\_bounds}[i] - 1]$. The exponents of the variable of index i are generated by calls to `n_randint(state, exp_bounds[i])`.

```
void fmpq_mpoly_randtest_bits(fmpq_mpoly_t A, flint_rand_t state, slong length, ulong coeff_bits,
                                ulong exp_bits, const fmpq_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents whose packed form does not exceed the given bit count.

The parameter `coeff_bits` to the three functions `fmpq_mpoly_randtest_{bound|bounds|bits}` is merely a suggestion for the approximate bit count of the resulting coefficients.

5.6.12 Addition/Subtraction

```
void fmpq_mpoly_add_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const
                            fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_add_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const
                            fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_add_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const fmpq_mpoly_ctx_t
                        ctx)
```

```
void fmpq_mpoly_add_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const fmpq_mpoly_ctx_t
                        ctx)
```

Set A to $B + c$.

```
void fmpq_mpoly_sub_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const
                        fmpq_mpoly_ctx_t ctx)
void fmpq_mpoly_sub_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const
                        fmpq_mpoly_ctx_t ctx)
void fmpq_mpoly_sub_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const fmpq_mpoly_ctx_t
                      ctx)
void fmpq_mpoly_sub_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const fmpq_mpoly_ctx_t
                      ctx)
```

Set A to $B - c$.

```
void fmpq_mpoly_add(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_t C, const
                   fmpq_mpoly_ctx_t ctx)
```

Set A to $B + C$.

```
void fmpq_mpoly_sub(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_t C, const
                   fmpq_mpoly_ctx_t ctx)
```

Set A to $B - C$.

5.6.13 Scalar operations

```
void fmpq_mpoly_neg(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
Set  $A$  to  $-B$ .
```

```
void fmpq_mpoly_scalar_mul_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const
                                fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_scalar_mul_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const
                                fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_scalar_mul_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const
                              fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_scalar_mul_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const
                              fmpq_mpoly_ctx_t ctx)
```

Set A to $B \times c$.

```
void fmpq_mpoly_scalar_div_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_t c, const
                                fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_scalar_div_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t c, const
                                fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_scalar_div_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong c, const
                              fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_scalar_div_si(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong c, const
                              fmpq_mpoly_ctx_t ctx)
```

Set A to B/c .

```
void fmpq_mpoly_make_monic(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t
                           ctx)
```

Set A to B divided by the leading coefficient of B . This throws if B is zero.

All of these functions run quickly if A and B are aliased.

5.6.14 Differentiation/Integration

```
void fmpq_mpoly_derivative(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong var, const
                          fmpq_mpoly_ctx_t ctx)
```

Set A to the derivative of B with respect to the variable of index var .

```
void fmpq_mpoly_integral(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong var, const
                        fmpq_mpoly_ctx_t ctx)
```

Set A to the integral with the fewest number of terms of B with respect to the variable of index var .

5.6.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int fmpq_mpoly_evaluate_all_fmpq(fmpq_t ev, const fmpq_mpoly_t A, fmpq *const *vals, const
                                fmpq_mpoly_ctx_t ctx)
```

Set ev to the evaluation of A where the variables are replaced by the corresponding elements of the array $vals$. Return 1 for success and 0 for failure.

```
int fmpq_mpoly_evaluate_one_fmpq(fmpq_mpoly_t A, const fmpq_mpoly_t B, slong var, const
                                fmpq_t val, const fmpq_mpoly_ctx_t ctx)
```

Set A to the evaluation of B where the variable of index var is replaced by val . Return 1 for success and 0 for failure.

```
int fmpq_mpoly_compose_fmpq_poly(fmpq_poly_t A, const fmpq_mpoly_t B, fmpq_poly_struct
                                *const *C, const fmpq_mpoly_ctx_t ctxB)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . The context object of B is $ctxB$. Return 1 for success and 0 for failure.

```
int fmpq_mpoly_compose_fmpq_mpoly(fmpq_mpoly_t A, const fmpq_mpoly_t B, fmpq_mpoly_struct
                                *const *C, const fmpq_mpoly_ctx_t ctxB, const
                                fmpq_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . Both A and the elements of C have context object $ctxAC$, while B has context object $ctxB$. Neither A nor B is allowed to alias any other polynomial. Return 1 for success and 0 for failure.

```
void fmpq_mpoly_compose_fmpq_mpoly_gen(fmpq_mpoly_t A, const fmpq_mpoly_t B, const slong *c,
                                       const fmpq_mpoly_ctx_t ctxB, const fmpq_mpoly_ctx_t
                                       ctxAC)
```

Set A to the evaluation of B where the variable of index i in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array C is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.

5.6.16 Multiplication

```
void fmpq_mpoly_mul(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_t C, const
                  fmpq_mpoly_ctx_t ctx)
```

Set A to $B \times C$.

5.6.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int fmpq_mpoly_pow_fmpz(fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpz_t k, const
                        fmpq_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

```
int fmpq_mpoly_pow_ui(fmpq_mpoly_t A, const fmpq_mpoly_t B, ulong k, const fmpq_mpoly_ctx_t
                     ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

5.6.18 Division

```
int fmpq_mpoly_divides(fmpq_mpoly_t Q, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const
                      fmpq_mpoly_ctx_t ctx)
```

If A is divisible by B , set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0. Note that the function `fmpq_mpoly_div()` may be faster if the quotient is known to be exact.

```
void fmpq_mpoly_div(fmpq_mpoly_t Q, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const
                   fmpq_mpoly_ctx_t ctx)
```

Set Q to the quotient of A by B , discarding the remainder.

```
void fmpq_mpoly_divrem(fmpq_mpoly_t Q, fmpq_mpoly_t R, const fmpq_mpoly_t A, const
                     fmpq_mpoly_t B, const fmpq_mpoly_ctx_t ctx)
```

Set Q and R to the quotient and remainder of A divided by B .

```
void fmpq_mpoly_divrem_ideal(fmpq_mpoly_struct **Q, fmpq_mpoly_t R, const fmpq_mpoly_t A,
                           fmpq_mpoly_struct *const *B, slong len, const fmpq_mpoly_ctx_t
                           ctx)
```

This function is as per `fmpq_mpoly_divrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q . The number of divisor (and hence quotient) polynomials is given by len .

5.6.19 Greatest Common Divisor

```
void fmpq_mpoly_content(fmpq_t g, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
```

Set g to the (nonnegative) gcd of the coefficients of A .

```
void fmpq_mpoly_term_content(fmpq_mpoly_t M, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t
                           ctx)
```

Set M to the GCD of the terms of A . If A is zero, M will be zero. Otherwise, M will be a monomial with coefficient one.

```
int fmpq_mpoly_content_vars(fmpq_mpoly_t g, const fmpq_mpoly_t A, slong *vars, slong
                          vars_length, const fmpq_mpoly_ctx_t ctx)
```

Set g to the GCD of the coefficients of A when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, g will be independent of the variables $vars$.

```
int fmpq_mpoly_gcd(fmpq_mpoly_t G, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const
                  fmpq_mpoly_ctx_t ctx)
```

Try to set G to the monic GCD of A and B . The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and G is left untouched.

```
int fmpq_mpoly_gcd_cofactors(fmpq_mpoly_t G, fmpq_mpoly_t Abar, fmpq_mpoly_t Bbar, const
                           fmpq_mpoly_t A, const fmpq_mpoly_t B, const fmpq_mpoly_ctx_t
                           ctx)
```

Do the operation of `fmpq_mpoly_gcd()` and also compute $Abar = A/G$ and $Bbar = B/G$ if successful.

```
int fmpq_mpoly_gcd_brown(fmpq_mpoly_t G, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const
                        fmpq_mpoly_ctx_t ctx)
int fmpq_mpoly_gcd_hensel(fmpq_mpoly_t G, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const
                          fmpq_mpoly_ctx_t ctx)
int fmpq_mpoly_gcd_subresultant(fmpq_mpoly_t G, const fmpq_mpoly_t A, const fmpq_mpoly_t
                                B, const fmpq_mpoly_ctx_t ctx)
int fmpq_mpoly_gcd_zippel(fmpq_mpoly_t G, const fmpq_mpoly_t A, const fmpq_mpoly_t B, const
                           fmpq_mpoly_ctx_t ctx)
int fmpq_mpoly_gcd_zippel2(fmpq_mpoly_t G, const fmpq_mpoly_t A, const fmpq_mpoly_t B,
                           const fmpq_mpoly_ctx_t ctx)
```

Try to set G to the GCD of A and B using various algorithms.

```
int fmpq_mpoly_resultant(fmpq_mpoly_t R, const fmpq_mpoly_t A, const fmpq_mpoly_t B, slong
                        var, const fmpq_mpoly_ctx_t ctx)
```

Try to set R to the resultant of A and B with respect to the variable of index var .

```
int fmpq_mpoly_discriminant(fmpq_mpoly_t D, const fmpq_mpoly_t A, slong var, const
                             fmpq_mpoly_ctx_t ctx)
```

Try to set D to the discriminant of A with respect to the variable of index var .

5.6.20 Square Root

```
int fmpq_mpoly_sqrt(fmpq_mpoly_t Q, const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    If  $A$  is a perfect square return 1 and set  $Q$  to the square root with positive leading coefficient.
    Otherwise return 0 and set  $Q$  to zero.
int fmpq_mpoly_is_square(const fmpq_mpoly_t A, const fmpq_mpoly_ctx_t ctx)
    Return 1 if  $A$  is a perfect square, otherwise return 0.
```

5.6.21 Univariate Functions

An `fmpq_mpoly_univar_t` holds a univariate polynomial in some main variable with `fmpq_mpoly_t` coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Q}[x_1, \dots, x_m]$ as an element of $(\mathbb{Q}[x_1, \dots, x_{v-1}, x_{v+1}, \dots, x_m])[x_v]$ and vice versa.

```
void fmpq_mpoly_univar_init(fmpq_mpoly_univar_t A, const fmpq_mpoly_ctx_t ctx)
    Initialize  $A$ .
void fmpq_mpoly_univar_clear(fmpq_mpoly_univar_t A, const fmpq_mpoly_ctx_t ctx)
    Clear  $A$ .
void fmpq_mpoly_univar_swap(fmpq_mpoly_univar_t A, fmpq_mpoly_univar_t B, const
                             fmpq_mpoly_ctx_t ctx)
    Swap  $A$  and  $B$ .
void fmpq_mpoly_to_univar(fmpq_mpoly_univar_t A, const fmpq_mpoly_t B, slong var, const
                           fmpq_mpoly_ctx_t ctx)
```

Set A to a univariate form of B by pulling out the variable of index var . The coefficients of A will still belong to the content ctx but will not depend on the variable of index var .

```
void fmpq_mpoly_from_univar(fmpq_mpoly_t A, const fmpq_mpoly_univar_t B, slong var, const
                           fmpq_mpoly_ctx_t ctx)
```

Set A to the normal form of B by putting in the variable of index var . This function is undefined if the coefficients of B depend on the variable of index var .

```
int fmpq_mpoly_univar_degree_fits_si(const fmpq_mpoly_univar_t A, const fmpq_mpoly_ctx_t
                                     ctx)
```

Return 1 if the degree of A with respect to the main variable fits an *slong*. Otherwise, return 0.

```
slong fmpq_mpoly_univar_length(const fmpq_mpoly_univar_t A, const fmpq_mpoly_ctx_t ctx)
```

Return the number of terms in A with respect to the main variable.

```
slong fmpq_mpoly_univar_get_term_exp_si(fmpq_mpoly_univar_t A, slong i, const
                                         fmpq_mpoly_ctx_t ctx)
```

Return the exponent of the term of index i of A .

```
void fmpq_mpoly_univar_get_term_coeff(fmpq_mpoly_t c, const fmpq_mpoly_univar_t A, slong i,
                                      const fmpq_mpoly_ctx_t ctx)
```

```
void fmpq_mpoly_univar_swap_term_coeff(fmpq_mpoly_t c, fmpq_mpoly_univar_t A, slong i,
                                       const fmpq_mpoly_ctx_t ctx)
```

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A .

5.7 fmpz_poly_q.h – rational functions over the rational numbers

The module `fmpz_poly_q` provides functions for performing arithmetic on rational functions in $\mathbf{Q}(t)$, represented as quotients of integer polynomials of type `fmpz_poly_t`. These functions start with the prefix `fmpz_poly_q_`.

Rational functions are stored in objects of type `fmpz_poly_q_t`, which is an array of `fmpz_poly_q_struct`'s of length one. This permits passing parameters of type `fmpz_poly_q_t` by reference.

The representation of a rational function as the quotient of two integer polynomials can be made canonical by demanding the numerator and denominator to be coprime (as integer polynomials) and the denominator to have positive leading coefficient. As the only special case, we represent the zero function as $0/1$. All arithmetic functions assume that the operands are in this canonical form, and canonicalize their result. If the numerator or denominator is modified individually, for example using the macros `fmpz_poly_q_numref()` and `fmpz_poly_q_denref()`, it is the user's responsibility to canonicalise the rational function using the function `fmpz_poly_q_canonicalise()` if necessary.

All methods support aliasing of their inputs and outputs *unless* explicitly stated otherwise, subject to the following caveat. If different rational functions (as objects in memory, not necessarily in the mathematical sense) share some of the underlying integer polynomial objects, the behaviour is undefined.

The basic arithmetic operations, addition, subtraction and multiplication, are all implemented using adapted versions of Henrici's algorithms, see [Hen1956]. Differentiation is implemented in a way slightly improving on the algorithm described in [Hor1972].

5.7.1 Simple example

The following example computes the product of two rational functions and prints the result:

```
#include "fmpz_poly_q.h"
int main()
{
    char * str, * strf, * strg;
    fmpz_poly_q_t f, g;
    fmpz_poly_q_init(f);
    fmpz_poly_q_init(g);
    fmpz_poly_q_set_str(f, "2 1 3/1 2");
    fmpz_poly_q_set_str(g, "1 3/2 2 7");
    strf = fmpz_poly_q_get_str_pretty(f, "t");
    strg = fmpz_poly_q_get_str_pretty(g, "t");
    fmpz_poly_q_mul(f, f, g);
    str = fmpz_poly_q_get_str_pretty(f, "t");
    flint_printf("%s * %s = %s\n", strf, strg, str);
    free(str);
    free(strf);
    free(strg);
    fmpz_poly_q_clear(f);
    fmpz_poly_q_clear(g);
}
```

The output is:

```
(3*t+1)/2 * 3/(7*t+2) = (9*t+3)/(14*t+4)
```

5.7.2 Types, macros and constants

type `fmpz_poly_q_struct`

type `fmpz_poly_q_t`

5.7.3 Memory management

void `fmpz_poly_q_init`(*fmpz_poly_q_t* rop)

Initialises rop.

void `fmpz_poly_q_clear`(*fmpz_poly_q_t* rop)

Clears the object rop.

fmpz_poly_struct *`fmpz_poly_q_numref`(const *fmpz_poly_q_t* op)

Returns a reference to the numerator of op.

fmpz_poly_struct *`fmpz_poly_q_denref`(const *fmpz_poly_q_t* op)

Returns a reference to the denominator of op.

void `fmpz_poly_q_canonicalise`(*fmpz_poly_q_t* rop)

Brings rop into canonical form, only assuming that the denominator is non-zero.

int `fmpz_poly_q_is_canonical`(const *fmpz_poly_q_t* op)

Checks whether the rational function op is in canonical form.

5.7.4 Randomisation

void **fmpz_poly_q_randtest**(*fmpz_poly_q_t* poly, *flint_rand_t* state, *slong* len1, *flint_bitcnt_t* bits1, *slong* len2, *flint_bitcnt_t* bits2)

Sets *poly* to a random rational function.

void **fmpz_poly_q_randtest_not_zero**(*fmpz_poly_q_t* poly, *flint_rand_t* state, *slong* len1, *flint_bitcnt_t* bits1, *slong* len2, *flint_bitcnt_t* bits2)

Sets *poly* to a random non-zero rational function.

5.7.5 Assignment

void **fmpz_poly_q_set**(*fmpz_poly_q_t* rop, const *fmpz_poly_q_t* op)

Sets the element *rop* to the same value as the element *op*.

void **fmpz_poly_q_set_si**(*fmpz_poly_q_t* rop, *slong* op)

Sets the element *rop* to the value given by the *slong* *op*.

void **fmpz_poly_q_swap**(*fmpz_poly_q_t* op1, *fmpz_poly_q_t* op2)

Swaps the elements *op1* and *op2*.

This is done efficiently by swapping pointers.

void **fmpz_poly_q_zero**(*fmpz_poly_q_t* rop)

Sets *rop* to zero.

void **fmpz_poly_q_one**(*fmpz_poly_q_t* rop)

Sets *rop* to one.

void **fmpz_poly_q_neg**(*fmpz_poly_q_t* rop, const *fmpz_poly_q_t* op)

Sets the element *rop* to the additive inverse of *op*.

void **fmpz_poly_q_inv**(*fmpz_poly_q_t* rop, const *fmpz_poly_q_t* op)

Sets the element *rop* to the multiplicative inverse of *op*.

Assumes that the element *op* is non-zero.

5.7.6 Comparison

int **fmpz_poly_q_is_zero**(const *fmpz_poly_q_t* op)

Returns whether the element *op* is zero.

int **fmpz_poly_q_is_one**(const *fmpz_poly_q_t* op)

Returns whether the element *rop* is equal to the constant polynomial 1.

int **fmpz_poly_q_equal**(const *fmpz_poly_q_t* op1, const *fmpz_poly_q_t* op2)

Returns whether the two elements *op1* and *op2* are equal.

5.7.7 Addition and subtraction

void **fmprz_poly_q_add**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op1, const *fmprz_poly_q_t* op2)
Sets rop to the sum of op1 and op2.

void **fmprz_poly_q_sub**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op1, const *fmprz_poly_q_t* op2)
Sets rop to the difference of op1 and op2.

void **fmprz_poly_q_addmul**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op1, const *fmprz_poly_q_t* op2)
Adds the product of op1 and op2 to rop.

void **fmprz_poly_q_submul**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op1, const *fmprz_poly_q_t* op2)
Subtracts the product of op1 and op2 from rop.

5.7.8 Scalar multiplication and division

void **fmprz_poly_q_scalar_mul_si**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, *slong* x)
Sets rop to the product of the rational function op and the *slong* integer x.

void **fmprz_poly_q_scalar_mul_fmpz**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, const *fmpz_t* x)
Sets rop to the product of the rational function op and the *fmpz_t* integer x.

void **fmprz_poly_q_scalar_mul_fmpq**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, const *fmpq_t* x)
Sets rop to the product of the rational function op and the *fmpq_t* rational x.

void **fmprz_poly_q_scalar_div_si**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, *slong* x)
Sets rop to the quotient of the rational function op and the *slong* integer x.

void **fmprz_poly_q_scalar_div_fmpz**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, const *fmpz_t* x)
Sets rop to the quotient of the rational function op and the *fmpz_t* integer x.

void **fmprz_poly_q_scalar_div_fmpq**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, const *fmpq_t* x)
Sets rop to the quotient of the rational function op and the *fmpq_t* rational x.

5.7.9 Multiplication and division

void **fmprz_poly_q_mul**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op1, const *fmprz_poly_q_t* op2)
Sets rop to the product of op1 and op2.

void **fmprz_poly_q_div**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op1, const *fmprz_poly_q_t* op2)
Sets rop to the quotient of op1 and op2.

5.7.10 Powering

void **fmprz_poly_q_pow**(*fmprz_poly_q_t* rop, const *fmprz_poly_q_t* op, *ulong* exp)
Sets rop to the exp-th power of op.

The corner case of `exp == 0` is handled by setting `rop` to the constant function 1. Note that this includes the case $0^0 = 1$.

5.7.11 Derivative

void **fmpz_poly_q_derivative**(*fmpz_poly_q_t* rop, const *fmpz_poly_q_t* op)
 Sets rop to the derivative of op.

5.7.12 Evaluation

int **fmpz_poly_q_evaluate_fmpq**(*fmpq_t* rop, const *fmpz_poly_q_t* f, const *fmpq_t* a)
 Sets rop to f evaluated at the rational a .
 If the denominator evaluates to zero at a , returns non-zero and does not modify any of the variables.
 Otherwise, returns 0 and sets rop to the rational $f(a)$.

5.7.13 Input and output

The following three methods enable users to construct elements of type *fmpz_poly_q_t* from strings or to obtain string representations of such elements. The format used is based on the FLINT format for integer polynomials of type *fmpz_poly_t*, which we recall first: A non-zero polynomial $a_0 + a_1X + \dots + a_nX^n$ of length $n + 1$ is represented by the string "**n+1 a_0 a_1 ... a_n**", where there are two space characters following the length and single space characters separating the individual coefficients. There is no leading or trailing white-space. The zero polynomial is simply represented by "0". We adapt this notation for rational functions as follows. We denote the zero function by "0". Given a non-zero function with numerator and denominator string representations **num** and **den**, respectively, we use the string **num/den** to represent the rational function, unless the denominator is equal to one, in which case we simply use **num**. There is also a **_pretty** variant available, which bases the string parts for the numerator and denominator on the output of the function **fmpz_poly_get_str_pretty** and introduces parentheses where necessary. Note that currently these functions are not optimised for performance and are intended to be used only for debugging purposes or one-off input and output, rather than as a low-level parser.

int **fmpz_poly_q_set_str**(*fmpz_poly_q_t* rop, const char *s)
 Sets rop to the rational function given by the string **s**.
 char ***fmpz_poly_q_get_str**(const *fmpz_poly_q_t* op)
 Returns the string representation of the rational function op.
 char ***fmpz_poly_q_get_str_pretty**(const *fmpz_poly_q_t* op, const char *x)
 Returns the pretty string representation of the rational function op.
 int **fmpz_poly_q_print**(const *fmpz_poly_q_t* op)
 Prints the representation of the rational function op to **stdout**.
 int **fmpz_poly_q_print_pretty**(const *fmpz_poly_q_t* op, const char *x)
 Prints the pretty representation of the rational function op to **stdout**.

5.8 fmpz_mpoly_q.h – multivariate rational functions over $\mathbb{Q}$

An *fmpz_mpoly_q_t* represents an element of $\mathbb{Q}(x_1, \dots, x_n)$ for fixed n as a pair of FLINT multivariate polynomials (*fmpz_mpoly_t*). Instances are always kept in canonical form by ensuring that the GCD of numerator and denominator is 1 and that the coefficient of the leading term of the denominator is positive.

The user must create a multivariate polynomial context (*fmpz_mpoly_ctx_t*) specifying the number of variables n and the monomial ordering.

5.8.1 Types and macros

type `fmpz_mpoly_q_struct`

type `fmpz_mpoly_q_t`

An *fmpz_mpoly_q_struct* consists of a pair of *fmpz_mpoly_struct*s. An *fmpz_mpoly_q_t* is defined as an array of length one of type *fmpz_mpoly_q_struct*, permitting an *fmpz_mpoly_q_t* to be passed by reference.

`fmpz_mpoly_q_numref(x)`

Macro returning a pointer to the numerator of *x* which can be used as an *fmpz_mpoly_t*.

`fmpz_mpoly_q_denref(x)`

Macro returning a pointer to the denominator of *x* which can be used as an *fmpz_mpoly_t*.

5.8.2 Memory management

void `fmpz_mpoly_q_init(fmpz_mpoly_q_t res, const fmpz_mpoly_ctx_t ctx)`

Initializes *res* for use, and sets its value to zero.

void `fmpz_mpoly_q_clear(fmpz_mpoly_q_t res, const fmpz_mpoly_ctx_t ctx)`

Clears *res*, freeing or recycling its allocated memory.

5.8.3 Assignment

void `fmpz_mpoly_q_swap(fmpz_mpoly_q_t x, fmpz_mpoly_q_t y, const fmpz_mpoly_ctx_t ctx)`

Swaps the values of *x* and *y* efficiently.

void `fmpz_mpoly_q_set(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)`

void `fmpz_mpoly_q_set_fmpz(fmpz_mpoly_q_t res, const fmpz_t x, const fmpz_mpoly_ctx_t ctx)`

void `fmpz_mpoly_q_set_si(fmpz_mpoly_q_t res, slong x, const fmpz_mpoly_ctx_t ctx)`

Sets *res* to the value *x*.

5.8.4 Canonicalisation

void `fmpz_mpoly_q_canonicalise(fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)`

Puts the numerator and denominator of *x* in canonical form by removing common content and making the leading term of the denominator positive.

int `fmpz_mpoly_q_is_canonical(const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)`

Returns whether *x* is in canonical form.

In addition to verifying that the numerator and denominator have no common content and that the leading term of the denominator is positive, this function checks that the denominator is nonzero and that the numerator and denominator have correctly sorted terms (these properties should normally hold; verifying them provides an extra consistency check for test code).

5.8.5 Properties

int **fmpz_mpoly_q_is_zero**(const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_ctx_t* ctx)

Returns whether x is the constant 0.

int **fmpz_mpoly_q_is_one**(const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_ctx_t* ctx)

Returns whether x is the constant 1.

void **fmpz_mpoly_q_used_vars**(int *used, const *fmpz_mpoly_q_t* f, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_used_vars_num**(int *used, const *fmpz_mpoly_q_t* f, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_used_vars_den**(int *used, const *fmpz_mpoly_q_t* f, const *fmpz_mpoly_ctx_t* ctx)

For each variable, sets the corresponding entry in *used* to the boolean flag indicating whether that variable appears in the rational function (respectively its numerator or denominator).

5.8.6 Special values

void **fmpz_mpoly_q_zero**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the constant 0.

void **fmpz_mpoly_q_one**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the constant 1.

void **fmpz_mpoly_q_gen**(*fmpz_mpoly_q_t* res, *long* i, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the generator x_{i+1} . Requires $0 \leq i < n$ where n is the number of variables of *ctx*.

5.8.7 Input and output

The variable strings in x start with the variable of most significance at index 0. If x is *NULL*, the variables are named x_1 , x_2 , etc.

void **fmpz_mpoly_q_print_pretty**(const *fmpz_mpoly_q_t* f, const char **x, const *fmpz_mpoly_ctx_t* ctx)

Prints *res* to standard output. If x is not *NULL*, the strings in x are used as the symbols for the variables.

char ***fmpz_mpoly_q_get_str_pretty**(const *fmpz_mpoly_q_t* f, const char **x, const *fmpz_mpoly_ctx_t* ctx)

Return a string, which the user is responsible for cleaning up, representing f , given an array of variable strings x .

int **fmpz_mpoly_q_set_str_pretty**(*fmpz_mpoly_q_t* res, const char *s, const char **x, *fmpz_mpoly_ctx_t* ctx)

Set *res* to the fraction in the null-terminated string *str* given an array x of variable strings. If parsing *str* fails, *res* is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in x . The character $^$ must be immediately followed by the (integer) exponent. If division by zero occurs, parsing fails.

5.8.8 Random generation

void **fmpz_mpoly_q_randtest**(*fmpz_mpoly_q_t* res, *flint_rand_t* state, *slong* length, *ulong* coeff_bits, *slong* exp_bound, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to a random rational function where both numerator and denominator have up to *length* terms, coefficients up to size *coeff_bits*, and exponents strictly smaller than *exp_bound*.

5.8.9 Comparisons

int **fmpz_mpoly_q_equal**(const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_q_t* y, const *fmpz_mpoly_ctx_t* ctx)

Returns whether *x* and *y* are equal.

5.8.10 Arithmetic

void **fmpz_mpoly_q_neg**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the negation of *x*.

void **fmpz_mpoly_q_add**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_q_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_add_fmpzq**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpzq_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_add_fmpz**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_add_si**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, *slong* y, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the sum of *x* and *y*.

void **fmpz_mpoly_q_sub**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_q_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_sub_fmpzq**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpzq_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_sub_fmpz**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_sub_si**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, *slong* y, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the difference of *x* and *y*.

void **fmpz_mpoly_q_mul**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_q_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_mul_fmpzq**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpzq_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_mul_fmpz**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_mul_si**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, *slong* y, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the product of *x* and *y*.

void **fmpz_mpoly_q_div**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpz_mpoly_q_t* y, const *fmpz_mpoly_ctx_t* ctx)

void **fmpz_mpoly_q_div_fmpzq**(*fmpz_mpoly_q_t* res, const *fmpz_mpoly_q_t* x, const *fmpzq_t* y, const *fmpz_mpoly_ctx_t* ctx)

```
void fmpz_mpoly_q_div_fmpz(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_t y, const  
                          fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_q_div_si(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, slong y, const  
                        fmpz_mpoly_ctx_t ctx)
```

Sets *res* to the quotient of *x* and *y*. Division by zero calls *flint_abort*.

```
void fmpz_mpoly_q_inv(fmpz_mpoly_q_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t  
                    ctx)
```

Sets *res* to the inverse of *x*. Division by zero calls *flint_abort*.

5.8.11 Content

```
void _fmpz_mpoly_q_content(fmpz_t num, fmpz_t den, const fmpz_mpoly_t xnum, const  
                          fmpz_mpoly_t xden, const fmpz_mpoly_ctx_t ctx)
```

```
void fmpz_mpoly_q_content(fmpz_t res, const fmpz_mpoly_q_t x, const fmpz_mpoly_ctx_t ctx)
```

Sets *res* to the content of the coefficients of *x*.

INTEGERS MOD N

6.1 `nmod.h` – integers mod n (word-size n)

6.1.1 Generic rings

```
int gr_ctx_init_nmod(gr_ctx_t ctx, ulong n)
int gr_ctx_init_nmod8(gr_ctx_t ctx, ulong n)
int gr_ctx_init_nmod32(gr_ctx_t ctx, ulong n)
int gr_ctx_init_nmod_redc(gr_ctx_t ctx, ulong n)
int gr_ctx_init_nmod_redc_fast(gr_ctx_t ctx, ulong n)
```

Initialize the context object `ctx` for generic arithmetic in the ring $R = \mathbb{Z}/n\mathbb{Z}$ for word-size n using the respective representation.

The element datatype is `ulong` for all representations except for `nmod8` and `nmod32` where the type is `uint8` or `uint32` respectively.

If the modulus is valid, returns `GR_SUCCESS`. Otherwise, the context object is not initialized and an error code is returned:

- `GR_DOMAIN` is returned if $n = 0$.
- `GR_UNABLE` is returned if n does not satisfy the technical restrictions of a specific representation. For example, `nmod8` requires $n \leq 255$ and `nmod_redc` requires that n is odd.

Note: Some generic algorithms need to check whether R is a field, i.e. whether n is a prime number. To avoid expensive on-the-fly primality tests, it is recommended to call `gr_ctx_set_is_field(ctx, T_TRUE)` (if n is prime) `gr_ctx_set_is_field(ctx, T_FALSE)` (if n is composite) after constructing a context object. This is not done automatically as it would slow down creating a context object in the case where one is just interested in basic arithmetic.

Note: Presently, many operations for `nmod8`, `nmod32`, `nmod_redc` and `nmod_redc_fast` are not as optimized as those for the general-purpose `nmod`. It is currently recommended to use `nmod8` and `nmod32` only if one specifically wants to minimize memory usage.

6.1.2 Modular reduction and arithmetic

`void nmod_init(nmod_t *mod, ulong n)`

Initialises the given `nmod_t` structure for reduction modulo n with a precomputed inverse.

`NMOD_BITS(mod)`

Macro giving the number of bits in `mod.n`.

`NMOD_CAN_USE_SHOUP(mod)`

Macro returning whether Shoup's algorithm can be used for preconditioned multiplication mod `mod.n`.

`NMOD_RED2(r, a_hi, a_lo, mod)`

Macro to set r to a reduced modulo `mod.n`, where a consists of two limbs (`a_hi`, `a_lo`). The `mod` parameter must be a valid `nmod_t` structure. It is assumed that `a_hi` is already reduced modulo `mod.n`.

`NMOD_RED(r, a, mod)`

Macro to set r to a reduced modulo `mod.n`. The `mod` parameter must be a valid `nmod_t` structure.

`NMOD2_RED2(r, a_hi, a_lo, mod)`

Macro to set r to a reduced modulo `mod.n`, where a consists of two limbs (`a_hi`, `a_lo`). The `mod` parameter must be a valid `nmod_t` structure. No assumptions are made about `a_hi`.

`NMOD_RED3(r, a_hi, a_me, a_lo, mod)`

Macro to set r to a reduced modulo `mod.n`, where a consists of three limbs (`a_hi`, `a_me`, `a_lo`). The `mod` parameter must be a valid `nmod_t` structure. It is assumed that `a_hi` is already reduced modulo `mod.n`.

`NMOD_MUL_PRENORM(res, a, b, mod)`

Macro to set r to ab modulo `mod.n`. The `mod` parameter must be a valid `nmod_t` structure. It is assumed that a , b are already reduced modulo `mod.n` and that either a or b is prenormalised by left-shifting by `mod.norm`.

`NMOD_MUL_FULLWORD(res, a, b, mod)`

Macro to set r to ab modulo `mod.n`. The `mod` parameter must be a valid `nmod_t` structure. It is assumed that a , b are already reduced modulo `mod.n` and that `mod.n` is exactly `FLINT_BITS` bits large.

`NMOD_ADDMUL(r, a, b, mod)`

Macro to set r to $r + ab$ reduced modulo `mod.n`. The `mod` parameter must be a valid `nmod_t` structure. It is assumed that r , a , b are already reduced modulo `mod.n`.

`ulong _nmod_add(ulong a, ulong b, nmod_t mod)`

Returns $a + b$ modulo `mod.n`. It is assumed that `mod` is no more than `FLINT_BITS - 1` bits. It is assumed that a and b are already reduced modulo `mod.n`.

`ulong nmod_add(ulong a, ulong b, nmod_t mod)`

Returns $a + b$ modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a and b are already reduced modulo `mod.n`.

`ulong nmod_ui_add_ui(ulong a, ulong b, nmod_t mod)`

Returns $a + b$ modulo `mod.n`. Does not require that a and b are already reduced modulo `mod.n`.

`ulong _nmod_sub(ulong a, ulong b, nmod_t mod)`

Returns $a - b$ modulo `mod.n`. It is assumed that `mod` is no more than `FLINT_BITS - 1` bits. It is assumed that a and b are already reduced modulo `mod.n`.

`ulong nmod_sub(ulong a, ulong b, nmod_t mod)`

Returns $a - b$ modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a and b are already reduced modulo `mod.n`.

ulong **nmod_neg**(*ulong* a, *nmod_t* mod)

Returns $-a$ modulo `mod.n`. It is assumed that a is already reduced modulo `mod.n`, but no assumptions are made about the latter.

ulong **nmod_mul**(*ulong* a, *ulong* b, *nmod_t* mod)

Returns ab modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a and b are already reduced modulo `mod.n`.

ulong **_nmod_mul_fullword**(*ulong* a, *ulong* b, *nmod_t* mod)

Returns ab modulo `mod.n`. Requires that `mod.n` is exactly `FLINT_BITS` large. It is assumed that a and b are already reduced modulo `mod.n`.

ulong **nmod_ui_mul_ui**(*ulong* a, *ulong* b, *nmod_t* mod)

Returns ab modulo `mod.n`. Does not require that a and b are already reduced modulo `mod.n`.

ulong **nmod_inv**(*ulong* a, *nmod_t* mod)

Returns a^{-1} modulo `mod.n`. The inverse is assumed to exist.

ulong **nmod_div**(*ulong* a, *ulong* b, *nmod_t* mod)

Returns ab^{-1} modulo `mod.n`. The inverse of b is assumed to exist. It is assumed that a is already reduced modulo `mod.n`.

int **nmod_divides**(*ulong* *a, *ulong* b, *ulong* c, *nmod_t* mod)

If $a \cdot c = b \pmod n$ has a solution for a return 1 and set a to such a solution. Otherwise return 0 and leave a undefined.

ulong **nmod_pow_ui**(*ulong* a, *ulong* e, *nmod_t* mod)

Returns a^e modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a is already reduced modulo `mod.n`.

ulong **nmod_pow_fmpz**(*ulong* a, const *fmpz_t* e, *nmod_t* mod)

Returns a^e modulo `mod.n`. No assumptions are made about `mod.n`. It is assumed that a is already reduced modulo `mod.n` and that e is not negative.

ulong **nmod_ui_pow_ui**(*ulong* a, *ulong* e, *nmod_t* mod)

Returns a^e modulo `mod.n`. Does not require that a is already reduced modulo `mod.n`.

ulong **nmod_2_pow_ui**(*ulong* e, *nmod_t* mod)

Returns 2^e modulo `mod.n`.

6.1.3 Montgomery arithmetic

Let n be an odd integer smaller than the machine word modulus $R = 2^{32}$ or $R = 2^{64}$. The Montgomery representation of an integer x is the residue $xR \pmod n$. We can use Montgomery representations for doing arithmetic in $\mathbb{Z}/n\mathbb{Z}$, following the rules

$$(x + y)R \pmod n = xR + yR \pmod n,$$

$$(xy)R \pmod n = (xR)(yR)/R \pmod n.$$

The advantage of using the Montgomery representation instead of the standard representation $x \pmod n$ is that it allows for faster multiplication. Montgomery arithmetic is also known as REDC arithmetic.

By default moduli up to the full word size are allowed and residues are strictly canonicalised to $[0, n)$. We provide sets of alternative methods for restricted moduli which, depending on the machine, can speed up arithmetic:

- The `half` (and `half_fast`) methods work with half-length R , avoiding the need for double-word operations in the internal arithmetic. They are not necessarily faster than the non-`half` versions. Mixing `half` and non-`half` methods and context objects is not allowed.

- The `fast` (and `half_fast`) accept non-canonical residues in $[0, 2n)$ and return non-canonical residues in $[0, 2n)$. The user must convert back to the canonical range $[0, n)$ before using any non-`fast` functions. The `fast` methods are generally faster than the non-`fast` methods.

The restrictions are summarized in the following table, assuming a 64-bit word size:

Prefix	Maximum n	R	Canonicalisation
<code>redc</code>	$2^{64} - 1$	2^{64}	$[0, n)$
<code>redc_fast</code>	$2^{62} - 1$	2^{64}	$[0, 2n)$
<code>redc_half</code>	$2^{31} - 1$	2^{32}	$[0, n)$
<code>redc_half_fast</code>	$2^{30} - 1$	2^{32}	$[0, 2n)$

For 32-bit machines, the maximum n become $2^{32} - 1$, $2^{30} - 1$, $2^{15} - 1$, and $2^{14} - 1$ respectively.

type `nmod_redc_ctx_struct`

type `nmod_redc_ctx_t`

A context object for Montgomery arithmetic. This holds the same content as an `nmod_t` context, plus the precomputed constant $-1/n$ modulo R .

void `nmod_redc_ctx_init_nmod`(*nmod_redc_ctx_t* ctx, *nmod_t* mod)

void `nmod_redc_ctx_init_ui`(*nmod_redc_ctx_t* ctx, *ulong* n)

Initialize ctx for Montgomery arithmetic modulo the given modulus n which is required to be odd.

ulong `nmod_redc_set_nmod`(*ulong* x, const *nmod_redc_ctx_t* ctx)

Convert x from the standard representation to Montgomery representation. If x is viewed a residue in the standard representation, this returns $xR \bmod n$.

ulong `nmod_redc_set_ui`(*ulong* x, const *nmod_redc_ctx_t* ctx)

Convert x which is not necessarily reduced modulo n to Montgomery representation.

ulong `nmod_redc_get_nmod`(*ulong* x, const *nmod_redc_ctx_t* ctx)

Convert x back from Montgomery representation to standard representation. If x is viewed as a residue in the standard representation, this returns $x/R \bmod n$. This function allows $x \in [0, 2n)$.

ulong `nmod_redc_neg`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_add`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_sub`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_mul`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

Arithmetic in the Montgomery representation.

int `nmod_redc_can_use_fast`(const *nmod_redc_ctx_t* ctx)

Return whether the modulus is small enough to safely use fast operations.

ulong `nmod_redc_fast_normalise`(*ulong* x, const *nmod_redc_ctx_t* ctx)

Convert a non-canonical residue in $[0, 2n)$ to a canonical residue in $[0, n)$.

ulong `nmod_redc_fast_neg`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_fast_add`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_fast_sub`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_fast_mul`(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong `nmod_redc_fast_mul_two`(*ulong* x, const *nmod_redc_ctx_t* ctx)

Arithmetic in Montgomery representation, using non-canonical residues.

ulong `_nmod_redc_pow_ui`(*ulong* a, *ulong* exp, const *nmod_redc_ctx_t* ctx)

ulong `_nmod_redc_fast_pow_ui`(*ulong* a, *ulong* exp, const *nmod_redc_ctx_t* ctx)

Return a^{exp} . The exponent is required to be positive.

ulong `_nmod_redc_2_pow_ui`(*ulong* exp, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_fast_2_pow_ui**(*ulong* exp, const *nmod_redc_ctx_t* ctx)

Return 2^{exp} . There are no restrictions on the exponent.

void **nmod_redc_half_ctx_init_nmod**(*nmod_redc_ctx_t* ctx, *nmod_t* mod)

void **nmod_redc_half_ctx_init_ui**(*nmod_redc_ctx_t* ctx, *ulong* n)

Initialize context for use with `half` or `half_fast` methods.

ulong **nmod_redc_half_set_nmod**(*ulong* x, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_set_ui**(*ulong* x, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_get_nmod**(*ulong* x, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_add**(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_sub**(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_mul**(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

int **nmod_redc_half_can_use_fast**(const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_fast_mul**(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_fast_add**(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

ulong **nmod_redc_half_fast_sub**(*ulong* x, *ulong* y, const *nmod_redc_ctx_t* ctx)

Methods analogous to their non-half counterparts.

6.1.4 Discrete Logarithms via Pohlig-Hellman

void **nmod_discrete_log_pohlig_hellman_init**(*nmod_discrete_log_pohlig_hellman_t* L)

Initialize L. Upon initialization L is not ready for computation.

void **nmod_discrete_log_pohlig_hellman_clear**(*nmod_discrete_log_pohlig_hellman_t* L)

Free any space used by L.

double **nmod_discrete_log_pohlig_hellman_precompute_prime**(*nmod_discrete_log_pohlig_hellman_t* L, *ulong* p)

Configure L for discrete logarithms modulo p to an internally chosen base. It is assumed that p is prime. The return is an estimate on the number of multiplications needed for one run.

ulong **nmod_discrete_log_pohlig_hellman_primitive_root**(const *nmod_discrete_log_pohlig_hellman_t* L)

Return the internally stored base.

ulong **nmod_discrete_log_pohlig_hellman_run**(const *nmod_discrete_log_pohlig_hellman_t* L, *ulong* y)

Return the logarithm of y with respect to the internally stored base. y is expected to be reduced modulo the p. The function is undefined if the logarithm does not exist.

6.2 nmod_vec.h – vectors over integers mod n (word-size n)

6.2.1 Memory management

nn_ptr **_nmod_vec_init**(*slong* len)

Returns a vector of the given length. The entries are not necessarily zero.

void **_nmod_vec_clear**(*nn_ptr* vec)

Frees the memory used by the given vector.

6.2.2 Random functions

void `_nmod_vec_randtest`(*nn_ptr* vec, *flint_rand_t* state, *slong* len, *nmod_t* mod)

Sets `vec` to a random vector of the given length with entries reduced modulo `mod.n`.

void `_nmod_vec_rand`(*nn_ptr* vec, *flint_rand_t* state, *slong* len, *nmod_t* mod)

Sets `vec` to a vector of the given length with entries picked uniformly at random in $[0, \text{mod.n})$.

6.2.3 Basic manipulation and comparison

void `_nmod_vec_set`(*nn_ptr* res, *nn_srcptr* vec, *slong* len)

Copies `len` entries from the vector `vec` to `res`.

void `_nmod_vec_zero`(*nn_ptr* vec, *slong* len)

Zeros the given vector of the given length.

void `_nmod_vec_swap`(*nn_ptr* a, *nn_ptr* b, *slong* length)

Swaps the vectors `a` and `b` of length `n` by actually swapping the entries.

void `_nmod_vec_reduce`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *nmod_t* mod)

Reduces the entries of `(vec, len)` modulo `mod.n` and set `res` to the result.

flint_bitcnt_t `_nmod_vec_max_bits`(*nn_srcptr* vec, *slong* len)

Returns the maximum number of bits of any entry in the vector.

int `_nmod_vec_equal`(*nn_srcptr* vec, *nn_srcptr* vec2, *slong* len)

Returns 1 if `(vec, len)` is equal to `(vec2, len)`, otherwise returns 0.

6.2.4 Printing

void `_nmod_vec_print_pretty`(*nn_srcptr* vec, *slong* len, *nmod_t* mod)

Pretty-prints `vec` to `stdout`. A header is printed followed by the vector enclosed in brackets. Each entry is right-aligned to the width of the modulus written in decimal, and the entries are separated by spaces. For example:

```
<length-12 integer vector mod 197>
[ 33 181 107  61  32  11  80 138  34 171  86 156]
```

int `_nmod_vec_fprint_pretty`(FILE *file, *nn_srcptr* vec, *slong* len, *nmod_t* mod)

Same as `_nmod_vec_print_pretty` but printing to file.

int `_nmod_vec_print`(*nn_srcptr* vec, *slong* len, *nmod_t* mod)

Currently, same as `_nmod_vec_print_pretty`.

int `_nmod_vec_fprint`(FILE *f, *nn_srcptr* vec, *slong* len, *nmod_t* mod)

Currently, same as `_nmod_vec_fprint_pretty`.

6.2.5 Arithmetic operations

void `_nmod_vec_add`(*nn_ptr* res, *nn_srcptr* vec1, *nn_srcptr* vec2, *slong* len, *nmod_t* mod)
 Sets (res, len) to the sum of (vec1, len) and (vec2, len).

void `_nmod_vec_sub`(*nn_ptr* res, *nn_srcptr* vec1, *nn_srcptr* vec2, *slong* len, *nmod_t* mod)
 Sets (res, len) to the difference of (vec1, len) and (vec2, len).

void `_nmod_vec_neg`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *nmod_t* mod)
 Sets (res, len) to the negation of (vec, len).

void `_nmod_vec_invert`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *nmod_t* mod)
 Sets each entry of (res, len) to the modular inverse of the corresponding entry in (vec, len). Assumes all entries in vec are invertible modulo *mod.n*. Aliasing of res and vec is allowed.

void `_nmod_vec_scalar_mul_nmod`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *ulong* c, *nmod_t* mod)
 Sets (res, len) to (vec, len) multiplied by *c*. The element *c* and all elements of vec are assumed to be less than *mod.n*.

void `_nmod_vec_scalar_mul_nmod_shoup`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *ulong* c, *nmod_t* mod)
 Sets (res, len) to (vec, len) multiplied by *c* using `n_mulmod_shoup()`. *mod.n* should be less than $2^{\text{FLINT\_BITS}-1}$, and *c* should be less than *mod.n*. There is no constraint on elements of vec.

void `_nmod_vec_scalar_mul_nmod_redc`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *ulong* c, *nmod_t* mod)
 Sets (res, len) to (vec, len) multiplied by *c* using Montgomery multiplication. Requires that *mod.n* is odd. The element *c* and all elements of vec are assumed to be less than *mod.n*.

void `_nmod_vec_scalar_addmul_nmod`(*nn_ptr* res, *nn_srcptr* vec, *slong* len, *ulong* c, *nmod_t* mod)
 Adds (vec, len) times *c* to the vector (res, len). The element *c* and all elements of vec are assumed to be less than *mod.n*.

6.2.6 Dot products

Dot products functions and macros rely on several implementations, depending on the length of this dot product and on the underlying modulus. What implementations will be called is determined via `_nmod_vec_dot_params`, which returns a `dot_params_t` element which can then be used as input to the dot product routines.

The efficiency of the different approaches range roughly as follows, from faster to slower, on 64 bit machines. In all cases, modular reduction is only performed at the very end of the computation.

- moduli up to 1515531528 (about $2^{30.5}$): implemented via single limb integer multiplication, using explicit vectorization if supported (current support is for AVX2);
- moduli that are a power of 2 up to 2^{32} : same efficiency as the above case;
- moduli that are a power of 2 between 2^{33} and 2^{63} : efficiency between that of the above case and that of the below one (depending on the machine and on automatic vectorization);
- other moduli up to 2^{32} : implemented via single limb integer multiplication combined with accumulation in two limbs;
- moduli more than 2^{32} , unreduced dot product fits in two limbs: implemented via two limbs integer multiplication, with a final modular reduction;
- unreduced dot product fits in three limbs, moduli up to about $2^{62.5}$: implemented via two limbs integer multiplication, with intermediate accumulation of sub-products in two limbs, and overall accumulation in three limbs;
- unreduced dot product fits in three limbs, other moduli: implemented via two limbs integer multiplication, with accumulation in three limbs.

type `dot_params_t`

dot_params_t `_nmod_vec_dot_params`(*slong* len, *nmod_t* mod)

Returns a `dot_params_t` element. This element can be used as input for the dot product macros and functions that require it, for any dot product of vector with entries reduced modulo `mod.n` and whose length is less than or equal to `len`.

Internals, subject to change: its field `method` indicates the method that will be used to compute a dot product of this length `len` when working with the given `mod`. Its field `pow2_precomp` is set to `2**DOT_SPLIT_BITS % mod.n` if `method == _DOT2_SPLIT`, and to 0 otherwise.

ulong `_nmod_vec_dot`(*nn_srcptr* vec1, *nn_srcptr* vec2, *slong* len, *nmod_t* mod, *dot_params_t* params)

Returns the dot product of `(vec1, len)` and `(vec2, len)`. The input `params` has type `dot_params_t` and must have been computed via `_nmod_vec_dot_params` with the specified `mod` and with a length greater than or equal to `len`.

ulong `_nmod_vec_dot_rev`(*nn_srcptr* vec1, *nn_srcptr* vec2, *slong* len, *nmod_t* mod, *dot_params_t* params)

The same as `_nmod_vec_dot`, but reverses `vec2`.

ulong `_nmod_vec_dot_ptr`(*nn_srcptr* vec1, const *nn_ptr* *vec2, *slong* offset, *slong* len, *nmod_t* mod, *dot_params_t* params)

Returns the dot product of `(vec1, len)` and the values at `vec2[i][offset]`. The input `params` has type `dot_params_t` and must have been computed via `_nmod_vec_dot_params` with the specified `mod` and with a length greater than or equal to `len`.

`NMOD_VEC_DOT`(res, i, len, expr1, expr2, mod, params)

Effectively performs the computation:

```
res = 0;
for (i = 0; i < len; i++)
    res += (expr1) * (expr2);
```

but with the arithmetic performed modulo `mod`. The input `params` has type `dot_params_t` and must have been computed via `_nmod_vec_dot_params` with the specified `mod` and with a length greater than or equal to `len`.

`nmod.h` has to be included in order for this macro to work (order of inclusions does not matter).

int `_nmod_vec_dot_bound_limbs`(*slong* len, *nmod_t* mod)

Returns the number of limbs (0, 1, 2 or 3) needed to represent the unreduced dot product of two vectors of length `len` having entries modulo `mod.n`, assuming that `len` is nonnegative and that `mod.n` is nonzero. The computed bound is tight. In other words, this function returns the precise limb size of `len` times `(mod.n - 1)**2`.

int `_nmod_vec_dot_bound_limbs_from_params`(*slong* len, *nmod_t* mod, *dot_params_t* params)

Same specification as `_nmod_vec_dot_bound_limbs`, but uses the additional input `params` to reduce the amount of computations; for correctness `params` must have been computed for the specified `len` and `mod`.

6.3 `nmod_mat.h` – matrices over integers mod n (word-size n)

An `nmod_mat_t` represents a matrix of integers modulo n , for any non-zero modulus n that fits in a single limb, up to $2^{32} - 1$ or $2^{64} - 1$.

The `nmod_mat_t` type is defined as an array of `nmod_mat_struct`'s of length one. This permits passing parameters of type `nmod_mat_t` by reference.

An `nmod_mat_t` internally consists of a single array of `ulong`'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

It is assumed that all matrices passed to a function have the same modulus. The modulus is assumed to be a prime number in functions that perform some kind of division, solving, or Gaussian elimination (including computation of rank and determinant), but can be composite in functions that only perform basic manipulation and ring operations (e.g. transpose and matrix multiplication).

Unless indicated, functions may assume that the stored values are in the range $[0, n)$. The user can manipulate matrix entries directly, but must assume responsibility for normalising all values to the range $[0, n)$.

6.3.1 Types, macros and constants

type `nmod_mat_struct`

type `nmod_mat_t`

6.3.2 Memory management

void `nmod_mat_init`(`nmod_mat_t` mat, *slong* rows, *slong* cols, *ulong* n)

Initialises `mat` to a `rows`-by-`cols` matrix with coefficients modulo n , where n can be any nonzero integer that fits in a limb. All elements are set to zero.

void `nmod_mat_init_set`(`nmod_mat_t` mat, const `nmod_mat_t` src)

Initialises `mat` and sets its dimensions, modulus and elements to those of `src`.

void `nmod_mat_clear`(`nmod_mat_t` mat)

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `nmod_mat_t` object.

void `nmod_mat_set`(`nmod_mat_t` mat, const `nmod_mat_t` src)

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

void `nmod_mat_swap`(`nmod_mat_t` mat1, `nmod_mat_t` mat2)

Exchanges `mat1` and `mat2`.

void `nmod_mat_swap_entrywise`(`nmod_mat_t` mat1, `nmod_mat_t` mat2)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

`nmod_t` `nmod_mat_mod`(const `nmod_mat_t` mat)

Returns the modulus of a matrix.

void **nmod_mat_set_mod**(*nmod_mat_t* mat, *ulong* n)

Sets the modulus of an already initialized matrix **mat** to be n . Row and column dimensions are unchanged, and allocated memory is unaffected. Caution: this does not reduce the entries of **mat** modulo n .

6.3.3 Basic properties and manipulation

nmod_mat_entry(mat, i, j)

Directly accesses the entry in **mat** in row i and column j , indexed from zero. No bounds checking is performed. This macro can be used both for reading and writing coefficients.

ulong **nmod_mat_get_entry**(const *nmod_mat_t* mat, *slong* i, *slong* j)

Get the entry at row i and column j of the matrix **mat**.

ulong ***nmod_mat_entry_ptr**(const *nmod_mat_t* mat, *slong* i, *slong* j)

Return a pointer to the entry at row i and column j of the matrix **mat**.

void **nmod_mat_set_entry**(*nmod_mat_t* mat, *slong* i, *slong* j, *ulong* x)

Set the entry at row i and column j of the matrix **mat** to x .

slong **nmod_mat_nrows**(const *nmod_mat_t* mat)

Returns the number of rows in **mat**.

slong **nmod_mat_ncols**(const *nmod_mat_t* mat)

Returns the number of columns in **mat**.

void **nmod_mat_zero**(*nmod_mat_t* mat)

Sets all entries of the matrix **mat** to zero.

int **nmod_mat_is_zero**(const *nmod_mat_t* mat)

Returns 1 if all entries of the matrix **mat** are zero.

6.3.4 Window

void **nmod_mat_window_init**(*nmod_mat_t* window, const *nmod_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2)

Initializes the matrix **window** to be an $r2 - r1$ by $c2 - c1$ submatrix of **mat** whose (0,0) entry is the (r1, c1) entry of **mat**. The memory for the elements of **window** is shared with **mat**.

void **nmod_mat_window_clear**(*nmod_mat_t* window)

Clears the matrix **window** and releases any memory that it uses. Note that the memory of the underlying matrix that **window** points to is not freed.

6.3.5 Concatenate

void **nmod_mat_concat_vertical**(*nmod_mat_t* res, const *nmod_mat_t* mat1, const *nmod_mat_t* mat2)

Sets **res** to vertical concatenation of (mat1, mat2) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $k \times n$, **res** : $(m + k) \times n$.

void **nmod_mat_concat_horizontal**(*nmod_mat_t* res, const *nmod_mat_t* mat1, const *nmod_mat_t* mat2)

Sets **res** to horizontal concatenation of (mat1, mat2) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $m \times k$, **res** : $m \times (n + k)$.

6.3.6 Printing

void **nmod_mat_print_pretty**(const *nmod_mat_t* mat)

Pretty-prints **mat** to **stdout**. A header is printed followed by the rows enclosed in brackets. Each column is right-aligned to the width of the modulus written in decimal, and the columns are separated by spaces. For example:

```
<2 x 3 integer matrix mod 2903>
[  0   0 2607]
[ 622   0   0]
```

int **nmod_mat_fprint_pretty**(FILE *file, const *nmod_mat_t* mat)

Same as **nmod_mat_print_pretty** but printing to file.

int **nmod_mat_print**(const *nmod_mat_t* mat)

Currently, same as **nmod_mat_print_pretty**.

int **nmod_mat_fprint**(FILE *f, const *nmod_mat_t* mat)

Currently, same as **nmod_mat_fprint_pretty**.

6.3.7 Random matrix generation

void **nmod_mat_randtest**(*nmod_mat_t* mat, *flint_rand_t* state)

Sets the elements to a random matrix with entries between 0 and $m - 1$ inclusive, where m is the modulus of **mat**. A sparse matrix is generated with increased probability.

void **nmod_mat_randfull**(*nmod_mat_t* mat, *flint_rand_t* state)

Sets the element to random numbers in $[0, n)$, likely to be close to the modulus n of the matrix. This is used to test potential overflow-related bugs.

void **nmod_mat_rand**(*nmod_mat_t* mat, *flint_rand_t* state)

Sets the element to uniformly generated random numbers in $[0, n)$, where n is the modulus of the matrix.

int **nmod_mat_randpermdiag**(*nmod_mat_t* mat, *flint_rand_t* state, *nn_srcptr* diag, *slong* n)

Sets **mat** to a random permutation of the diagonal matrix with n leading entries given by the vector **diag**. It is assumed that the main diagonal of **mat** has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

void **nmod_mat_randrank**(*nmod_mat_t* mat, *flint_rand_t* state, *slong* rank)

Sets **mat** to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random integers between 0 and $m - 1$ inclusive, where m is the modulus of **mat**.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling **nmod_mat_randops()**.

void **nmod_mat_randops**(*nmod_mat_t* mat, *flint_rand_t* state, *slong* count)

Randomises **mat** by performing elementary row or column operations. More precisely, at most **count** random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void **nmod_mat_randtril**(*nmod_mat_t* mat, *flint_rand_t* state, int unit)

Sets **mat** to a random lower triangular matrix. If **unit** is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

void **nmod_mat_randtriu**(*nmod_mat_t* mat, *flint_rand_t* state, int unit)

Sets **mat** to a random upper triangular matrix. If **unit** is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

6.3.8 Comparison

int **nmod_mat_equal**(const *nmod_mat_t* mat1, const *nmod_mat_t* mat2)

Returns nonzero if **mat1** and **mat2** have the same dimensions and elements, and zero otherwise. The moduli are ignored.

int **nmod_mat_is_zero_row**(const *nmod_mat_t* mat, *slong* i)

Returns a non-zero value if row *i* of **mat** is zero.

6.3.9 Transposition and permutations

void **nmod_mat_transpose**(*nmod_mat_t* B, const *nmod_mat_t* A)

Sets *B* to the transpose of *A*. Dimensions must be compatible. Aliasing is allowed for square matrices.

void **nmod_mat_swap_rows**(*nmod_mat_t* mat, *slong* *perm, *slong* r, *slong* s)

Swaps rows *r* and *s* of **mat**. If **perm** is non-NULL, the permutation of the rows will also be applied to **perm**.

void **nmod_mat_swap_cols**(*nmod_mat_t* mat, *slong* *perm, *slong* r, *slong* s)

Swaps columns *r* and *s* of **mat**. If **perm** is non-NULL, the permutation of the columns will also be applied to **perm**.

void **nmod_mat_invert_rows**(*nmod_mat_t* mat, *slong* *perm)

Swaps rows *i* and *r - i* of **mat** for $0 \leq i < r/2$, where *r* is the number of rows of **mat**. If **perm** is non-NULL, the permutation of the rows will also be applied to **perm**.

void **nmod_mat_invert_cols**(*nmod_mat_t* mat, *slong* *perm)

Swaps columns *i* and *c - i* of **mat** for $0 \leq i < c/2$, where *c* is the number of columns of **mat**. If **perm** is non-NULL, the permutation of the columns will also be applied to **perm**.

void **nmod_mat_permute_rows**(*nmod_mat_t* mat, const *slong* *perm_act, *slong* *perm_store)

Permutates rows of the matrix **mat** according to permutation **perm_act** and, if **perm_store** is not NULL, apply the same permutation to it.

6.3.10 Addition and subtraction

void **nmod_mat_add**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)

Computes $C = A + B$. Dimensions must be identical.

void **nmod_mat_sub**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)

Computes $C = A - B$. Dimensions must be identical.

void **nmod_mat_neg**(*nmod_mat_t* B, const *nmod_mat_t* A)

Sets $B = -A$. Dimensions must be identical.

6.3.11 Matrix-scalar arithmetic

void **nmod_mat_scalar_mul**(*nmod_mat_t* B, const *nmod_mat_t* A, *ulong* c)

Sets $B = cA$, where the scalar *c* is assumed to be reduced modulo the modulus. Dimensions of *A* and *B* must be identical.

void **nmod_mat_scalar_addmul_ui**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B, const *ulong* c)

Sets $C = A + cB$, where the scalar *c* is assumed to be reduced modulo the modulus. Dimensions of *C*, *A* and *B* must be identical. *C* can be aliased with *A* or *B*.

void **nmod_mat_scalar_mul_fmpz**(*nmod_mat_t* B, const *nmod_mat_t* A, const *fmpz_t* c)
 Sets $B = cA$, where the scalar c is of type *fmpz_t*. Dimensions of A and B must be identical.

6.3.12 Matrix multiplication

void **nmod_mat_mul**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and Strassen multiplication.

void **_nmod_mat_mul_classical_op**(*nmod_mat_t* D, const *nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B, int op)
 Sets $D = A * B \text{ op } C$ where op is +1 for addition, -1 for subtraction and 0 to ignore C.

void **nmod_mat_mul_classical**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses classical matrix multiplication, creating a temporary transposed copy of B to improve memory locality if the matrices are large enough, and packing several entries of B into each word if the modulus is very small.

void **_nmod_mat_mul_classical_threaded_pool_op**(*nmod_mat_t* D, const *nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B, int op, *thread_pool_handle* *threads, *slong* num_threads)
 Multithreaded version of **_nmod_mat_mul_classical**.

void **_nmod_mat_mul_classical_threaded_op**(*nmod_mat_t* D, const *nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B, int op)
 Multithreaded version of **_nmod_mat_mul_classical**.

void **nmod_mat_mul_classical_threaded**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Multithreaded version of **nmod_mat_mul_classical**.

void **nmod_mat_mul_strassen**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses Strassen multiplication (the Strassen-Winograd variant).

int **nmod_mat_mul_blas**(*nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Tries to set $C = AB$ using BLAS and returns 1 for success and 0 for failure. Dimensions must be compatible for matrix multiplication.

void **nmod_mat_addmul**(*nmod_mat_t* D, const *nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Sets $D = C + AB$. C and D may be aliased with each other but not with A or B . Automatically selects between classical and Strassen multiplication.

void **nmod_mat_submul**(*nmod_mat_t* D, const *nmod_mat_t* C, const *nmod_mat_t* A, const *nmod_mat_t* B)
 Sets $D = C - AB$. C and D may be aliased with each other but not with A or B .

void **nmod_mat_mul_nmod_vec**(*ulong* *c, const *nmod_mat_t* A, const *ulong* *b, *slong* blen)

void **nmod_mat_mul_nmod_vec_ptr**(*ulong* *const *c, const *nmod_mat_t* A, const *ulong* *const *b, *slong* blen)

Compute a matrix-vector product of A and (b, blen) and store the result in c . The vector (b, blen) is either truncated or zero-extended to the number of columns of A . The number entries written to c is always equal to the number of rows of A .

void **nmod_mat_nmod_vec_mul**(*ulong* *c, const *ulong* *a, *slong* alen, const *nmod_mat_t* B)

```
void nmod_mat_nmod_vec_mul_ptr(ulong *const *c, const ulong *const *a, slong alen, const
                               nmod_mat_t B)
```

Compute a vector-matrix product of $(a, alen)$ and B and store the result in c . The vector $(a, alen)$ is either truncated or zero-extended to the number of rows of B . The number entries written to c is always equal to the number of columns of B .

6.3.13 Matrix Exponentiation

```
void _nmod_mat_pow(nmod_mat_t dest, const nmod_mat_t mat, ulong pow)
```

Sets $dest = mat^{pow}$. **dest** and **mat** cannot be aliased. Implements exponentiation by squaring.

```
void nmod_mat_pow(nmod_mat_t dest, const nmod_mat_t mat, ulong pow)
```

Sets $dest = mat^{pow}$. **dest** and **mat** may be aliased. Implements exponentiation by squaring.

6.3.14 Trace

```
ulong nmod_mat_trace(const nmod_mat_t mat)
```

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

6.3.15 Determinant and rank

```
ulong nmod_mat_det_howell(const nmod_mat_t A)
```

Returns the determinant of A .

```
ulong nmod_mat_det(const nmod_mat_t A)
```

Returns the determinant of A .

```
slong nmod_mat_rank(const nmod_mat_t A)
```

Returns the rank of A . The modulus of A must be a prime number.

6.3.16 Inverse

```
int nmod_mat_inv(nmod_mat_t B, const nmod_mat_t A)
```

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of B to undefined values.

A and B must be square matrices with the same dimensions and modulus. The modulus must be prime.

6.3.17 Triangular solving

```
void nmod_mat_solve_tril(nmod_mat_t X, const nmod_mat_t L, const nmod_mat_t B, int unit)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If **unit** = 1, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void nmod_mat_solve_tril_classical(nmod_mat_t X, const nmod_mat_t L, const nmod_mat_t
                                B, int unit)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit = 1`, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void nmod_mat_solve_tril_recursive(nmod_mat_t X, const nmod_mat_t L, const nmod_mat_t
                                  B, int unit)
```

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit = 1`, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}X \\ D^{-1}(Y - CA^{-1}X) \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

```
void nmod_mat_solve_triu(nmod_mat_t X, const nmod_mat_t U, const nmod_mat_t B, int unit)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit = 1`, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

```
void nmod_mat_solve_triu_classical(nmod_mat_t X, const nmod_mat_t U, const nmod_mat_t
                                   B, int unit)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit = 1`, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Uses forward substitution.

```
void nmod_mat_solve_triu_recursive(nmod_mat_t X, const nmod_mat_t U, const nmod_mat_t
                                   B, int unit)
```

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit = 1`, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed.

Uses the block inversion formula

$$\begin{pmatrix} A & B \\ 0 & D \end{pmatrix}^{-1} \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} A^{-1}(X - BD^{-1}Y) \\ D^{-1}Y \end{pmatrix}$$

to reduce the problem to matrix multiplication and triangular solving of smaller systems.

6.3.18 Nonsingular square solving

```
int nmod_mat_solve(nmod_mat_t X, const nmod_mat_t A, const nmod_mat_t B)
```

Solves the matrix-matrix equation $AX = B$ over $\mathbb{Z}/p\mathbb{Z}$ where p is the modulus of X which must be a prime number. X , A , and B should have the same moduli.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

The matrix A must be square.

```
int nmod_mat_can_solve_inner(slong *rank, slong *perm, slong *pivots, nmod_mat_t X, const
                             nmod_mat_t A, const nmod_mat_t B)
```

As for `nmod_mat_solve()` except that if `rank` is not `NULL` the value it points to will be set to the rank of A . If `perm` is not `NULL` then it must be a valid initialised permutation whose length is the number of rows of A . After the function call it will be set to the row permutation given by LU decomposition of A . If `pivots` is not `NULL` then it must be an initialised vector. Only the first `*rank` of these will be set by the function call. They are set to the columns of the pivots chosen by the LU decomposition of A .

int **nmod_mat_can_solve**(*nmod_mat_t* X, const *nmod_mat_t* A, const *nmod_mat_t* B)

Solves the matrix-matrix equation $AX = B$ over $\mathbb{Z}/p\mathbb{Z}$ where p is the modulus of X which must be a prime number. X , A , and B should have the same moduli.

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of X to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of A and it may be singular.

int **nmod_mat_solve_vec**(*nn_ptr* x, const *nmod_mat_t* A, *nn_srcptr* b)

Solves the matrix-vector equation $Ax = b$ over $\mathbb{Z}/p\mathbb{Z}$ where p is the modulus of A which must be a prime number.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of x to undefined values.

6.3.19 LU decomposition

slong **nmod_mat_lu**(*slong* *P, *nmod_mat_t* A, int rank_check)

slong **nmod_mat_lu_classical**(*slong* *P, *nmod_mat_t* A, int rank_check)

slong **nmod_mat_lu_classical_delayed**(*slong* *P, *nmod_mat_t* A, int rank_check)

slong **nmod_mat_lu_recursive**(*slong* *P, *nmod_mat_t* A, int rank_check)

Computes a generalised LU decomposition $PLU = A$ of a given matrix A , returning the rank of A .

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for **rank_check** is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

The *classical* version uses direct Gaussian elimination. The *classical_delayed* version also uses Gaussian elimination, but performs delayed modular reductions. The *recursive* version uses block recursive decomposition. The default function chooses an algorithm automatically.

6.3.20 Reduced row echelon form

slong **nmod_mat_rref**(*nmod_mat_t* A)

Puts A in reduced row echelon form and returns the rank of A .

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

slong **nmod_mat_reduce_row**(*nmod_mat_t* A, *slong* *P, *slong* *L, *slong* n)

Reduce row n of the matrix A , assuming the prior rows are in Gauss form. However those rows may not be in order. The entry i of the array P is the row of A which has a pivot in the i -th column. If no such row exists, the entry of P will be -1 . The function returns the column in which the n -th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in P . Entry i of the array L contains the number of possibly nonzero columns of A row i . This speeds up reduction in the case that A is chambered on the right. Otherwise the entries of L can all be set to the number of columns of A . We require the entries of L to be monotonic increasing.

6.3.21 Nullspace

slong **nmod_mat_nullspace**(*nmod_mat_t* X, const *nmod_mat_t* A)

Computes the nullspace of A and returns the nullity.

More precisely, this function sets X to a maximum rank matrix such that $AX = 0$ and returns the rank of X . The columns of X will form a basis for the nullspace of A .

X must have sufficient space to store all basis vectors in the nullspace.

This function computes the reduced row echelon form and then reads off the basis vectors.

6.3.22 Transforms

void **nmod_mat_similarity**(*nmod_mat_t* M, *slong* r, *ulong* d)

Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value d is required to be reduced modulo the modulus of the entries in the matrix.

6.3.23 Characteristic polynomial

void **nmod_mat_charpoly_berkowitz**(*nmod_poly_t* p, const *nmod_mat_t* M)

void **nmod_mat_charpoly_danilevsky**(*nmod_poly_t* p, const *nmod_mat_t* M)

void **nmod_mat_charpoly**(*nmod_poly_t* p, const *nmod_mat_t* M)

Compute the characteristic polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised. The *danilevsky* algorithm assumes that the modulus is prime.

6.3.24 Minimal polynomial

void **nmod_mat_minpoly**(*nmod_poly_t* p, const *nmod_mat_t* M)

Compute the minimal polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

6.3.25 Strong echelon form and Howell form

void **nmod_mat_strong_echelon_form**(*nmod_mat_t* A)

Puts A into strong echelon form. The Howell form and the strong echelon form are equal up to permutation of the rows, see [FieHof2014] for a definition of the strong echelon form and the algorithm used here. Note that [FieHof2014] defines strong echelon form as a lower left normal form, while the implemented version returns an upper right normal form, agreeing with the definition of Howell form in [StoMul1998].

A must have at least as many rows as columns.

slong **nmod_mat_howell_form**(*nmod_mat_t* A)

Puts A into Howell form and returns the number of non-zero rows. For a definition of the Howell form see [StoMul1998]. The Howell form is computed by first putting A into strong echelon form and then ordering the rows.

A must have at least as many rows as columns.

6.4 `nmod_poly.h` – univariate polynomials over integers mod n (word-size n)

The `nmod_poly_t` data type represents elements of $\mathbb{Z}/n\mathbb{Z}[x]$ for a fixed modulus n . The `nmod_poly` module provides routines for memory management, basic arithmetic and some higher level functions such as GCD, etc.

Each coefficient of an `nmod_poly_t` is of type `ulong` and represents an integer reduced modulo the fixed modulus n .

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

The `nmod_poly_t` type is a typedef for an array of length 1 of `nmod_poly_struct`'s. This permits passing parameters of type `nmod_poly_t` by reference.

In reality one never deals directly with the `struct` and simply deals with objects of type `nmod_poly_t`. For simplicity we will think of an `nmod_poly_t` as a `struct`, though in practice to access fields of this `struct`, one needs to dereference first, e.g., to access the `length` field of an `nmod_poly_t` called `poly1` one writes `poly1->length`.

An `nmod_poly_t` is said to be *normalised* if either `length` is zero, or if the leading coefficient of the polynomial is non-zero. All `nmod_poly` functions expect their inputs to be normalised and for all coefficients to be reduced modulo n and unless otherwise specified they produce output that is normalised with coefficients reduced modulo n .

It is recommended that users do not access the fields of an `nmod_poly_t` or its coefficient data directly, but make use of the functions designed for this purpose, detailed below.

Functions in `nmod_poly` do all the memory management for the user. One does not need to specify the maximum length in advance before using a polynomial object. FLINT reallocates space automatically as the computation proceeds, if more space is required.

6.4.1 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbb{Z}/7\mathbb{Z}[x]$.

```
#include "nmod_poly.h"
int main()
{
    nmod_poly_t x, y;
    nmod_poly_init(x, 7);
    nmod_poly_init(y, 7);
    nmod_poly_set_coeff_ui(x, 3, 5);
    nmod_poly_set_coeff_ui(x, 0, 6);
    nmod_poly_mul(y, x, x);
    nmod_poly_print(x); flint_printf("\n");
    nmod_poly_print(y); flint_printf("\n");
    nmod_poly_clear(x);
    nmod_poly_clear(y);
}
```

The output is:

```
4 7 6 0 0 5
7 7 1 0 0 4 0 0 4
```

6.4.2 Types, macros and constants

type `nmod_poly_struct`

type `nmod_poly_t`

6.4.3 Memory management

void `nmod_poly_init`(*nmod_poly_t* poly, *ulong* n)

Initialises `poly`. It will have coefficients modulo n .

void `nmod_poly_init_preinv`(*nmod_poly_t* poly, *ulong* n, *ulong* ninv)

Initialises `poly`. It will have coefficients modulo n . The caller supplies a precomputed inverse limb generated by `n_preinvert_limb()`.

void `nmod_poly_init_mod`(*nmod_poly_t* poly, const *nmod_t* mod)

Initialises `poly` using an already initialised modulus `mod`.

void `nmod_poly_init2`(*nmod_poly_t* poly, *ulong* n, *slong* alloc)

Initialises `poly`. It will have coefficients modulo n . Up to `alloc` coefficients may be stored in `poly`.

void `nmod_poly_init2_preinv`(*nmod_poly_t* poly, *ulong* n, *ulong* ninv, *slong* alloc)

Initialises `poly`. It will have coefficients modulo n . The caller supplies a precomputed inverse limb generated by `n_preinvert_limb()`. Up to `alloc` coefficients may be stored in `poly`.

void `nmod_poly_realloc`(*nmod_poly_t* poly, *slong* alloc)

Reallocates `poly` to the given length. If the current length is less than `alloc`, the polynomial is truncated and normalised. If `alloc` is zero, the polynomial is cleared.

void `nmod_poly_clear`(*nmod_poly_t* poly)

Clears the polynomial and releases any memory it used. The polynomial cannot be used again until it is initialised.

void `nmod_poly_fit_length`(*nmod_poly_t* poly, *slong* alloc)

Ensures `poly` has space for at least `alloc` coefficients. This function only ever grows the allocated space, so no data loss can occur.

void `_nmod_poly_normalise`(*nmod_poly_t* poly)

Internal function for normalising a polynomial so that the top coefficient, if there is one at all, is not zero.

6.4.4 Polynomial properties

slong `nmod_poly_length`(const *nmod_poly_t* poly)

Returns the length of the polynomial `poly`. The zero polynomial has length zero.

slong `nmod_poly_degree`(const *nmod_poly_t* poly)

Returns the degree of the polynomial `poly`. The zero polynomial is deemed to have degree -1 .

ulong `nmod_poly_modulus`(const *nmod_poly_t* poly)

Returns the modulus of the polynomial `poly`. This will be a positive integer.

flint_bitcnt_t `nmod_poly_max_bits`(const *nmod_poly_t* poly)

Returns the maximum number of bits of any coefficient of `poly`.

int `nmod_poly_is_unit`(const *nmod_poly_t* poly)

Returns 1 if the polynomial is a nonzero constant (in the case of prime modulus, this is equivalent to being a unit), otherwise 0.

int **nmod_poly_is_monic**(const *nmod_poly_t* poly)

Returns 1 if the polynomial is monic, i.e. nonzero with leading coefficient 1, otherwise 0.

6.4.5 Assignment and basic manipulation

void **nmod_poly_set**(*nmod_poly_t* a, const *nmod_poly_t* b)

Sets **a** to a copy of **b**.

void **nmod_poly_swap**(*nmod_poly_t* poly1, *nmod_poly_t* poly2)

Efficiently swaps **poly1** and **poly2** by swapping pointers internally.

void **nmod_poly_zero**(*nmod_poly_t* res)

Sets **res** to the zero polynomial.

void **nmod_poly_truncate**(*nmod_poly_t* poly, *slong* len)

Truncates **poly** to the given length and normalises it. If **len** is greater than the current length of **poly**, then nothing happens.

void **nmod_poly_set_trunc**(*nmod_poly_t* res, const *nmod_poly_t* poly, *slong* len)

Notionally truncate **poly** to length **len** and set **res** to the result. The result is normalised.

void **_nmod_poly_reverse**(*nn_ptr* output, *nn_srcptr* input, *slong* len, *slong* m)

Sets **output** to the reverse of **input**, which is of length **len**, but thinking of it as a polynomial of length **m**, notionally zero-padded if necessary. The length **m** must be non-negative, but there are no other restrictions. The polynomial **output** must have space for **m** coefficients. Supports aliasing of **output** and **input**, but the behaviour is undefined in case of partial overlap.

void **nmod_poly_reverse**(*nmod_poly_t* output, const *nmod_poly_t* input, *slong* m)

Sets **output** to the reverse of **input**, thinking of it as a polynomial of length **m**, notionally zero-padded if necessary). The length **m** must be non-negative, but there are no other restrictions. The output polynomial will be set to length **m** and then normalised.

6.4.6 Randomization

void **nmod_poly_randtest**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random polynomial with length up to **len**.

void **nmod_poly_randtest_monic**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random monic polynomial with length **len**.

void **nmod_poly_randtest_trinomial**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random monic trinomial of length **len**.

void **nmod_poly_randtest_pentomial**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random monic pentomial of length **len**.

6.4.7 Construction of irreducible polynomials

The following functions assume a prime modulus.

void **nmod_poly_minimal_irreducible**(*nmod_poly_t* res, *ulong* n)

Generates a monic irreducible polynomial of degree **n** with minimal weight (minimal number of nonzero terms). We generate a binomial if possible, otherwise a trinomial, etc. It is conjectured that one never needs more than a pentanomial modulo $p = 2$ and a tetranomial modulo $p > 2$.

More specifically, this function returns the first among all minimal-weight polynomials in the following ordering. Firstly, for trinomials, $x^n + ax^k + b$ comes before its monic reversal $x^n + a'x^{n-k} + b'$ if $k < n - k$. Secondly, writing $f = x^n + a_1x^{k_1} + a_2x^{k_2} + \dots + a_tx^{k_t}$ with $n > k_1 > k_2 > \dots > k_t$,

we order tuples $(a_1, \dots, a_t, k_1, \dots, k_t)$ lexicographically. We thus favor polynomials with smaller coefficients (all 1 if possible), and secondly with smaller degrees for the middle terms.

void **nmod_poly_randtest_irreducible**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random irreducible polynomial with length up to *len*.

void **nmod_poly_randtest_monic_irreducible**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random monic irreducible polynomial with length *len*.

void **nmod_poly_randtest_monic_primitive**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Generates a random monic irreducible primitive polynomial with length *len*.

int **nmod_poly_randtest_trinomial_irreducible**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len, *slong* max_attempts)

Attempts to set *poly* to a monic irreducible trinomial of length *len*. It will generate up to *max_attempts* trinomials in attempt to find an irreducible one. If *max_attempts* is 0, then it will keep generating trinomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

int **nmod_poly_randtest_pentomial_irreducible**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len, *slong* max_attempts)

Attempts to set *poly* to a monic irreducible pentomial of length *len*. It will generate up to *max_attempts* pentomials in attempt to find an irreducible one. If *max_attempts* is 0, then it will keep generating pentomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

void **nmod_poly_randtest_sparse_irreducible**(*nmod_poly_t* poly, *flint_rand_t* state, *slong* len)

Attempts to set *poly* to a sparse, monic irreducible polynomial with length *len*. It attempts to find an irreducible trinomial. If that does not succeed, it attempts to find an irreducible pentomial. If that fails, then *poly* is just set to a random monic irreducible polynomial.

6.4.8 Getting and setting coefficients

ulong **nmod_poly_get_coeff_ui**(const *nmod_poly_t* poly, *slong* j)

Returns the coefficient of *poly* at index *j*, where coefficients are numbered with zero being the constant coefficient, and returns it as an *ulong*. If *j* refers to a coefficient beyond the end of *poly*, zero is returned.

void **nmod_poly_set_coeff_ui**(*nmod_poly_t* poly, *slong* j, *ulong* c)

Sets the coefficient of *poly* at index *j*, where coefficients are numbered with zero being the constant coefficient, to the value *c* reduced modulo the modulus of *poly*. If *j* refers to a coefficient beyond the current end of *poly*, the polynomial is first resized, with intervening coefficients being set to zero.

6.4.9 Input and output

char ***nmod_poly_get_str**(const *nmod_poly_t* poly)

Writes *poly* to a string representation. The format is as described for *nmod_poly_print()*. The string must be freed by the user when finished. For this it is sufficient to call *flint_free()*.

char ***nmod_poly_get_str_pretty**(const *nmod_poly_t* poly, const char *x)

Writes *poly* to a pretty string representation. The format is as described for *nmod_poly_print_pretty()*. The string must be freed by the user when finished. For this it is sufficient to call *flint_free()*.

It is assumed that the top coefficient is non-zero.

int **`nmod_poly_set_str`**(*nmod_poly_t* poly, const char *s)

Reads *poly* from a string *s*. The format is as described for *nmod_poly_print()*. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

int **`nmod_poly_print`**(const *nmod_poly_t* a)

Prints the polynomial to `stdout`. The length is printed, followed by a space, then the modulus. If the length is zero this is all that is printed, otherwise two spaces followed by a space separated list of coefficients is printed, beginning with the constant coefficient.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **`nmod_poly_print_pretty`**(const *nmod_poly_t* a, const char *x)

Prints the polynomial to `stdout` using the string *x* to represent the indeterminate.

It is assumed that the top coefficient is non-zero.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **`nmod_poly_fread`**(FILE *f, *nmod_poly_t* poly)

Reads *poly* from the file stream *f*. If this is a file that has just been written, the file should be closed then opened again. The format is as described for *nmod_poly_print()*. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

int **`nmod_poly_fprint`**(FILE *f, const *nmod_poly_t* poly)

Writes a polynomial to the file stream *f*. If this is a file then the file should be closed and reopened before being read. The format is as described for *nmod_poly_print()*. If the polynomial is written correctly, a positive value is returned, otherwise a non-positive value is returned.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **`nmod_poly_fprint_pretty`**(FILE *f, const *nmod_poly_t* poly, const char *x)

Writes a polynomial to the file stream *f*. If this is a file then the file should be closed and reopened before being read. The format is as described for *nmod_poly_print_pretty()*. If the polynomial is written correctly, a positive value is returned, otherwise a non-positive value is returned.

It is assumed that the top coefficient is non-zero.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **`nmod_poly_read`**(*nmod_poly_t* poly)

Read *poly* from `stdin`. The format is as described for *nmod_poly_print()*. If a polynomial in the correct format is read, a positive value is returned, otherwise a non-positive value is returned.

6.4.10 Comparison

int **`nmod_poly_equal`**(const *nmod_poly_t* a, const *nmod_poly_t* b)

Returns 1 if the polynomials are equal, otherwise 0.

int **`nmod_poly_equal_nmod`**(const *nmod_poly_t* poly, *ulong* cst)

Returns 1 if the polynomial *poly* is constant, equal to *cst*, otherwise 0. *cst* is assumed to be already reduced, i.e. less than the modulus of *poly*.

int **`nmod_poly_equal_ui`**(const *nmod_poly_t* poly, *ulong* cst)

Returns 1 if the polynomial *poly* is constant and equal to *cst* up to reduction modulo the modulus of *poly*, otherwise returns 0.

int **`nmod_poly_equal_trunc`**(const *nmod_poly_t* poly1, const *nmod_poly_t* poly2, *slong* n)

Notionally truncate *poly1* and *poly2* to length *n* and return 1 if the truncations are equal, otherwise return 0.

int **`nmod_poly_is_zero`**(const *nmod_poly_t* poly)

Returns 1 if the polynomial *poly* is the zero polynomial, otherwise returns 0.

int **nmod_poly_is_one**(const *nmod_poly_t* poly)

Returns 1 if the polynomial *poly* is the constant polynomial 1, otherwise returns 0.

int **nmod_poly_is_gen**(const *nmod_poly_t* poly)

Returns 1 if the polynomial is the generating indeterminate (i.e. has degree 1, constant coefficient 0, and leading coefficient 1), otherwise returns 0.

6.4.11 Shifting

void **_nmod_poly_shift_left**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *slong* k)

Sets (res, len + k) to (poly, len) shifted left by k coefficients. Assumes that *res* has space for len + k coefficients.

void **nmod_poly_shift_left**(*nmod_poly_t* res, const *nmod_poly_t* poly, *slong* k)

Sets *res* to *poly* shifted left by k coefficients, i.e. multiplied by x^k .

void **_nmod_poly_shift_right**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *slong* k)

Sets (res, len - k) to (poly, len) shifted left by k coefficients. It is assumed that $k \leq \text{len}$ and that *res* has space for at least len - k coefficients.

void **nmod_poly_shift_right**(*nmod_poly_t* res, const *nmod_poly_t* poly, *slong* k)

Sets *res* to *poly* shifted right by k coefficients, i.e. divide by x^k and throw away the remainder. If k is greater than or equal to the length of *poly*, the result is the zero polynomial.

6.4.12 Addition and subtraction

void **_nmod_poly_add**(*nn_ptr* res, *nn_srcptr* poly1, *slong* len1, *nn_srcptr* poly2, *slong* len2, *nmod_t* mod)

Sets *res* to the sum of (poly1, len1) and (poly2, len2). There are no restrictions on the lengths.

void **nmod_poly_add**(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2)

Sets *res* to the sum of *poly1* and *poly2*.

void **nmod_poly_add_series**(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2, *slong* n)

Notionally truncate *poly1* and *poly2* to length *n* and set *res* to the sum.

void **_nmod_poly_sub**(*nn_ptr* res, *nn_srcptr* poly1, *slong* len1, *nn_srcptr* poly2, *slong* len2, *nmod_t* mod)

Sets *res* to the difference of (poly1, len1) and (poly2, len2). There are no restrictions on the lengths.

void **nmod_poly_sub**(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2)

Sets *res* to the difference of *poly1* and *poly2*.

void **nmod_poly_sub_series**(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2, *slong* n)

Notionally truncate *poly1* and *poly2* to length *n* and set *res* to the difference.

void **nmod_poly_neg**(*nmod_poly_t* res, const *nmod_poly_t* poly)

Sets *res* to the negation of *poly*.

6.4.13 Scalar multiplication and division

void `nmod_poly_scalar_mul_nmod`(*nmod_poly_t* res, const *nmod_poly_t* poly, *ulong* c)

Sets `res` to `poly` multiplied by `c`. The element `c` is assumed to be less than the modulus of `poly`.

void `nmod_poly_scalar_addmul_nmod`(*nmod_poly_t* res, const *nmod_poly_t* poly, *ulong* c)

Adds `poly` multiplied by `c` to `res`. The element `c` is assumed to be less than the modulus of `poly`.

void `_nmod_poly_make_monic`(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *nmod_t* mod)

Requires that `res` and `poly` have length at least `len`, with `len > 0`, and that `poly[len-1]` is invertible modulo `mod.n`. Sets `res[i]` to the modular product of `c` and `poly[i]` for `i` from 0 to `len-1`, where `c` is the inverse of `poly[len-1]`.

void `nmod_poly_make_monic`(*nmod_poly_t* res, const *nmod_poly_t* poly)

Sets `res` to be the scalar multiple of `poly` with leading coefficient one. If `poly` is zero, an exception is raised.

6.4.14 Bit packing and unpacking

void `_nmod_poly_bit_pack`(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *flint_bitcnt_t* bits)

Packs `len` coefficients of `poly` into fields of the given number of bits in the large integer `res`, i.e. evaluates `poly` at 2^{bits} and store the result in `res`. Assumes `len > 0` and `bits > 0`. Also assumes that no coefficient of `poly` is bigger than `bits/2` bits. We also assume `bits < 3 * FLINT_BITS`.

void `_nmod_poly_bit_unpack`(*nn_ptr* res, *slong* nlo, *slong* nhi, *nn_srcptr* mpn, *ulong* bits, *nmod_t* mod)

Unpacks `nhi - nlo` coefficients stored in the big integer `mpn` in bit fields of the given number of bits, starting at offset `nlo`, reduces them modulo the given modulus, then stores them in the polynomial `res`. We assume $3 * FLINT\_BITS > \text{bits} > 0$. There are no restrictions on the size of the actual coefficients as stored within the bitfields.

void `nmod_poly_bit_pack`(*fmpz_t* f, const *nmod_poly_t* poly, *flint_bitcnt_t* bit_size)

Packs `poly` into bitfields of size `bit_size`, writing the result to `f`.

void `nmod_poly_bit_unpack`(*nmod_poly_t* poly, const *fmpz_t* f, *flint_bitcnt_t* bit_size)

Unpacks the polynomial from fields of size `bit_size` as represented by the integer `f`.

void `_nmod_poly_KS2_pack1`(*nn_ptr* res, *nn_srcptr* op, *slong* n, *slong* s, *ulong* b, *ulong* k, *slong* r)

Same as `_nmod_poly_KS2_pack`, but requires `b <= FLINT_BITS`.

void `_nmod_poly_KS2_pack`(*nn_ptr* res, *nn_srcptr* op, *slong* n, *slong* s, *ulong* b, *ulong* k, *slong* r)

Bit packing routine used by KS2 and KS4 multiplication.

void `_nmod_poly_KS2_unpack1`(*nn_ptr* res, *nn_srcptr* op, *slong* n, *ulong* b, *ulong* k)

Same as `_nmod_poly_KS2_unpack`, but requires `b <= FLINT_BITS` (i.e. writes one word per coefficient).

void `_nmod_poly_KS2_unpack2`(*nn_ptr* res, *nn_srcptr* op, *slong* n, *ulong* b, *ulong* k)

Same as `_nmod_poly_KS2_unpack`, but requires $FLINT\_BITS < b <= 2 * FLINT\_BITS$ (i.e. writes two words per coefficient).

void `_nmod_poly_KS2_unpack3`(*nn_ptr* res, *nn_srcptr* op, *slong* n, *ulong* b, *ulong* k)

Same as `_nmod_poly_KS2_unpack`, but requires $2 * FLINT\_BITS < b < 3 * FLINT\_BITS$ (i.e. writes three words per coefficient).

void `_nmod_poly_KS2_unpack`(*nn_ptr* res, *nn_srcptr* op, *slong* n, *ulong* b, *ulong* k)

Bit unpacking code used by KS2 and KS4 multiplication.

6.4.15 KS2/KS4 Reduction

void `_nmod_poly_KS2_reduce`(*nn_ptr* res, *slong* s, *nn_srcptr* op, *slong* n, *ulong* w, *nmod_t* mod)

Reduction code used by KS2 and KS4 multiplication.

void `_nmod_poly_KS2_recover_reduce1`(*nn_ptr* res, *slong* s, *nn_srcptr* op1, *nn_srcptr* op2, *slong* n, *ulong* b, *nmod_t* mod)

Same as `_nmod_poly_KS2_recover_reduce`, but requires $0 < 2 * b \leq \text{FLINT\_BITS}$.

void `_nmod_poly_KS2_recover_reduce2`(*nn_ptr* res, *slong* s, *nn_srcptr* op1, *nn_srcptr* op2, *slong* n, *ulong* b, *nmod_t* mod)

Same as `_nmod_poly_KS2_recover_reduce`, but requires $\text{FLINT\_BITS} < 2 * b < 2 * \text{FLINT\_BITS}$.

void `_nmod_poly_KS2_recover_reduce2b`(*nn_ptr* res, *slong* s, *nn_srcptr* op1, *nn_srcptr* op2, *slong* n, *ulong* b, *nmod_t* mod)

Same as `_nmod_poly_KS2_recover_reduce`, but requires $b == \text{FLINT\_BITS}$.

void `_nmod_poly_KS2_recover_reduce3`(*nn_ptr* res, *slong* s, *nn_srcptr* op1, *nn_srcptr* op2, *slong* n, *ulong* b, *nmod_t* mod)

Same as `_nmod_poly_KS2_recover_reduce`, but requires $2 * \text{FLINT\_BITS} < 2 * b \leq 3 * \text{FLINT\_BITS}$.

void `_nmod_poly_KS2_recover_reduce`(*nn_ptr* res, *slong* s, *nn_srcptr* op1, *nn_srcptr* op2, *slong* n, *ulong* b, *nmod_t* mod)

Reduction code used by KS4 multiplication.

6.4.16 Multiplication

void `_nmod_poly_mul_classical`(*nn_ptr* res, *nn_srcptr* poly1, *slong* len1, *nn_srcptr* poly2, *slong* len2, *nmod_t* mod)

Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2). Assumes len1 $\geq$ len2 > 0 . Aliasing of inputs and output is not permitted.

void `nmod_poly_mul_classical`(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2)

Sets res to the product of poly1 and poly2.

void `_nmod_poly_mullo_classical`(*nn_ptr* res, *nn_srcptr* poly1, *slong* len1, *nn_srcptr* poly2, *slong* len2, *slong* trunc, *nmod_t* mod)

Sets res to the lower trunc coefficients of the product of (poly1, len1) and (poly2, len2). Assumes that len1 $\geq$ len2 > 0 and trunc > 0 . Aliasing of inputs and output is not permitted.

void `nmod_poly_mullo_classical`(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2, *slong* trunc)

Sets res to the lower trunc coefficients of the product of poly1 and poly2.

void `_nmod_poly_mulhigh_classical`(*nn_ptr* res, *nn_srcptr* poly1, *slong* len1, *nn_srcptr* poly2, *slong* len2, *slong* start, *nmod_t* mod)

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that len1 $\geq$ len2 > 0 . Aliasing of inputs and output is not permitted.

void `nmod_poly_mulhigh_classical`(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2, *slong* start)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced.

```
void _nmod_poly_mul_KS(nn_ptr out, nn_srcptr in1, slong len1, nn_srcptr in2, slong len2, nmod_t mod)
```

Sets `res` to the product of `in1` and `in2`. Assumes that `len1` $\geq$ `len2` $>$ 0.

```
void nmod_poly_mul_KS(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mul_KS2(nn_ptr res, nn_srcptr op1, slong n1, nn_srcptr op2, slong n2, nmod_t mod)
```

Sets `res` to the product of `op1` and `op2`. Assumes that `len1` $\geq$ `len2` $>$ 0.

```
void nmod_poly_mul_KS2(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mul_KS4(nn_ptr res, nn_srcptr op1, slong n1, nn_srcptr op2, slong n2, nmod_t mod)
```

Sets `res` to the product of `op1` and `op2`. Assumes that `len1` $\geq$ `len2` $>$ 0.

```
void nmod_poly_mul_KS4(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mulalow_KS(nn_ptr out, nn_srcptr in1, slong len1, nn_srcptr in2, slong len2, slong n, nmod_t mod)
```

Sets `out` to the low `n` coefficients of `in1` of length `len1` times `in2` of length `len2`. The output must have space for `n` coefficients. We assume that `len1` $\geq$ `len2` $>$ 0 and that $0 < n \leq \text{len1} + \text{len2} - 1$.

```
void nmod_poly_mulalow_KS(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong n)
```

Set `res` to the low `n` coefficients of `in1` of length `len1` times `in2` of length `len2`.

```
void _nmod_poly_mul(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, nmod_t mod)
```

Sets `res` to the product of `poly1` of length `len1` and `poly2` of length `len2`. Assumes `len1` $\geq$ `len2` $>$ 0. No aliasing is permitted between the inputs and the output.

```
void nmod_poly_mul(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
```

Sets `res` to the product of `poly1` and `poly2`.

```
void _nmod_poly_mulalow(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong n, nmod_t mod)
```

Sets `res` to the first `n` coefficients of the product of `poly1` of length `len1` and `poly2` of length `len2`. It is assumed that $0 < n \leq \text{len1} + \text{len2} - 1$ and that `len1` $\geq$ `len2` $>$ 0. No aliasing of inputs and output is permitted.

```
void nmod_poly_mulalow(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, slong trunc)
```

Sets `res` to the first `trunc` coefficients of the product of `poly1` and `poly2`.

```
void _nmod_poly_mulmid(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong nlo, slong nhi, nmod_t mod)
```

```
void nmod_poly_mulmid(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong nlo, slong nhi, nmod_t mod)
```

```
void _nmod_poly_mulmid_classical(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong nlo, slong nhi, nmod_t mod)
```

```
void nmod_poly_mulmid_classical(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong nlo, slong nhi, nmod_t mod)
```

```
void _nmod_poly_mulmid_KS(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong nlo, slong nhi, nmod_t mod)
```

```
void nmod_poly_mulmid_KS(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                        slong nlo, slong nhi, nmod_t mod)
```

Sets `res` to the first `nhi - nlo` middle coefficients of the product of `poly1` of length `len1` and `poly2` of length `len2` starting at offset `nlo`. It is assumed that $0 \leq nlo < nhi \leq len1 + len2 - 1$. The function `_nmod_poly_mulmid_classical()` does not support aliasing between inputs and outputs, but all others do.

```
void _nmod_poly_mulhigh(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2, slong
n, nmod_t mod)
```

Sets all but the low `n` coefficients of `res` to the corresponding coefficients of the product of `poly1` of length `len1` and `poly2` of length `len2`, the other coefficients being arbitrary. It is assumed that $len1 \geq len2 > 0$ and that $0 < n \leq len1 + len2 - 1$. Aliasing of inputs and output is not permitted.

```
void nmod_poly_mulhigh(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2,
                      slong n)
```

Sets all but the low `n` coefficients of `res` to the corresponding coefficients of the product of `poly1` and `poly2`, the remaining coefficients being arbitrary.

```
int _nmod_poly_mulalow_want_fft_small(slong len1, slong len2, slong n, int squaring, nmod_t mod)
```

Estimate whether `fft_small` multiplication should be used instead of other multiplication algorithms, given inputs of length `len1` and `len2` and output truncation to length `n`.

```
int _nmod_poly_mulmid_fft_small_repack(nn_ptr z, nn_srcptr a, slong an, nn_srcptr b, slong bn,
                                      slong znlo, slong zn, nmod_t mod)
```

Internal helper function for `_nmod_poly_mulalow_fft_small()`: if the inputs are small enough to perform a repacked convolution of half the length, multiply and return 1, otherwise do nothing and return 0. The conditions on the arguments are the same as for `_nmod_poly_mulmid_fft_small()`.

```
void _nmod_poly_mulmid_fft_small(nn_ptr z, nn_srcptr a, slong an, nn_srcptr b, slong bn, slong
znlo, slong zn, nmod_t mod)
```

```
void _nmod_poly_mulalow_fft_small(nn_ptr z, nn_srcptr a, slong an, nn_srcptr b, slong bn, slong
zn, nmod_t mod)
```

Multiplication via the `fft_small` module. Throws an error if `fft_small` is not available. The conditions on the arguments are the same as for `_nmod_poly_mulalow()`.

```
void _nmod_poly_mulmod(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                     nn_srcptr f, slong lenf, nmod_t mod)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that $len1 + len2 - lenf > 0$, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_nmod_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void nmod_poly_mulmod(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2, const
nmod_poly_t f)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _nmod_poly_mulmod_preinv(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong
len2, nn_srcptr f, slong lenf, nn_srcptr finv, slong lenfinv, nmod_t
mod)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f mod xlenf`. It is required that $len1 + len2 - lenf > 0$, which is equivalent to requiring that the result will actually be reduced. It is required that $len1 < lenf$ and $len2 < lenf$. Otherwise, simply use `_nmod_poly_mul` instead.

Aliasing of `res` with any of the inputs is not permitted.

```
void nmod_poly_mulmod_preinv(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t
                             poly2, const nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`. It is required that `poly1` and `poly2` are reduced modulo `f`.

6.4.17 Preconditioned modular multiplication

```
type nmod_poly_mulmod_precond_struct
```

```
type nmod_poly_mulmod_precond_t
```

Stores precomputed data for evaluating $ab \bmod d$ where both a and d are fixed.

```
void _nmod_poly_mulmod_precond_init_method(nmod_poly_mulmod_precond_t precondition, nn_srcptr
                                           a, slong alen, nn_srcptr d, slong dlen, nn_srcptr
                                           dinv, slong lendinv, int method, nmod_t mod)
```

```
void nmod_poly_mulmod_precond_init_method(nmod_poly_mulmod_precond_t precondition, const
                                           nmod_poly_t a, const nmod_poly_t d, const
                                           nmod_poly_t dinv, int method)
```

```
void _nmod_poly_mulmod_precond_init_num(nmod_poly_mulmod_precond_t precondition, nn_srcptr a,
                                        slong alen, nn_srcptr d, slong dlen, nn_srcptr dinv,
                                        slong lendinv, slong num, nmod_t mod)
```

```
void nmod_poly_mulmod_precond_init_num(nmod_poly_mulmod_precond_t precondition, const
                                        nmod_poly_t a, const nmod_poly_t d, const
                                        nmod_poly_t dinv, slong num)
```

Initialize `precond` for computing $ab \bmod d$. It is assumed that a is already reduced modulo d . The `method` parameter must be one of the following:

- `NMOD_POLY_MULMOD_PRECOND_NONE` (no precomputation; multiplication will simply delegate to `_nmod_poly_mulmod_preinv()`)
- `NMOD_POLY_MULMOD_PRECOND_SHOUP` (use Shoup multiplication)
- `NMOD_POLY_MULMOD_PRECOND_MATRIX` (use the matrix algorithm)

The `num` versions of these functions attempt to choose the optimal method automatically assuming that one intends to perform `num` multiplications.

Shallow references to `a`, `d` and `dinv` may be stored in `precond`; the original objects must therefore be kept alive without modification as long as `precond` is used. The user must supply the precomputed inverse of `d`, with the same meaning as in `_nmod_poly_mulmod_preinv()` and `nmod_poly_mulmod_preinv()`.

```
void nmod_poly_mulmod_precond_clear(nmod_poly_mulmod_precond_t precondition)
```

Clears `precond`, freeing any allocated memory.

```
void _nmod_poly_mulmod_precond(nn_ptr res, const nmod_poly_mulmod_precond_t precondition,
                              nn_srcptr b, slong blen, nmod_t mod)
```

```
void nmod_poly_mulmod_precond(nmod_poly_t res, const nmod_poly_mulmod_precond_t precondition,
                              const nmod_poly_t b)
```

Compute $ab \bmod d$ where both a and d are fixed and represented by the `precond` object. We require that b is already reduced modulo d . The underscore method requires nonzero lengths and does not allow aliasing between the output and any inputs (including `a` and `d`). The non-underscore method allows aliasing between `b` and `res`.

6.4.18 Powering

- `void _nmod_poly_pow_binexp(nn_ptr res, nn_srcptr poly, slong len, ulong e, nmod_t mod)`
 Raises `poly` of length `len` to the power `e` and sets `res` to the result. We require that `res` has enough space for $(len - 1)*e + 1$ coefficients. Assumes that `len` > 0, `e` > 1. Aliasing is not permitted. Uses the binary exponentiation method.
- `void nmod_poly_pow_binexp(nmod_poly_t res, const nmod_poly_t poly, ulong e)`
 Raises `poly` to the power `e` and sets `res` to the result. Uses the binary exponentiation method.
- `void _nmod_poly_pow(nn_ptr res, nn_srcptr poly, slong len, ulong e, nmod_t mod)`
 Raises `poly` of length `len` to the power `e` and sets `res` to the result. We require that `res` has enough space for $(len - 1)*e + 1$ coefficients. Assumes that `len` > 0, `e` > 1. Aliasing is not permitted.
- `void nmod_poly_pow(nmod_poly_t res, const nmod_poly_t poly, ulong e)`
 Raises `poly` to the power `e` and sets `res` to the result.
- `void _nmod_poly_pow_trunc_binexp(nn_ptr res, nn_srcptr poly, ulong e, slong trunc, nmod_t mod)`
 Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc` > 0 and that `e` > 1. Aliasing is not permitted. Uses the binary exponentiation method.
- `void nmod_poly_pow_trunc_binexp(nmod_poly_t res, const nmod_poly_t poly, ulong e, slong trunc)`
 Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.
- `void _nmod_poly_pow_trunc(nn_ptr res, nn_srcptr poly, ulong e, slong trunc, nmod_t mod)`
 Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc` > 0 and that `e` > 1. Aliasing is not permitted.
- `void nmod_poly_pow_trunc(nmod_poly_t res, const nmod_poly_t poly, ulong e, slong trunc)`
 Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.
- `void _nmod_poly_powmod_ui_binexp(nn_ptr res, nn_srcptr poly, ulong e, nn_srcptr f, slong lenf, nmod_t mod)`
 Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` > 0. We require `lenf` > 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.
- `void nmod_poly_powmod_ui_binexp(nmod_poly_t res, const nmod_poly_t poly, ulong e, const nmod_poly_t f)`
 Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` >= 0.
- `void _nmod_poly_powmod_fmpz_binexp(nn_ptr res, nn_srcptr poly, fmpz_t e, nn_srcptr f, slong lenf, nmod_t mod)`
 Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` > 0. We require `lenf` > 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.
- `void nmod_poly_powmod_fmpz_binexp(nmod_poly_t res, const nmod_poly_t poly, fmpz_t e, const nmod_poly_t f)`

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $\geq$ 0.

```
void _nmod_poly_powmod_ui_binexp_preinv(nn_ptr res, nn_srcptr poly, ulong e, nn_srcptr f, ulong lenf, nn_srcptr finv, ulong lenfinv, nmod_t mod)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $>$ 0. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf` $>$ 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.

```
void nmod_poly_powmod_ui_binexp_preinv(nmod_poly_t res, const nmod_poly_t poly, ulong e, const nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`.

```
void _nmod_poly_powmod_fmpz_binexp_preinv(nn_ptr res, nn_srcptr poly, fmpz_t e, nn_srcptr f, ulong lenf, nn_srcptr finv, ulong lenfinv, nmod_t mod)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $>$ 0. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf` $>$ 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.

```
void nmod_poly_powmod_fmpz_binexp_preinv(nmod_poly_t res, const nmod_poly_t poly, fmpz_t e, const nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`.

```
void _nmod_poly_powmod_x_ui_preinv(nn_ptr res, ulong e, nn_srcptr f, ulong lenf, nn_srcptr finv, ulong lenfinv, nmod_t mod)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e` $>$ 0. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf` $>$ 2. The output `res` must have room for `lenf` - 1 coefficients.

```
void nmod_poly_powmod_x_ui_preinv(nmod_poly_t res, ulong e, const nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`.

```
void _nmod_poly_powmod_x_fmpz_preinv(nn_ptr res, fmpz_t e, nn_srcptr f, ulong lenf, nn_srcptr finv, ulong lenfinv, nmod_t mod)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e` $>$ 0. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf` $>$ 2. The output `res` must have room for `lenf` - 1 coefficients.

```
void nmod_poly_powmod_x_fmpz_preinv(nmod_poly_t res, fmpz_t e, const nmod_poly_t f, const nmod_poly_t finv)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`.

```
void _nmod_poly_powers_mod_preinv_naive(nn_ptr *res, nn_srcptr f, ulong flen, ulong n, nn_srcptr g, ulong glen, nn_srcptr ginv, ulong ginvlen, const nmod_t mod)
```

Compute $f^0, f^1, \dots, f^{(n-1)} \bmod g$, where `g` has length `glen` and `f` is reduced mod `g` and has length `flen` (possibly zero spaced). Assumes `res` is an array of `n` arrays each with space for

at least `glen - 1` coefficients and that `flen > 0`. We require that `ginv` of length `ginvlen` is set to the power series inverse of the reverse of `g`.

```
void nmod_poly_powers_mod_naive(nmod_poly_struct *res, const nmod_poly_t f, slong n, const
                               nmod_poly_t g)
```

Set the entries of the array `res` to $f^0, f^1, \dots, f^{(n-1)} \bmod g$. No aliasing is permitted between the entries of `res` and either of the inputs.

```
void _nmod_poly_powers_mod_preinv_threaded_pool(nn_ptr *res, nn_srcptr f, slong flen, slong n,
                                                nn_srcptr g, slong glen, nn_srcptr ginv, slong
                                                ginvlen, const nmod_t mod,
                                                thread_pool_handle *threads, slong
                                                num_threads)
```

Compute $f^0, f^1, \dots, f^{(n-1)} \bmod g$, where `g` has length `glen` and `f` is reduced mod `g` and has length `flen` (possibly zero spaced). Assumes `res` is an array of `n` arrays each with space for at least `glen - 1` coefficients and that `flen > 0`. We require that `ginv` of length `ginvlen` is set to the power series inverse of the reverse of `g`.

```
void _nmod_poly_powers_mod_preinv_threaded(nn_ptr *res, nn_srcptr f, slong flen, slong n,
                                           nn_srcptr g, slong glen, nn_srcptr ginv, slong
                                           ginvlen, const nmod_t mod)
```

Compute $f^0, f^1, \dots, f^{(n-1)} \bmod g$, where `g` has length `glen` and `f` is reduced mod `g` and has length `flen` (possibly zero spaced). Assumes `res` is an array of `n` arrays each with space for at least `glen - 1` coefficients and that `flen > 0`. We require that `ginv` of length `ginvlen` is set to the power series inverse of the reverse of `g`.

```
void nmod_poly_powers_mod_bsgs(nmod_poly_struct *res, const nmod_poly_t f, slong n, const
                               nmod_poly_t g)
```

Set the entries of the array `res` to $f^0, f^1, \dots, f^{(n-1)} \bmod g$. No aliasing is permitted between the entries of `res` and either of the inputs.

6.4.19 Division

```
void _nmod_poly_divrem_basecase(nn_ptr Q, nn_ptr R, nn_srcptr A, slong A_len, nn_srcptr B,
                               slong B_len, nmod_t mod)
```

Finds Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. If $\text{len}(B) = 0$ an exception is raised. We require that `W` is temporary space of `NMOD_DIVREM_BC_ITCH(A_len, B_len, mod)` coefficients.

```
void nmod_poly_divrem_basecase(nmod_poly_t Q, nmod_poly_t R, const nmod_poly_t A, const
                               nmod_poly_t B)
```

Finds Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. If $\text{len}(B) = 0$ an exception is raised.

```
void _nmod_poly_divrem(nn_ptr Q, nn_ptr R, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB,
                      nmod_t mod)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`. We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_divrem(nmod_poly_t Q, nmod_poly_t R, const nmod_poly_t A, const nmod_poly_t
                      B)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$.

```
void _nmod_poly_div(nn_ptr Q, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB, nmod_t mod)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where `A` is of length `lenA` and `B` is of length `lenB`, but returns only Q . We require that `Q` have space for `lenA - lenB + 1` coefficients.

```
void nmod_poly_div(nmod_poly_t Q, const nmod_poly_t A, const nmod_poly_t B)
```

Computes the quotient Q on polynomial division of A and B .

void `_nmod_poly_rem_q1`(*nn_ptr* R, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nmod_t* mod)

void `_nmod_poly_rem`(*nn_ptr* R, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nmod_t* mod)
Computes the remainder R on polynomial division of A by B .

void `nmod_poly_rem`(*nmod_poly_t* R, const *nmod_poly_t* A, const *nmod_poly_t* B)
Computes the remainder R on polynomial division of A by B .

void `_nmod_poly_divexact`(*nn_ptr* Q, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nmod_t* mod)

void `nmod_poly_divexact`(*nmod_poly_t* Q, const *nmod_poly_t* A, const *nmod_poly_t* B)
Computes the quotient Q of A and B assuming that the division is exact.

void `_nmod_poly_inv_series_basecase`(*nn_ptr* Qinv, *nn_srcptr* Q, *slong* Qlen, *slong* n, *nmod_t* mod)

Given Q of length $Qlen$ whose leading coefficient is invertible modulo the given modulus, finds a polynomial $Qinv$ of length n such that the top n coefficients of the product $Q * Qinv$ is x^{n-1} . Requires that $n > 0$. This function can be viewed as inverting a power series.

void `nmod_poly_inv_series_basecase`(*nmod_poly_t* Qinv, const *nmod_poly_t* Q, *slong* n)

Given Q of length at least n find $Qinv$ of length n such that the top n coefficients of the product $Q * Qinv$ is x^{n-1} . An exception is raised if $n = 0$ or if the length of Q is less than n . The leading coefficient of Q must be invertible modulo the modulus of Q . This function can be viewed as inverting a power series.

void `_nmod_poly_inv_series_newton`(*nn_ptr* Qinv, *nn_srcptr* Q, *slong* Qlen, *slong* n, *nmod_t* mod)
Given Q of length $Qlen$ whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length n such that $Q * Qinv$ is 1 modulo x^n . Requires $n > 0$. This function can be viewed as inverting a power series via Newton iteration.

void `nmod_poly_inv_series_newton`(*nmod_poly_t* Qinv, const *nmod_poly_t* Q, *slong* n)
Given Q find $Qinv$ such that $Q * Qinv$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series via Newton iteration.

void `_nmod_poly_inv_series`(*nn_ptr* Qinv, *nn_srcptr* Q, *slong* Qlen, *slong* n, *nmod_t* mod)
Given Q of length $Qlen$ whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length n such that $Q * Qinv$ is 1 modulo x^n . Requires $n > 0$. This function can be viewed as inverting a power series.

void `nmod_poly_inv_series`(*nmod_poly_t* Qinv, const *nmod_poly_t* Q, *slong* n)
Given Q find $Qinv$ such that $Q * Qinv$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series.

void `_nmod_poly_div_series_basecase`(*nn_ptr* Q, *nn_srcptr* A, *slong* Alen, *nn_srcptr* B, *slong* Blen, *slong* n, *nmod_t* mod)

Given polynomials A and B of length $Alen$ and $Blen$, finds the polynomial Q of length n such that $Q * B = A$ modulo x^n . We assume $n > 0$ and that the constant coefficient of B is invertible modulo the given modulus. The polynomial Q must have space for n coefficients.

void `nmod_poly_div_series_basecase`(*nmod_poly_t* Q, const *nmod_poly_t* A, const *nmod_poly_t* B, *slong* n)

Given polynomials A and B considered modulo n , finds the polynomial Q of length at most n such that $Q * B = A$ modulo x^n . We assume $n > 0$ and that the constant coefficient of B is invertible modulo the modulus. An exception is raised if $n == 0$ or the constant coefficient of B is zero.

void `_nmod_poly_div_series`(*nn_ptr* Q, *nn_srcptr* A, *slong* Alen, *nn_srcptr* B, *slong* Blen, *slong* n, *nmod_t* mod)

Given polynomials A and B of length $Alen$ and $Blen$, finds the polynomial Q of length n such that Q

* $B = A$ modulo x^n . We assume $n > 0$ and that the constant coefficient of B is invertible modulo the given modulus. The polynomial Q must have space for n coefficients.

void `nmod_poly_div_series`(*nmod_poly_t* Q, const *nmod_poly_t* A, const *nmod_poly_t* B, *slong* n)
 Given polynomials A and B considered modulo n , finds the polynomial Q of length at most n such that $Q * B = A$ modulo x^n . We assume $n > 0$ and that the constant coefficient of B is invertible modulo the modulus. An exception is raised if $n == 0$ or the constant coefficient of B is zero.

void `_nmod_poly_div_newton_n_preinv`(*nn_ptr* Q, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nn_srcptr* Binv, *slong* lenBinv, *nmod_t* mod)

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but return only Q .

We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void `nmod_poly_div_newton_n_preinv`(*nmod_poly_t* Q, const *nmod_poly_t* A, const *nmod_poly_t* B, const *nmod_poly_t* Binv)

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q .

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 * \text{the length of } B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

void `_nmod_poly_divrem_newton_n_preinv`(*nn_ptr* Q, *nn_ptr* R, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nn_srcptr* Binv, *slong* lenBinv, *nmod_t* mod)

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

void `nmod_poly_divrem_newton_n_preinv`(*nmod_poly_t* Q, *nmod_poly_t* R, const *nmod_poly_t* A, const *nmod_poly_t* B, const *nmod_poly_t* Binv)

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of B mod $x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 * \text{the length of } B - 2$.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

ulong `_nmod_poly_div_root`(*nn_ptr* Q, *nn_srcptr* A, *slong* len, *ulong* c, *nmod_t* mod)

Sets $(Q, \text{len}-1)$ to the quotient of (A, len) on division by $(x - c)$, and returns the remainder, equal to the value of A evaluated at c . A and Q are allowed to be the same, but may not overlap partially in any other way.

ulong `nmod_poly_div_root`(*nmod_poly_t* Q, const *nmod_poly_t* A, *ulong* c)

Sets Q to the quotient of A on division by $(x - c)$, and returns the remainder, equal to the value of A evaluated at c .

6.4.20 Divisibility testing

`int _nmod_poly_divides_classical(nn_ptr Q, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB, nmod_t mod)`

Returns 1 if $(B, lenB)$ divides $(A, lenA)$ and sets $(Q, lenA - lenB + 1)$ to the quotient. Otherwise, returns 0 and sets $(Q, lenA - lenB + 1)$ to zero. We require that $lenA \geq lenB > 0$.

`int nmod_poly_divides_classical(nmod_poly_t Q, const nmod_poly_t A, const nmod_poly_t B)`

Returns 1 if B divides A and sets Q to the quotient. Otherwise returns 0 and sets Q to zero.

`int _nmod_poly_divides(nn_ptr Q, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB, nmod_t mod)`

Returns 1 if $(B, lenB)$ divides $(A, lenA)$ and sets $(Q, lenA - lenB + 1)$ to the quotient. Otherwise, returns 0 and sets $(Q, lenA - lenB + 1)$ to zero. We require that $lenA \geq lenB > 0$.

`int nmod_poly_divides(nmod_poly_t Q, const nmod_poly_t A, const nmod_poly_t B)`

Returns 1 if B divides A and sets Q to the quotient. Otherwise returns 0 and sets Q to zero.

`ulong nmod_poly_remove(nmod_poly_t f, const nmod_poly_t p)`

Removes the highest possible power of p from f and returns the exponent.

6.4.21 Derivative and integral

`void _nmod_poly_derivative(nn_ptr x_prime, nn_srcptr x, slong len, nmod_t mod)`

Sets the first $len - 1$ coefficients of x_prime to the derivative of x which is assumed to be of length len . It is assumed that $len > 0$.

`void nmod_poly_derivative(nmod_poly_t x_prime, const nmod_poly_t x)`

Sets x_prime to the derivative of x .

`void _nmod_poly_integral(nn_ptr x_int, nn_srcptr x, slong len, nmod_t mod)`

Set the first len coefficients of x_int to the integral of x which is assumed to be of length $len - 1$. The constant term of x_int is set to zero. It is assumed that $len > 0$. The result is only well-defined if the modulus is a prime number strictly larger than the degree of x . Supports aliasing between the two polynomials.

`void nmod_poly_integral(nmod_poly_t x_int, const nmod_poly_t x)`

Set x_int to the indefinite integral of x with constant term zero. The result is only well-defined if the modulus is a prime number strictly larger than the degree of x .

6.4.22 Evaluation

`ulong _nmod_poly_evaluate_nmod_precomp(nn_srcptr poly, slong len, ulong c, ulong c_precomp, ulong n)`

Evaluates $poly$ at the value c and reduces modulo the given modulus $modn$. The value c should be reduced modulo the modulus, and the modulus must be less than $2^{\text{FLINT\_BITS}-1}$. The algorithm used is Horner's method, with multiplications done via `n_mulmod_shoup()` using the precomputed `c_precomp` obtained via `n_mulmod_precomp_shoup()`.

`ulong _nmod_poly_evaluate_nmod_precomp_lazy(nn_srcptr poly, slong len, ulong c, ulong c_precomp, ulong n)`

Evaluates $poly$ at the value c modulo n , with lazy reductions modulo n . Precisely, if all coefficients of $poly$ are less than m , the input requirement is $m \leq 2^{\text{FLINT\_BITS}} - 2n + 1$, and the output value is in $[0, m + 2n - 1)$ and equal to the sought evaluation modulo n . In particular the coefficients of $poly$ need not be reduced modulo n , and the output may not be either. However, the value c should be reduced modulo n .

In the case where $m = n$ (coefficients of $poly$ are reduced modulo n), then the above leads to the requirement $3n - 1 \leq 2^{\text{FLINT\_BITS}}$ (this is $n \leq 6148914691236517205$ for 64 bits, and $n \leq 1431655765$ for 32 bits).

for 32 bits), and reducing the output just amounts to subtracting n or $2n$. The algorithm used is Horner's method, with multiplications done as in `n_mulmod_shoup()` using the precomputed `c_precomp` obtained via `n_mulmod_precomp_shoup()`.

```
ulong _nmod_poly_evaluate_nmod(nn_srcptr poly, slong len, ulong c, nmod_t mod)
```

Evaluates `poly` at the value `c` and reduces modulo the given modulus of `poly`. The value `c` should be reduced modulo the modulus. The algorithm used is Horner's method, with multiplications done via `nmod_mul()`.

```
ulong nmod_poly_evaluate_nmod(const nmod_poly_t poly, ulong c)
```

Evaluates `poly` at the value `c` and reduces modulo the modulus of `poly`. The value `c` should be reduced modulo the modulus. The algorithm used is Horner's method, with multiplications and additions done differently depending on the modulus `poly->mod` and on the degree (calls one of `_nmod_poly_evaluate_nmod()`, `_nmod_poly_evaluate_nmod_precomp()`, `_nmod_poly_evaluate_nmod_precomp_lazy()`).

```
void nmod_poly_evaluate_mat_horner(nmod_mat_t dest, const nmod_poly_t poly, const nmod_mat_t c)
```

Evaluates `poly` with matrix as an argument at the value `c` and stores the result in `dest`. The dimension and modulus of `dest` is assumed to be same as that of `c`. `dest` and `c` may be aliased. Horner's Method is used to compute the result.

```
void nmod_poly_evaluate_mat_paterson_stockmeyer(nmod_mat_t dest, const nmod_poly_t poly, const nmod_mat_t c)
```

Evaluates `poly` with matrix as an argument at the value `c` and stores the result in `dest`. The dimension and modulus of `dest` is assumed to be same as that of `c`. `dest` and `c` may be aliased. Paterson-Stockmeyer algorithm is used to compute the result. The algorithm is described in [Paterson1973].

```
void nmod_poly_evaluate_mat(nmod_mat_t dest, const nmod_poly_t poly, const nmod_mat_t c)
```

Evaluates `poly` with matrix as an argument at the value `c` and stores the result in `dest`. The dimension and modulus of `dest` is assumed to be same as that of `c`. `dest` and `c` may be aliased. This function automatically switches between Horner's method and the Paterson-Stockmeyer algorithm.

6.4.23 Multipoint evaluation

```
void _nmod_poly_evaluate_nmod_vec_iter(nn_ptr ys, nn_srcptr poly, slong len, nn_srcptr xs, slong n, nmod_t mod)
```

Evaluates `(coeffs, len)` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void nmod_poly_evaluate_nmod_vec_iter(nn_ptr ys, const nmod_poly_t poly, nn_srcptr xs, slong n)
```

Evaluates `poly` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void _nmod_poly_evaluate_nmod_vec_fast_precomp(nn_ptr vs, nn_srcptr poly, slong plen, const nn_ptr *tree, slong len, nmod_t mod)
```

Evaluates `(poly, plen)` at the `len` values given by the precomputed subproduct tree `tree`.

```
void _nmod_poly_evaluate_nmod_vec_fast(nn_ptr ys, nn_srcptr poly, slong len, nn_srcptr xs, slong n, nmod_t mod)
```

Evaluates `(coeffs, len)` at the `n` values given in the vector `xs`, writing the output values to `ys`. The values in `xs` should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void nmod_poly_evaluate_nmod_vec_fast(nn_ptr ys, const nmod_poly_t poly, nn_srcptr xs, slong
                                     n)
```

Evaluates *poly* at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void _nmod_poly_evaluate_nmod_vec(nn_ptr ys, nn_srcptr poly, slong len, nn_srcptr xs, slong n,
                                  nmod_t mod)
```

Evaluates (*poly*, *len*) at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

```
void nmod_poly_evaluate_nmod_vec(nn_ptr ys, const nmod_poly_t poly, nn_srcptr xs, slong n)
```

Evaluates *poly* at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

```
void _nmod_poly_evaluate_geometric_nmod_vec_iter(nn_ptr ys, nn_srcptr coeffs, slong len, ulong
                                                  r, slong n, nmod_t mod)
```

Evaluates (*coeffs*, *len*) at the first *n* powers of the square of *r*, writing the output values to *ys*. The value of *r* should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void nmod_poly_evaluate_geometric_nmod_vec_iter(nn_ptr ys, const nmod_poly_t poly, ulong r,
                                                  slong n)
```

Evaluates *poly* at the first *n* powers of the square of *r*, writing the output values to *ys*. The value of *r* should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void _nmod_poly_evaluate_geometric_nmod_vec_fast_precomp(nn_ptr vs, nn_srcptr poly, slong
                                                         plen, const
                                                         nmod_geometric_progression_t G,
                                                         slong len, nmod_t mod)
```

Evaluates (*poly*, *plen*) at the *len* values given by the precomputed geometric progression *G*. The value of *len* should be less than or equal to the precomputation size parameter *G->len*.

```
void _nmod_poly_evaluate_geometric_nmod_vec_fast(nn_ptr ys, nn_srcptr coeffs, slong len, ulong
                                                  r, slong n, nmod_t mod)
```

Evaluates (*coeffs*, *len*) at the first *n* powers of the square of *r*, writing the output values to *ys*. The value of *r* should be reduced modulo the modulus *mod* and of sufficient multiplicative order such that none of the first *n* powers of r^2 is one.

Uses fast geometric multipoint evaluation, building a temporary geometric progression precomputation.

```
void nmod_poly_evaluate_geometric_nmod_vec_fast(nn_ptr ys, const nmod_poly_t poly, ulong r,
                                                  slong n)
```

Evaluates *poly* at the first *n* powers of the square of *r*, writing the output values to *ys*. The value of *r* should be reduced modulo the modulus of the polynomial and of sufficient multiplicative order such that none of the first *n* powers of r^2 is one.

Uses fast geometric multipoint evaluation, building a temporary geometric progression precomputation.

6.4.24 Interpolation

void `_nmod_poly_interpolate_nmod_vec`(*nn_ptr* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n, *nmod_t* mod)

Sets `poly` to the unique polynomial of length at most `n` that interpolates the `n` given evaluation points `xs` and values `ys`. If the interpolating polynomial is shorter than length `n`, the leading coefficients are set to zero.

The values in `xs` and `ys` should be reduced modulo the modulus, and all `xs` must be distinct. Aliasing between `poly` and `xs` or `ys` is not allowed.

void `nmod_poly_interpolate_nmod_vec`(*nmod_poly_t* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n)

Sets `poly` to the unique polynomial of length `n` that interpolates the `n` given evaluation points `xs` and values `ys`. The values in `xs` and `ys` should be reduced modulo the modulus, and all `xs` must be distinct.

void `_nmod_poly_interpolation_weights`(*nn_ptr* w, const *nn_ptr* *tree, *slong* len, *nmod_t* mod)

Sets `w` to the barycentric interpolation weights for fast Lagrange interpolation with respect to a given subproduct tree.

void `_nmod_poly_interpolate_nmod_vec_fast_precomp`(*nn_ptr* poly, *nn_srcptr* ys, const *nn_ptr* *tree, *nn_srcptr* weights, *slong* len, *nmod_t* mod)

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

The function values are given as `ys`. The function takes a precomputed subproduct tree `tree` and barycentric interpolation weights `weights` corresponding to the roots.

void `_nmod_poly_interpolate_nmod_vec_fast`(*nn_ptr* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n, *nmod_t* mod)

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

void `nmod_poly_interpolate_nmod_vec_fast`(*nmod_poly_t* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n)

Performs interpolation using the fast Lagrange interpolation algorithm, generating a temporary subproduct tree.

void `_nmod_poly_interpolate_nmod_vec_newton`(*nn_ptr* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n, *nmod_t* mod)

Forms the interpolating polynomial in the Newton basis using the method of divided differences and then converts it to monomial form.

void `nmod_poly_interpolate_nmod_vec_newton`(*nmod_poly_t* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n)

Forms the interpolating polynomial in the Newton basis using the method of divided differences and then converts it to monomial form.

void `_nmod_poly_interpolate_nmod_vec_barycentric`(*nn_ptr* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n, *nmod_t* mod)

Forms the interpolating polynomial using a naive implementation of the barycentric form of Lagrange interpolation.

void `nmod_poly_interpolate_nmod_vec_barycentric`(*nmod_poly_t* poly, *nn_srcptr* xs, *nn_srcptr* ys, *slong* n)

Forms the interpolating polynomial using a naive implementation of the barycentric form of Lagrange interpolation.

void `_nmod_poly_interpolate_geometric_nmod_vec_fast_precomp`(*nn_ptr* poly, *nn_srcptr* v, const *nmod_geometric_progression_t* G, *slong* len, *nmod_t* mod)


```
void nmod_poly_interpolate_geometric_nmod_vec_fast_precomp(nmod_poly_t poly, nn_srcptr v,  
                                                         const  
                                                         nmod_geometric_progression_t  
                                                         G, slong len)
```

Performs interpolation using the geometric progression precomputation *G*.

Sets *poly* to the unique polynomial of length at most *len* that interpolates according to the parameter set of *G*. The value of *len* should be equal to the precomputation size parameter *G*->*len*.

Uses fast geometric multipoint interpolation using a supplied geometric progression precomputation.

```
void nmod_poly_interpolate_geometric_nmod_vec_fast(nmod_poly_t poly, ulong r, nn_srcptr ys,  
                                                  slong n)
```

Sets *poly* to the unique polynomial of length at most *n* that interpolates the first *n* powers of *r* and values *ys*.

The values *ys* and *r* should be reduced modulo the modulus, and all *r* should be of sufficient order such that none of the first *n* powers of r^2 is one. Aliasing between *poly* and *ys* is not allowed.

Uses fast geometric multipoint interpolation, building a temporary geometric progression precomputation.

6.4.25 Composition

```
void _nmod_poly_compose_horner(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong  
                             len2, nmod_t mod)
```

Composes *poly1* of length *len1* with *poly2* of length *len2* and sets *res* to the result, i.e. evaluates *poly1* at *poly2*. The algorithm used is Horner's algorithm. We require that *res* have space for $(len1 - 1) * (len2 - 1) + 1$ coefficients. It is assumed that *len1* > 0 and *len2* > 0.

```
void nmod_poly_compose_horner(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t  
                             poly2)
```

Composes *poly1* with *poly2* and sets *res* to the result, i.e. evaluates *poly1* at *poly2*. The algorithm used is Horner's algorithm.

```
void _nmod_poly_compose_divconquer(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,  
                                  slong len2, nmod_t mod)
```

Composes *poly1* of length *len1* with *poly2* of length *len2* and sets *res* to the result, i.e. evaluates *poly1* at *poly2*. The algorithm used is the divide and conquer algorithm. We require that *res* have space for $(len1 - 1) * (len2 - 1) + 1$ coefficients. It is assumed that *len1* > 0 and *len2* > 0.

```
void nmod_poly_compose_divconquer(nmod_poly_t res, const nmod_poly_t poly1, const  
                                  nmod_poly_t poly2)
```

Composes *poly1* with *poly2* and sets *res* to the result, i.e. evaluates *poly1* at *poly2*. The algorithm used is the divide and conquer algorithm.

```
void _nmod_poly_compose(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,  
                       nmod_t mod)
```

Composes *poly1* of length *len1* with *poly2* of length *len2* and sets *res* to the result, i.e. evaluates *poly1* at *poly2*. We require that *res* have space for $(len1 - 1) * (len2 - 1) + 1$ coefficients. It is assumed that *len1* > 0 and *len2* > 0.

```
void nmod_poly_compose(nmod_poly_t res, const nmod_poly_t poly1, const nmod_poly_t poly2)
```

Composes *poly1* with *poly2* and sets *res* to the result, that is, evaluates *poly1* at *poly2*.

6.4.26 Taylor shift

void `_nmod_poly_taylor_shift_horner`(*nn_ptr* poly, *ulong* c, *slong* len, *nmod_t* mod)
 Performs the Taylor shift composing `poly` by $x + c$ in-place. Uses an efficient version Horner's rule.

void `nmod_poly_taylor_shift_horner`(*nmod_poly_t* g, const *nmod_poly_t* f, *ulong* c)
 Performs the Taylor shift composing `f` by $x + c$.

void `_nmod_poly_taylor_shift_convolution`(*nn_ptr* poly, *ulong* c, *slong* len, *nmod_t* mod)
 Performs the Taylor shift composing `poly` by $x + c$ in-place. Writes the composition as a single convolution with cost $O(M(n))$. We require that the modulus is a prime at least as large as the length.

void `nmod_poly_taylor_shift_convolution`(*nmod_poly_t* g, const *nmod_poly_t* f, *ulong* c)
 Performs the Taylor shift composing `f` by $x + c$. Writes the composition as a single convolution with cost $O(M(n))$. We require that the modulus is a prime at least as large as the length.

void `_nmod_poly_taylor_shift`(*nn_ptr* poly, *ulong* c, *slong* len, *nmod_t* mod)
 Performs the Taylor shift composing `poly` by $x + c$ in-place. We require that the modulus is a prime.

void `nmod_poly_taylor_shift`(*nmod_poly_t* g, const *nmod_poly_t* f, *ulong* c)
 Performs the Taylor shift composing `f` by $x + c$. We require that the modulus is a prime.

6.4.27 Modular composition

void `_nmod_poly_compose_mod_horner`(*nn_ptr* res, *nn_srcptr* f, *slong* lenf, *nn_srcptr* g, *nn_srcptr* h, *slong* lenh, *nmod_t* mod)
 Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.
 The algorithm used is Horner's rule.

void `nmod_poly_compose_mod_horner`(*nmod_poly_t* res, const *nmod_poly_t* f, const *nmod_poly_t* g, const *nmod_poly_t* h)
 Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. The algorithm used is Horner's rule.

void `_nmod_poly_compose_mod_brent_kung`(*nn_ptr* res, *nn_srcptr* f, *slong* lenf, *nn_srcptr* g, *nn_srcptr* h, *slong* lenh, *nmod_t* mod)
 Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . The output is not allowed to be aliased with any of the inputs.
 The algorithm used is the Brent-Kung matrix algorithm.

void `nmod_poly_compose_mod_brent_kung`(*nmod_poly_t* res, const *nmod_poly_t* f, const *nmod_poly_t* g, const *nmod_poly_t* h)
 Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . The algorithm used is the Brent-Kung matrix algorithm.

void `_nmod_poly_compose_mod_brent_kung_preinv`(*nn_ptr* res, *nn_srcptr* f, *slong* lenf, *nn_srcptr* g, *nn_srcptr* h, *slong* lenh, *nn_srcptr* hin, *slong* lenhin, *nmod_t* mod)
 Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hin` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void nmod_poly_compose_mod_brent_kung_preinv(nmod_poly_t res, const nmod_poly_t f, const  
                                             nmod_poly_t g, const nmod_poly_t h, const  
                                             nmod_poly_t hinv)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _nmod_poly_mod_matrix_rows_evaluate(nn_ptr res, const nmod_mat_t A, nn_srcptr h, slong  
                                         n, nn_srcptr poly3, slong len3, nn_srcptr poly3inv,  
                                         slong len3inv, nmod_t mod)
```

Set `res` to the polynomial $\sum_{i=0}^{d-1} A_i h^i$ modulo `poly3` where A_i denotes the i -th row of the $d \times n$ matrix `A` and `res` and `h` have length `n`. The length of `poly3` must be equal to `n + 1`.

```
void _nmod_poly_reduce_matrix_mod_poly(nmod_mat_t A, const nmod_mat_t B, const  
                                       nmod_poly_t f)
```

Sets the i th row of `A` to the reduction of the i th row of `B` modulo f for $i = 1, \dots, \sqrt{\deg(f)}$. We require `B` to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _nmod_poly_precompute_matrix_worker(void *arg_ptr)
```

Worker function version of `_nmod_poly_precompute_matrix`. Input/output is stored in `nmod_poly_matrix_precompute_arg_t`.

```
void _nmod_poly_precompute_matrix(nmod_mat_t A, nn_srcptr f, nn_srcptr g, slong leng,  
                                  nn_srcptr ginv, slong lenginv, nmod_t mod)
```

Sets the i th row of `A` to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require `A` to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g and g to be nonzero. f has to be reduced modulo g and of length one less than `leng` (possibly with zero padding).

```
void nmod_poly_precompute_matrix(nmod_mat_t A, const nmod_poly_t f, const nmod_poly_t g,  
                                 const nmod_poly_t ginv)
```

Sets the i th row of `A` to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require `A` to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g .

```
void _nmod_poly_compose_mod_brent_kung_precomp_preinv_worker(void *arg_ptr)
```

Worker function version of `_nmod_poly_compose_mod_brent_kung_precomp_preinv`. Input/output is stored in `nmod_poly_compose_mod_precomp_preinv_arg_t`.

```
void _nmod_poly_compose_mod_brent_kung_precomp_preinv(nn_ptr res, nn_srcptr f, slong lenf,  
                                                       const nmod_mat_t A, nn_srcptr h,  
                                                       slong lenh, nn_srcptr hinv, slong  
                                                       lenhinv, nmod_t mod)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. We require that the i th row of `A` contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. `A` is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void nmod_poly_compose_mod_brent_kung_precomp_preinv(nmod_poly_t res, const nmod_poly_t f,  
                                                     const nmod_mat_t A, const  
                                                     nmod_poly_t h, const nmod_poly_t  
                                                     hinv)
```

Sets `res` to the composition $f(g)$ modulo h . We require that the i th row of `A` contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. `A` is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h .

```
void _nmod_poly_compose_mod_brent_kung_vec_preinv(nmod_poly_struct *res, const
                                                    nmod_poly_struct *polys, slong len1, slong
                                                    l, nn_srcptr g, slong leng, nn_srcptr h,
                                                    slong lenh, nn_srcptr hinv, slong lenhinv,
                                                    nmod_t mod)
```

Sets **res** to the composition $f_i(g)$ modulo h for $1 \leq i \leq l$, where f_i are the first l elements of **polys**. We require that h is nonzero and that the length of g is less than the length of h . We also require that the length of f_i is less than the length of h . We require **res** to have enough memory allocated to hold l *nmod_poly_struct*'s. The entries of **res** need to be initialised and l needs to be less than **len1**. Furthermore, we require **hinv** to be the inverse of the reverse of **h**. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void nmod_poly_compose_mod_brent_kung_vec_preinv(nmod_poly_struct *res, const
                                                    nmod_poly_struct *polys, slong len1, slong
                                                    n, const nmod_poly_t g, const nmod_poly_t
                                                    h, const nmod_poly_t hinv)
```

Sets **res** to the composition $f_i(g)$ modulo h for $1 \leq i \leq n$ where f_i are the first n elements of **polys**. We require **res** to have enough memory allocated to hold n *nmod_poly_struct*. The entries of **res** need to be initialised and n needs to be less than **len1**. We require that h is nonzero and that f_i and g have smaller degree than h . Furthermore, we require **hinv** to be the inverse of the reverse of **h**. No aliasing of **res** and **polys** is allowed. The algorithm used is the Brent-Kung matrix algorithm.

```
void _nmod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(nmod_poly_struct *res,
                                                                const nmod_poly_struct
                                                                *polys, slong lenpolys,
                                                                slong l, nn_srcptr g, slong
                                                                glen, nn_srcptr poly, slong
                                                                len, nn_srcptr polyinv,
                                                                slong leninv, nmod_t mod,
                                                                thread_pool_handle
                                                                *threads, slong
                                                                num_threads)
```

Multithreaded version of *_nmod_poly_compose_mod_brent_kung_vec_preinv()*. Distributing the Horner evaluations across *flint_get_num_threads()* threads.

```
void nmod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(nmod_poly_struct *res,
                                                                const nmod_poly_struct
                                                                *polys, slong len1, slong n,
                                                                const nmod_poly_t g, const
                                                                nmod_poly_t poly, const
                                                                nmod_poly_t polyinv,
                                                                thread_pool_handle
                                                                *threads, slong
                                                                num_threads)
```

Multithreaded version of *nmod_poly_compose_mod_brent_kung_vec_preinv()*. Distributing the Horner evaluations across *flint_get_num_threads()* threads.

```
void nmod_poly_compose_mod_brent_kung_vec_preinv_threaded(nmod_poly_struct *res, const
                                                            nmod_poly_struct *polys, slong
                                                            len1, slong n, const nmod_poly_t
                                                            g, const nmod_poly_t poly, const
                                                            nmod_poly_t polyinv)
```

Multithreaded version of *nmod_poly_compose_mod_brent_kung_vec_preinv()*. Distributing the Horner evaluations across *flint_get_num_threads()* threads.

```
void _nmod_poly_compose_mod(nn_ptr res, nn_srcptr f, slong lenf, nn_srcptr g, nn_srcptr h, slong
                            lenh, nmod_t mod)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void nmod_poly_compose_mod(nmod_poly_t res, const nmod_poly_t f, const nmod_poly_t g, const
                           nmod_poly_t h)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero.

6.4.28 Greatest common divisor

```
slong _nmod_poly_gcd_euclidean(nn_ptr G, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB,
                               nmod_t mod)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA` $\geq$ `lenB` $>$ 0. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
void nmod_poly_gcd_euclidean(nmod_poly_t G, const nmod_poly_t A, const nmod_poly_t B)
```

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

```
slong _nmod_poly_gcd_euclidean_redc_half(nn_ptr G, nn_srcptr A, slong lenA, nn_srcptr B,
                                         slong lenB, nmod_t mod)
```

```
void nmod_poly_gcd_euclidean_redc_half(nmod_poly_t G, const nmod_poly_t A, const
                                       nmod_poly_t B)
```

Analogous to the Euclidean GCD implementations, but uses `nmod_redc_fast` arithmetic internally. The modulus is required to be in range for `nmod_redc_fast`, i.e. odd and smaller than $2^{\text{FLINT\_BITS}-2}$.

```
slong _nmod_poly_hgcd(nn_ptr *M, slong *lenM, nn_ptr A, slong *lenA, nn_ptr B, slong *lenB,
                     nn_srcptr a, slong lena, nn_srcptr b, slong lenb, nmod_t mod)
```

Computes the HGCD of a and b , that is, a matrix M , a sign σ and two polynomials A and B such that

$$(A, B)^t = M^{-1}(a, b)^t, \sigma = \det(M),$$

and A and B are consecutive remainders in the Euclidean remainder sequence for the division of a by b satisfying $\deg(A) \geq \frac{\deg(a)}{2} > \deg(B)$. Furthermore, M will be the product of $\begin{bmatrix} q & 1 \\ 1 & 0 \end{bmatrix}$ for the quotients q generated by such a remainder sequence. Assumes that $\deg(a) > \deg(b) > 0$, i.e. $\deg(a) > \deg(b) > 1$.

Assumes that A and B have space of size at least $\deg(a)$ and $\deg(b)$, respectively. On exit, `*lenA` and `*lenB` will contain the correct lengths of A and B .

Assumes that `M[0]`, `M[1]`, `M[2]`, and `M[3]` each point to a vector of size at least $\deg(a)$.

```
slong _nmod_poly_gcd_hgcd(nn_ptr G, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB, nmod_t
                          mod)
```

Computes the monic GCD of A and B , assuming that $\deg(A) \geq \deg(B) > 0$.

Assumes that G has space for $\deg(B)$ coefficients and returns the length of G on output.

```
void nmod_poly_gcd_hgcd(nmod_poly_t G, const nmod_poly_t A, const nmod_poly_t B)
```

Computes the monic GCD of A and B using the HGCD algorithm.

As a special case, the GCD of two zero polynomials is defined to be the zero polynomial.

The time complexity of the algorithm is $\mathcal{O}(n \log^2 n)$. For further details, see [ThullYap1990].

slong **_nmod_poly_gcd**(*nn_ptr* G, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nmod_t* mod)

Computes the GCD of A of length lenA and B of length lenB , where $\text{lenA} \geq \text{lenB} > 0$. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for lenB coefficients.

void **nmod_poly_gcd**(*nmod_poly_t* G, const *nmod_poly_t* A, const *nmod_poly_t* B)

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

slong **_nmod_poly_xgcd_euclidean**(*nn_ptr* G, *nn_ptr* S, *nn_ptr* T, *nn_srcptr* A, *slong* A_len, *nn_srcptr* B, *slong* B_len, *nmod_t* mod)

Computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

void **nmod_poly_xgcd_euclidean**(*nmod_poly_t* G, *nmod_poly_t* S, *nmod_poly_t* T, const *nmod_poly_t* A, const *nmod_poly_t* B)

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S \cdot A + T \cdot B = G$. The length of S will be at most lenB and the length of T will be at most lenA .

slong **_nmod_poly_xgcd_hgcd**(*nn_ptr* G, *nn_ptr* S, *nn_ptr* T, *nn_srcptr* A, *slong* A_len, *nn_srcptr* B, *slong* B_len, *nmod_t* mod)

Computes the GCD of A and B , where $\text{len}(A) \geq \text{len}(B) > 0$, together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \text{len}(B) - \text{len}(G)$ and $\text{len}(T) \leq \text{len}(A) - \text{len}(G)$.

Both S and T must have space for at least 2 coefficients.

No aliasing of input and output operands is permitted.

void **nmod_poly_xgcd_hgcd**(*nmod_poly_t* G, *nmod_poly_t* S, *nmod_poly_t* T, const *nmod_poly_t* A, const *nmod_poly_t* B)

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S \cdot A + T \cdot B = G$. The length of S will be at most lenB and the length of T will be at most lenA .

slong **_nmod_poly_xgcd**(*nn_ptr* G, *nn_ptr* S, *nn_ptr* T, *nn_srcptr* A, *slong* lenA, *nn_srcptr* B, *slong* lenB, *nmod_t* mod)

Computes the GCD of A and B , where $\text{len}(A) \geq \text{len}(B) > 0$, together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \text{len}(B) - \text{len}(G)$ and $\text{len}(T) \leq \text{len}(A) - \text{len}(G)$.

No aliasing of input and output operands is permitted.

```
void nmod_poly_xgcd(nmod_poly_t G, nmod_poly_t S, nmod_poly_t T, const nmod_poly_t A, const
                  nmod_poly_t B)
```

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

The polynomials S and T are set such that $S*A + T*B = G$. The length of S will be at most $\text{len}B$ and the length of T will be at most $\text{len}A$.

```
ulong _nmod_poly_resultant_euclidean(nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                                   nmod_t mod)
```

Returns the resultant of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$ using the Euclidean algorithm.

Assumes that $\text{len1} \geq \text{len2} > 0$.

Assumes that the modulus is prime.

```
ulong nmod_poly_resultant_euclidean(const nmod_poly_t f, const nmod_poly_t g)
```

Computes the resultant of f and g using the Euclidean algorithm.

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

```
ulong _nmod_poly_resultant_hgcd(nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                               nmod_t mod)
```

Returns the resultant of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$ using the half-gcd algorithm.

This algorithm computes the half-gcd as per `_nmod_poly_gcd_hgcd()` but additionally updates the resultant every time a division occurs. The half-gcd algorithm computes the GCD recursively. Given inputs a and b it lets $m = \text{len}(a)/2$ and (recursively) performs all quotients in the Euclidean algorithm which do not require the low m coefficients of a and b .

This performs quotients in exactly the same order as the ordinary Euclidean algorithm except that the low m coefficients of the polynomials in the remainder sequence are not computed. A correction step after `hgcd` has been called computes these low m coefficients (by matrix multiplication by a transformation matrix also computed by `hgcd`).

This means that from the point of view of the resultant, all but the last quotient performed by a recursive call to `hgcd` is an ordinary quotient as per the usual Euclidean algorithm. However, the final quotient may give a remainder of less than $m + 1$ coefficients, which won't be corrected until the `hgcd` correction step is performed afterwards.

To compute the adjustments to the resultant coming from this corrected quotient, we save the relevant information in an `nmod_poly_res_t` struct at the time the quotient is performed so that when the correction step is performed later, the adjustments to the resultant can be computed at that time also.

The only time an adjustment to the resultant is not required after a call to `hgcd` is if `hgcd` does nothing (the remainder may already have had less than $m + 1$ coefficients when `hgcd` was called).

Assumes that $\text{len1} \geq \text{len2} > 0$.

Assumes that the modulus is prime.

ulong **nmod_poly_resultant_hgcd**(const *nmod_poly_t* f, const *nmod_poly_t* g)

Computes the resultant of f and g using the half-gcd algorithm.

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

ulong **_nmod_poly_resultant**(*nn_srcptr* poly1, *ulong* len1, *nn_srcptr* poly2, *ulong* len2, *nmod_t* mod)

Returns the resultant of (poly1, len1) and (poly2, len2).

Assumes that len1 >= len2 > 0.

Assumes that the modulus is prime.

ulong **nmod_poly_resultant**(const *nmod_poly_t* f, const *nmod_poly_t* g)

Computes the resultant of f and g .

For two non-zero polynomials $f(x) = a_m x^m + \dots + a_0$ and $g(x) = b_n x^n + \dots + b_0$ of degrees m and n , the resultant is defined to be

$$a_m^n b_n^m \prod_{(x,y): f(x)=g(y)=0} (x-y).$$

For convenience, we define the resultant to be equal to zero if either of the two polynomials is zero.

ulong **_nmod_poly_gcdinv**(*ulong* *G, *ulong* *S, const *ulong* *A, *ulong* lenA, const *ulong* *B, *ulong* lenB, const *nmod_t* mod)

Computes (G, lenA) , $(S, \text{lenB}-1)$ such that $G \cong SA \pmod{B}$, returning the actual length of G .

Assumes that $0 < \text{len}(A) < \text{len}(B)$.

void **nmod_poly_gcdinv**(*nmod_poly_t* G, *nmod_poly_t* S, const *nmod_poly_t* A, const *nmod_poly_t* B)

Computes polynomials G and S , both reduced modulo B , such that $G \cong SA \pmod{B}$, where B is assumed to have $\text{len}(B) \geq 2$.

In the case that $A = 0 \pmod{B}$, returns $G = S = 0$.

int **_nmod_poly_invmod**(*ulong* *A, const *ulong* *B, *ulong* lenB, const *ulong* *P, *ulong* lenP, const *nmod_t* mod)

Attempts to set $(A, \text{lenP}-1)$ to the inverse of (B, lenB) modulo the polynomial (P, lenP) . Returns 1 if (B, lenB) is invertible and 0 otherwise.

Assumes that $0 < \text{len}(B) < \text{len}(P)$, and hence also $\text{len}(P) \geq 2$, but supports zero-padding in (B, lenB) .

Does not support aliasing.

Assumes that *mod* is a prime number.

int **nmod_poly_invmod**(*nmod_poly_t* A, const *nmod_poly_t* B, const *nmod_poly_t* P)

Attempts to set A to the inverse of B modulo P in the polynomial ring $(\mathbf{Z}/p\mathbf{Z})[X]$, where we assume that p is a prime number.

If $\text{len}(P) < 2$, raises an exception.

If the greatest common divisor of B and P is 1, returns 1 and sets A to the inverse of B . Otherwise, returns 0 and the value of A on exit is undefined.

6.4.29 Discriminant

ulong **_nmod_poly_discriminant**(*nn_srcptr* poly, *slong* len, *nmod_t* mod)

Return the discriminant of (poly, len). Assumes len > 1.

ulong **nmod_poly_discriminant**(const *nmod_poly_t* f)

Return the discriminant of f . We normalise the discriminant so that $\text{disc}(f) = (-1)^{n(n-1)/2} \text{res}(f, f') / \text{lc}(f)^{n-m-2}$, where $n = \text{len}(f)$ and $m = \text{len}(f')$. Thus $\text{disc}(f) = \text{lc}(f)^{2n-2} \prod_{i < j} (r_i - r_j)^2$, where $\text{lc}(f)$ is the leading coefficient of f and r_i are the roots of f .

6.4.30 Power series composition

void **_nmod_poly_compose_series**(*nn_ptr* res, *nn_srcptr* poly1, *slong* len1, *nn_srcptr* poly2, *slong* len2, *slong* n, *nmod_t* mod)

Sets **res** to the composition of **poly1** and **poly2** modulo x^n , where the constant term of **poly2** is required to be zero.

Assumes that **len1**, **len2**, **n** > 0, that **len1**, **len2** ≤ **n**, and that (**len1**-1) * (**len2**-1) + 1 ≤ **n**, and that **res** has space for **n** coefficients. Does not support aliasing between any of the inputs and the output.

Wraps **_gr_poly_compose_series()** which chooses automatically between various algorithms.

void **nmod_poly_compose_series**(*nmod_poly_t* res, const *nmod_poly_t* poly1, const *nmod_poly_t* poly2, *slong* n)

Sets **res** to the composition of **poly1** and **poly2** modulo x^n , where the constant term of **poly2** is required to be zero.

6.4.31 Power series reversion

void **_nmod_poly_revert_series**(*nn_ptr* Qinv, *nn_srcptr* Q, *slong* Qlen, *slong* n, *nmod_t* mod)

void **nmod_poly_revert_series**(*nmod_poly_t* Qinv, const *nmod_poly_t* Q, *slong* n)

Sets **Qinv** to the compositional inverse or reversion of **Q** as a power series, i.e. computes Q^{-1} such that $Q(Q^{-1}(x)) = Q^{-1}(Q(x)) = x \bmod x^n$.

It is required that $Q_0 = 0$ and that Q_1 as well as the integers $1, 2, \dots, n-1$ are invertible modulo the modulus.

Wraps **_gr_poly_revert_series()** which chooses automatically between various algorithms.

6.4.32 Square roots

The series expansions for $\sqrt{h}$ and $1/\sqrt{h}$ are defined by means of the generalised binomial theorem $(1+y)^r = \sum_{k=0}^{\infty} \binom{r}{k} y^k$. It is assumed that h has constant term 1 and that the coefficients 2^{-k} exist in the coefficient ring (i.e. 2 must be invertible).

void **_nmod_poly_invsqrt_series**(*nn_ptr* g, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set the first n terms of **g** to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1. Aliasing is not permitted.

void **nmod_poly_invsqrt_series**(*nmod_poly_t* g, const *nmod_poly_t* h, *slong* n)

Set **g** to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

void **_nmod_poly_sqrt_series**(*nn_ptr* g, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set the first n terms of **g** to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1. Aliasing is not permitted.

void **nmod_poly_sqrt_series**(*nmod_poly_t* g, const *nmod_poly_t* h, *slong* n)

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

int **_nmod_poly_sqrt**(*nn_ptr* s, *nn_srcptr* p, *slong* n, *nmod_t* mod)

If (p, n) is a perfect square, sets $(s, n / 2 + 1)$ to a square root of p and returns 1. Otherwise returns 0.

int **nmod_poly_sqrt**(*nmod_poly_t* s, const *nmod_poly_t* p)

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

6.4.33 Power sums

void **_nmod_poly_power_sums_naive**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *slong* n, *nmod_t* mod)

Compute the (truncated) power sums series of the polynomial $(poly, len)$ up to length n using Newton identities.

void **nmod_poly_power_sums_naive**(*nmod_poly_t* res, const *nmod_poly_t* poly, *slong* n)

Compute the (truncated) power sum series of the polynomial $poly$ up to length n using Newton identities.

void **_nmod_poly_power_sums_schoenhage**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *slong* n, *nmod_t* mod)

Compute the (truncated) power sums series of the polynomial $(poly, len)$ up to length n using a series expansion (a formula due to Schoenhage).

void **nmod_poly_power_sums_schoenhage**(*nmod_poly_t* res, const *nmod_poly_t* poly, *slong* n)

Compute the (truncated) power sums series of the polynomial $poly$ up to length n using a series expansion (a formula due to Schoenhage).

void **_nmod_poly_power_sums**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *slong* n, *nmod_t* mod)

Compute the (truncated) power sums series of the polynomial $(poly, len)$ up to length n .

void **nmod_poly_power_sums**(*nmod_poly_t* res, const *nmod_poly_t* poly, *slong* n)

Compute the (truncated) power sums series of the polynomial $poly$ up to length n .

void **_nmod_poly_power_sums_to_poly_naive**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *nmod_t* mod)

Compute the (monic) polynomial given by its power sums series $(poly, len)$ using Newton identities.

void **nmod_poly_power_sums_to_poly_naive**(*nmod_poly_t* res, const *nmod_poly_t* Q)

Compute the (monic) polynomial given by its power sums series Q using Newton identities.

void **_nmod_poly_power_sums_to_poly_schoenhage**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *nmod_t* mod)

Compute the (monic) polynomial given by its power sums series $(poly, len)$ using series expansion (a formula due to Schoenhage).

void **nmod_poly_power_sums_to_poly_schoenhage**(*nmod_poly_t* res, const *nmod_poly_t* Q)

Compute the (monic) polynomial given by its power sums series Q using series expansion (a formula due to Schoenhage).

void **_nmod_poly_power_sums_to_poly**(*nn_ptr* res, *nn_srcptr* poly, *slong* len, *nmod_t* mod)

Compute the (monic) polynomial given by its power sums series $(poly, len)$.

void **nmod_poly_power_sums_to_poly**(*nmod_poly_t* res, const *nmod_poly_t* Q)

Compute the (monic) polynomial given by its power sums series Q .

6.4.34 Transcendental functions

The elementary transcendental functions of a formal power series h are defined as

$$\exp(h(x)) = \sum_{k=0}^{\infty} \frac{(h(x))^k}{k!}$$

$$\log(h(x)) = \int_0^x \frac{h'(t)}{h(t)} dt$$

$$\operatorname{atan}(h(x)) = \int_0^x \frac{h'(t)}{1+(h(t))^2} dt$$

$$\operatorname{atanh}(h(x)) = \int_0^x \frac{h'(t)}{1-(h(t))^2} dt$$

$$\operatorname{asin}(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1-(h(t))^2}} dt$$

$$\operatorname{asinh}(h(x)) = \int_0^x \frac{h'(t)}{\sqrt{1+(h(t))^2}} dt$$

The functions $\sin$, $\cos$, $\tan$, etc. are defined using standard inverse or functional relations. The logarithm function assumes that h has constant term 1. All other functions assume that h has constant term 0. All functions assume that the coefficient $1/k$ or $1/k!$ exists for all indices k . When computing to order $O(x^n)$, the modulus p must therefore be a prime satisfying $p \geq n$. Further, we always require that $p > 2$ in order to be able to multiply by $1/2$ for internal purposes. If the input does not satisfy all these conditions, results are undefined. Except where otherwise noted, functions are implemented with optimal (up to constants) complexity $O(M(n))$, where $M(n)$ is the cost of polynomial multiplication.

void `_nmod_poly_log_series`(*nn_ptr* g, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set $g = \log(h) + O(x^n)$. Assumes $n > 0$ and $\text{hlen} > 0$. Aliasing of g and h is allowed.

void `nmod_poly_log_series`(*nmod_poly_t* g, const *nmod_poly_t* h, *slong* n)

Set $g = \log(h) + O(x^n)$. The case $h = 1 + cx^r$ is automatically detected and handled efficiently.

void `_nmod_poly_exp_series`(*nn_ptr* f, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set $f = \exp(h) + O(x^n)$ where h is a polynomial. Assume $n > 0$. Aliasing of g and h is not allowed.

Uses Newton iteration (an improved version of the algorithm in [HanZim2004]). For small n , falls back to the basecase algorithm.

void `_nmod_poly_exp_expinv_series`(*nn_ptr* f, *nn_ptr* g, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set $f = \exp(h) + O(x^n)$ and $g = \exp(-h) + O(x^n)$, more efficiently for large n than performing a separate inversion to obtain g . Assumes $n > 0$ and that h is zero-padded as necessary to length n . Aliasing is not allowed.

Uses Newton iteration (the version given in [HanZim2004]). For small n , falls back to the basecase algorithm.

void `nmod_poly_exp_series`(*nmod_poly_t* g, const *nmod_poly_t* h, *slong* n)

Set $g = \exp(h) + O(x^n)$. The case $h = cx^r$ is automatically detected and handled efficiently. Otherwise this function automatically uses the basecase algorithm for small n and Newton iteration otherwise.

void `_nmod_poly_atan_series`(*nn_ptr* g, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set $g = \operatorname{atan}(h) + O(x^n)$. Assumes $n > 0$. Aliasing of g and h is allowed.

void `nmod_poly_atan_series`(*nmod_poly_t* g, const *nmod_poly_t* h, *slong* n)

Set $g = \operatorname{atan}(h) + O(x^n)$.

void `_nmod_poly_atanh_series`(*nn_ptr* g, *nn_srcptr* h, *slong* hlen, *slong* n, *nmod_t* mod)

Set $g = \operatorname{atanh}(h) + O(x^n)$. Assumes $n > 0$. Aliasing of g and h is allowed.

void `nmod_poly_atanh_series`(*nmod_poly_t* g, const *nmod_poly_t* h, *slong* n)

Set $g = \operatorname{atanh}(h) + O(x^n)$.

```

void _nmod_poly_asin_series(nn_ptr g, nn_srcptr h, slong hlen, slong n, nmod_t mod)
    Set  $g = \operatorname{asin}(h) + O(x^n)$ . Assumes  $n > 0$ . Aliasing of  $g$  and  $h$  is allowed.

void nmod_poly_asin_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \operatorname{asin}(h) + O(x^n)$ .

void _nmod_poly_asinh_series(nn_ptr g, nn_srcptr h, slong hlen, slong n, nmod_t mod)
    Set  $g = \operatorname{asinh}(h) + O(x^n)$ . Assumes  $n > 0$ . Aliasing of  $g$  and  $h$  is allowed.

void nmod_poly_asinh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \operatorname{asinh}(h) + O(x^n)$ .

void _nmod_poly_sin_series(nn_ptr g, nn_srcptr h, slong n, nmod_t mod)
    Set  $g = \sin(h) + O(x^n)$ . Assumes  $n > 0$  and that  $h$  is zero-padded as necessary to length  $n$ . Aliasing
    of  $g$  and  $h$  is allowed. The value is computed using the identity  $\sin(x) = 2 \tan(x/2)/(1 + \tan^2(x/2))$ .

void nmod_poly_sin_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \sin(h) + O(x^n)$ .

void _nmod_poly_cos_series(nn_ptr g, nn_srcptr h, slong n, nmod_t mod)
    Set  $g = \cos(h) + O(x^n)$ . Assumes  $n > 0$  and that  $h$  is zero-padded as necessary to length  $n$ . Aliasing
    of  $g$  and  $h$  is allowed. The value is computed using the identity  $\cos(x) = (1 - \tan^2(x/2))/(1 + \tan^2(x/2))$ .

void nmod_poly_cos_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \cos(h) + O(x^n)$ .

void _nmod_poly_tan_series(nn_ptr g, nn_srcptr h, slong hlen, slong n, nmod_t mod)
    Set  $g = \tan(h) + O(x^n)$ . Assumes  $n > 0$  and that  $h$  is zero-padded as necessary to length  $n$ .
    Aliasing of  $g$  and  $h$  is not allowed. Uses Newton iteration to invert the  $\operatorname{atan}$  function.

void nmod_poly_tan_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \tan(h) + O(x^n)$ .

void _nmod_poly_sinh_series(nn_ptr g, nn_srcptr h, slong n, nmod_t mod)
    Set  $g = \sinh(h) + O(x^n)$ . Assumes  $n > 0$  and that  $h$  is zero-padded as necessary to length  $n$ .
    Aliasing of  $g$  and  $h$  is not allowed. Uses the identity  $\sinh(x) = (e^x - e^{-x})/2$ .

void nmod_poly_sinh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \sinh(h) + O(x^n)$ .

void _nmod_poly_cosh_series(nn_ptr g, nn_srcptr h, slong n, nmod_t mod)
    Set  $g = \cosh(h) + O(x^n)$ . Assumes  $n > 0$  and that  $h$  is zero-padded as necessary to length  $n$ .
    Aliasing of  $g$  and  $h$  is not allowed. Uses the identity  $\cosh(x) = (e^x + e^{-x})/2$ .

void nmod_poly_cosh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \cosh(h) + O(x^n)$ .

void _nmod_poly_tanh_series(nn_ptr g, nn_srcptr h, slong n, nmod_t mod)
    Set  $g = \tanh(h) + O(x^n)$ . Assumes  $n > 0$  and that  $h$  is zero-padded as necessary to length  $n$ . Uses
    the identity  $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$ .

void nmod_poly_tanh_series(nmod_poly_t g, const nmod_poly_t h, slong n)
    Set  $g = \tanh(h) + O(x^n)$ .

```

6.4.35 Special polynomials

int `_nmod_poly_conway`(*nn_ptr* op, *ulong* prime, *slong* deg)

Sets op to the coefficients to the Conway polynomial $C_{p,d}$, where p is **prime** and d is **deg**. This is done by checking against Frank Lübeck's database [Lüb2004], which has been compressed in FLINT. Returns 1 in case of success and returns 0 in case of failure.

ulong `_nmod_poly_conway_rand`(*slong* *degree, *flint_rand_t* state, int type)

Returns a pseudorandom prime and sets **degree** that when put into `_nmod_poly_conway()` will always succeed.

Here, **type** can be the following values:

- 0 for which there is a bijection between the image of this function and the database of Conway polynomials,
- 1 returns a random prime found in the database and sets **degree** to some degree less than 15 along with some prime found in the database,
- 2 returns a random prime less than 2^{10} and sets **degree** to some random degree found in the database,
- 3 returns a random prime less than 2^{10} and sets **degree** to some random degree less than 15.

6.4.36 Products

void `_nmod_poly_product_roots_nmod_vec`(*nn_ptr* poly, *nn_srcptr* xs, *slong* n, *nmod_t* mod)

Sets (poly, n + 1) to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by xs.

Aliasing of the input and output is not allowed.

void `nmod_poly_product_roots_nmod_vec`(*nmod_poly_t* poly, *nn_srcptr* xs, *slong* n)

Sets poly to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by xs.

int `nmod_poly_find_distinct_nonzero_roots`(*ulong* *roots, const *nmod_poly_t* A)

If A has $\deg(A)$ distinct nonzero roots in $\mathbb{F}_p$, write these roots out to roots[0] to roots[deg(A) - 1] and return 1. Otherwise, return 0. It is assumed that A is nonzero and that the modulus of A is prime. This function uses Rabin's probabilistic method via gcd's with $(x + \delta)^{\frac{p-1}{2}} - 1$.

6.4.37 Subproduct trees

nn_ptr *`_nmod_poly_tree_alloc`(*slong* len)

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

Entry i in the tree is a pointer to a single array of limbs, capable of storing $\lfloor n/2^i \rfloor$ subproducts of degree 2^i adjacently, plus a trailing entry if $n/2^i$ is not an integer.

For example, a tree of length 7 built from monic linear factors has the following structure, where spaces have been inserted for illustrative purposes:

```
X1 X1 X1 X1 X1 X1 X1
XX1  XX1  XX1  X1
XXXX1      XX1  X1
XXXXXXX1
```

void `_nmod_poly_tree_free`(*nn_ptr* *tree, *slong* len)

Free the allocated space for the subproduct.

void **nmod_poly_tree_build**(*nn_ptr* *tree, *nn_srcptr* roots, *slong* len, *nmod_t* mod)

Builds a subproduct tree in the preallocated space from the *len* monic linear factors $(x - r_i)$. The top level product is not computed.

6.4.38 Geometric progression

void **nmod_geometric_progression_init**(*nmod_geometric_progression_t* G, *ulong* r, *slong* len, *nmod_t* mod)

Builds a geometric progression multipoint evaluation / interpolation structure.

The set of points used will be $1, r^2, r^4, \dots, r^{2(len-1)}$.

The value of *r* should be reduced modulo the modulus *mod* and of sufficient multiplicative order such that none of the powers $r^2, r^4, \dots, r^{2(len-1)}$ is one.

The value of *len* should be both greater than or equal to the number of evaluation points to be considered, and greater than or equal to the length of the polynomials to be evaluated / interpolated. This allocates vectors and polynomials for a total space of $8len - 1$ coefficients.

If the modulus is not prime, this function will work under the additional assumption that all the used points r^{2k} as well as the auxiliary values $r^{2k} - 1$ are invertible.

void **nmod_geometric_progression_clear**(*nmod_geometric_progression_t* G)

Clears the allocated polynomials and vectors used in the geometric progression precomputation G.

6.4.39 Inflation and deflation

void **nmod_poly_inflate**(*nmod_poly_t* result, const *nmod_poly_t* input, *slong* inflation)

Sets *result* to the inflated polynomial $p(x^n)$ where *p* is given by *input* and *n* is given by *inflation*.

void **nmod_poly_deflate**(*nmod_poly_t* result, const *nmod_poly_t* input, *slong* deflation)

Sets *result* to the deflated polynomial $p(x^{1/n})$ where *p* is given by *input* and *n* is given by *deflation*. Requires $n > 0$.

slong **nmod_poly_deflation**(const *nmod_poly_t* input)

Returns the largest integer by which *input* can be deflated. As special cases, returns 0 if *input* is the zero polynomial and 1 if *input* is a constant polynomial.

6.4.40 Chinese Remaindering

In all of these functions the moduli (*mod.n*) of all of the *nmod_poly*'s involved is assumed to match and be prime.

void **nmod_poly_multi_crt_init**(*nmod_poly_multi_crt_t* CRT)

Initialize CRT for Chinese remaindering.

int **nmod_poly_multi_crt_precompute**(*nmod_poly_multi_crt_t* CRT, const *nmod_poly_struct* *moduli, *slong* len)

int **nmod_poly_multi_crt_precompute_p**(*nmod_poly_multi_crt_t* CRT, const *nmod_poly_struct* *const *moduli, *slong* len)

Configure CRT for repeated Chinese remaindering of *moduli*. The number of moduli, *len*, should be positive. A return of 0 indicates that the compilation failed and future calls to *nmod_poly_multi_crt_precomp()* will leave the output undefined. A return of 1 indicates that the compilation was successful, which occurs if and only if either (1) $len == 1$ and *modulus* $\neq 0$ is nonzero, or (2) all of the moduli have positive degree and are pairwise relatively prime.

void **nmod_poly_multi_crt_precomp**(*nmod_poly_t* output, const *nmod_poly_multi_crt_t* CRT, const *nmod_poly_struct* *values)


```
void nmod_poly_multi_crt_precomp_p(nmod_poly_t output, const nmod_poly_multi_crt_t CRT,
                                   const nmod_poly_struct *const *values)
```

Set output to the polynomial of lowest possible degree that is congruent to `values + i` modulo the `moduli + i` in `nmod_poly_multi_crt_precompute()`. The inputs `values + 0, ..., values + len - 1` where `len` was used in `nmod_poly_multi_crt_precompute()` are expected to be valid and have modulus matching the modulus of the moduli used in `nmod_poly_multi_crt_precompute()`.

```
int nmod_poly_multi_crt(nmod_poly_t output, const nmod_poly_struct *moduli, const
                       nmod_poly_struct *values, slong len)
```

Perform the same operation as `nmod_poly_multi_crt_precomp()` while internally constructing and destroying the precomputed data. All of the remarks in `nmod_poly_multi_crt_precompute()` apply.

```
void nmod_poly_multi_crt_clear(nmod_poly_multi_crt_t CRT)
```

Free all space used by CRT.

```
slong _nmod_poly_multi_crt_local_size(const nmod_poly_multi_crt_t CRT)
```

Return the required length of the output for `_nmod_poly_multi_crt_run()`.

```
void _nmod_poly_multi_crt_run(nmod_poly_struct *outputs, const nmod_poly_multi_crt_t CRT,
                             const nmod_poly_struct *inputs)
```

```
void _nmod_poly_multi_crt_run_p(nmod_poly_struct *outputs, const nmod_poly_multi_crt_t
                                CRT, const nmod_poly_struct *const *inputs)
```

Perform the same operation as `nmod_poly_multi_crt_precomp()` using supplied temporary space. The actual output is placed in `outputs + 0`, and `outputs` should contain space for all temporaries and should be at least as long as `_nmod_poly_multi_crt_local_size(CRT)`. Of course the moduli of these temporaries should match the modulus of the inputs.

6.4.41 Berlekamp-Massey Algorithm

The `nmod_berlekamp_massey_t` manages an unlimited stream of points $a_1, a_2, \dots$. At any point in time, after, say, n points have been added, a call to `nmod_berlekamp_massey_reduce()` will calculate the polynomials U , V and R in the extended euclidean remainder sequence with

$$Ux^n + V(a_1x^{n-1} + a_{n-1}x + \dots + a_n) = R, \quad \deg(U) < \deg(V) \leq n/2, \quad \deg(R) < n/2.$$

The polynomials V and R may be obtained with `nmod_berlekamp_massey_V_poly()` and `nmod_berlekamp_massey_R_poly()`. This class differs from `fmpz_mod_poly_minpoly()` in the following respect. Let v_i denote the coefficient of x^i in V . `fmpz_mod_poly_minpoly()` will return a polynomial V of lowest degree that annihilates the whole sequence $a_1, \dots, a_n$ as

$$\sum_i v_i a_{j+i} = 0, \quad 1 \leq j \leq n - \deg(V).$$

The cost is that a polynomial of degree $n-1$ might be returned and the return is not generally uniquely determined by the input sequence. For the `nmod_berlekamp_massey_t` we have

$$\sum_{i,j} v_i a_{j+i} x^{-j} = -U + \frac{R}{x^n},$$

and it can be seen that $\sum_i v_i a_{j+i}$ is zero for $1 \leq j < n - \deg(R)$. Thus whether or not V has annihilated the whole sequence may be checked by comparing the degrees of V and R .

```
void nmod_berlekamp_massey_init(nmod_berlekamp_massey_t B, ulong p)
```

Initialize B in characteristic p with an empty stream.

```
void nmod_berlekamp_massey_clear(nmod_berlekamp_massey_t B)
```

Free any space used by B.

void **nmod_berlekamp_massey_start_over**(nmod_berlekamp_massey_t B)

Empty the stream of points in B.

void **nmod_berlekamp_massey_set_prime**(nmod_berlekamp_massey_t B, *ulong* p)

Set the characteristic of the field and empty the stream of points in B.

void **nmod_berlekamp_massey_add_points**(nmod_berlekamp_massey_t B, const *ulong* *a, *slong* count)

void **nmod_berlekamp_massey_add_zeros**(nmod_berlekamp_massey_t B, *slong* count)

void **nmod_berlekamp_massey_add_point**(nmod_berlekamp_massey_t B, *ulong* a)

Add point(s) to the stream processed by B. The addition of any number of points will not update the V and R polynomial.

int **nmod_berlekamp_massey_reduce**(nmod_berlekamp_massey_t B)

Ensure that the polynomials V and R are up to date. The return value is 1 if this function changed V and 0 otherwise. For example, if this function is called twice in a row without adding any points in between, the return of the second call should be 0. As another example, suppose the object is emptied, the points 1, 1, 2, 3 are added, then reduce is called. This reduce should return 1 with $\deg(R) < \deg(V) = 2$ because the Fibonacci sequence has been recognized. The further addition of the two points 5, 8 and a reduce will result in a return value of 0.

slong **nmod_berlekamp_massey_point_count**(const nmod_berlekamp_massey_t B)

Return the number of points stored in B.

const *ulong* ***nmod_berlekamp_massey_points**(const nmod_berlekamp_massey_t B)

Return a pointer to the array of points stored in B. This may be NULL if **nmod_berlekamp_massey_point_count**() returns 0.

const *nmod_poly_struct* ***nmod_berlekamp_massey_V_poly**(const nmod_berlekamp_massey_t B)

Return the polynomial V in B.

const *nmod_poly_struct* ***nmod_berlekamp_massey_R_poly**(const nmod_berlekamp_massey_t B)

Return the polynomial R in B.

6.5 nmod_poly_mat.h – matrices of univariate polynomials over integers mod n (word-size n)

The *nmod_poly_mat_t* data type represents matrices whose entries are polynomials having coefficients in $\mathbb{Z}/n\mathbb{Z}$. We generally assume that n is a prime number.

The *nmod_poly_mat_t* type is defined as an array of *nmod_poly_mat_struct*'s of length one. This permits passing parameters of type *nmod_poly_mat_t* by reference.

A matrix internally consists of a single array of *nmod_poly_struct*'s, representing a dense matrix in row-major order. This array is only directly indexed during memory allocation and deallocation. A separate array holds pointers to the start of each row, and is used for all indexing. This allows the rows of a matrix to be permuted quickly by swapping pointers.

Matrices having zero rows or columns are allowed.

The shape of a matrix is fixed upon initialisation. The user is assumed to provide input and output variables whose dimensions are compatible with the given operation.

6.5.1 Types, macros and constants

type `nmod_poly_mat_struct`

type `nmod_poly_mat_t`

6.5.2 Memory management

void `nmod_poly_mat_init`(*nmod_poly_mat_t* mat, *slong* rows, *slong* cols, *ulong* n)

Initialises a matrix with the given number of rows and columns for use. The modulus is set to n .

void `nmod_poly_mat_init_set`(*nmod_poly_mat_t* mat, const *nmod_poly_mat_t* src)

Initialises a matrix `mat` of the same dimensions and modulus as `src`, and sets it to a copy of `src`.

void `nmod_poly_mat_clear`(*nmod_poly_mat_t* mat)

Frees all memory associated with the matrix. The matrix must be reinitialised if it is to be used again.

6.5.3 Truncate, shift

void `nmod_poly_mat_set_trunc`(*nmod_poly_mat_t* res, const *nmod_poly_mat_t* pmat, long len)

Set `res` to the truncation of `pmat` to length `len`. Entries of `res` are normalized.

void `nmod_poly_mat_truncate`(*nmod_poly_mat_t* pmat, long len)

Truncates `pmat` to the given length `len`, and normalize its entries. If `len` is greater than the maximum length of the entries of `pmat`, then nothing happens.

void `nmod_poly_mat_shift_left`(*nmod_poly_mat_t* res, const *nmod_poly_mat_t* pmat, *slong* k)

Sets `res` to `pmat` shifted left by k coefficients, that is, multiplied by x^k .

void `nmod_poly_mat_shift_right`(*nmod_poly_mat_t* res, const *nmod_poly_mat_t* pmat, *slong* k)

Sets `res` to `pmat` shifted right by k coefficients, that is, divide by x^k and throw away the remainder. If k is greater than or equal to the length of `pmat`, the result is the zero polynomial matrix.

6.5.4 Basic properties

slong `nmod_poly_mat_nrows`(const *nmod_poly_mat_t* mat)

Returns the number of rows in `mat`.

slong `nmod_poly_mat_ncols`(const *nmod_poly_mat_t* mat)

Returns the number of columns in `mat`.

ulong `nmod_poly_mat_modulus`(const *nmod_poly_mat_t* mat)

Returns the modulus of `mat`.

6.5.5 Basic assignment and manipulation

nmod_poly_struct *`nmod_poly_mat_entry`(const *nmod_poly_mat_t* mat, *slong* i, *slong* j)

Gives a reference to the entry at row i and column j . The reference can be passed as an input or output variable to any `nmod_poly` function for direct manipulation of the matrix element. No bounds checking is performed.

void `nmod_poly_mat_set`(*nmod_poly_mat_t* mat1, const *nmod_poly_mat_t* mat2)

Sets `mat1` to a copy of `mat2`.

void **nmod_poly_mat_set_nmod_mat**(*nmod_poly_mat_t* pmat, const *nmod_mat_t* cmat)

Sets the already-initialized polynomial matrix **pmat** to a constant matrix with the same entries as **cmat**. Both input matrices must have the same dimensions and modulus.

void **nmod_poly_mat_swap**(*nmod_poly_mat_t* mat1, *nmod_poly_mat_t* mat2)

Swaps **mat1** and **mat2** efficiently.

void **nmod_poly_mat_swap_entrywise**(*nmod_poly_mat_t* mat1, *nmod_poly_mat_t* mat2)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

6.5.6 Window

void **nmod_poly_mat_window_init**(*nmod_poly_mat_t* window, const *nmod_poly_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2)

Initializes the matrix **window** to be an $r2 - r1$ by $c2 - c1$ submatrix of **mat** whose (0,0) entry is the (r1, c1) entry of **mat**. The memory for the elements of **window** is shared with **mat**.

void **nmod_poly_mat_window_clear**(*nmod_poly_mat_t* window)

Clears the matrix **window** and releases any memory that it uses. Note that the memory of the underlying matrix that **window** points to is not freed.

6.5.7 Concatenate

void **nmod_poly_mat_concat_vertical**(*nmod_poly_mat_t* res, const *nmod_poly_mat_t* mat1, const *nmod_poly_mat_t* mat2)

Sets **res** to vertical concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $k \times n$, **res** : $(m + k) \times n$.

void **nmod_poly_mat_concat_horizontal**(*nmod_poly_mat_t* res, const *nmod_poly_mat_t* mat1, const *nmod_poly_mat_t* mat2)

Sets **res** to horizontal concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $m \times k$, **res** : $m \times (n + k)$.

6.5.8 Input and output

void **nmod_poly_mat_print**(const *nmod_poly_mat_t* mat, const char *x)

Prints the matrix **mat** to standard output, using the variable **x**.

6.5.9 Random matrix generation

void **nmod_poly_mat_randtest**(*nmod_poly_mat_t* mat, *flint_rand_t* state, *slong* len)

This is equivalent to applying **nmod_poly_randtest** to all entries in the matrix.

void **nmod_poly_mat_randtest_sparse**(*nmod_poly_mat_t* A, *flint_rand_t* state, *slong* len, float density)

Creates a random matrix with the amount of nonzero entries given approximately by the **density** variable, which should be a fraction between 0 (most sparse) and 1 (most dense).

The nonzero entries will have random lengths between 1 and **len**.

6.5.10 Special matrices

void **nmod_poly_mat_zero**(*nmod_poly_mat_t* mat)

Sets **mat** to the zero matrix.

void **nmod_poly_mat_one**(*nmod_poly_mat_t* mat)

Sets **mat** to the unit or identity matrix of given shape, having the element 1 on the main diagonal and zeros elsewhere. If **mat** is nonsquare, it is set to the truncation of a unit matrix.

6.5.11 Basic comparison and properties

int **nmod_poly_mat_equal**(const *nmod_poly_mat_t* mat1, const *nmod_poly_mat_t* mat2)

Returns nonzero if **mat1** and **mat2** have the same shape and all their entries agree, and returns zero otherwise.

int **nmod_poly_mat_equal_nmod_mat**(const *nmod_poly_mat_t* pmat, const *nmod_mat_t* cmat)

Returns nonzero if **pmat** is a constant matrix with the same dimensions and entries as **cmat**; returns zero otherwise.

int **nmod_poly_mat_is_zero**(const *nmod_poly_mat_t* mat)

Returns nonzero if all entries in **mat** are zero, and returns zero otherwise.

int **nmod_poly_mat_is_one**(const *nmod_poly_mat_t* mat)

Returns nonzero if all entry of **mat** on the main diagonal are the constant polynomial 1 and all remaining entries are zero, and returns zero otherwise. The matrix need not be square.

int **nmod_poly_mat_is_empty**(const *nmod_poly_mat_t* mat)

Returns a non-zero value if the number of rows or the number of columns in **mat** is zero, and otherwise returns zero.

int **nmod_poly_mat_is_square**(const *nmod_poly_mat_t* mat)

Returns a non-zero value if the number of rows is equal to the number of columns in **mat**, and otherwise returns zero.

void **nmod_poly_mat_get_coeff_mat**(*nmod_mat_t* coeff, const *nmod_poly_mat_t* pmat, *slong* deg)

Sets **coeff** to be the coefficient of **pmat** of degree **deg**, where **pmat** is seen as a polynomial with matrix coefficients and coefficients are numbered from zero. **coeff** must be already initialized with the right dimensions and modulus. For entries of **pmat** of degree less than **deg**, the corresponding entry of **coeff** is zero.

void **nmod_poly_mat_set_coeff_mat**(*nmod_poly_mat_t* pmat, const *nmod_mat_t* coeff, *slong* deg)

Sets the coefficient of **pmat** of degree **deg** to **coeff**, where **pmat** is seen as a polynomial with matrix coefficients and coefficients are numbered from zero. For each entry of **pmat**, if **deg** is larger than its degree, this entry is first resized to the appropriate length, with intervening coefficients being set to zero.

6.5.12 Transposition

void **nmod_poly_mat_transpose**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A)

Sets **B** to the transpose of **A**. Dimensions must be compatible. Aliasing is allowed for square matrices.

6.5.13 Norms

ulong **nmod_poly_mat_max_length**(const *nmod_poly_mat_t* A)

Returns the maximum polynomial length among all the entries in A.

ulong **nmod_poly_mat_degree**(const *nmod_poly_mat_t* pmat)

Returns the degree of the polynomial matrix pmat. The zero matrix is deemed to have degree -1 .

6.5.14 Evaluation

void **nmod_poly_mat_evaluate_nmod**(*nmod_mat_t* B, const *nmod_poly_mat_t* A, *ulong* x)

Sets the *nmod_mat_t* B to A evaluated entrywise at the point x.

6.5.15 Arithmetic

void **nmod_poly_mat_scalar_mul_nmod_poly**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A, const *nmod_poly_t* c)

Sets B to A multiplied entrywise by the polynomial c.

void **nmod_poly_mat_scalar_mul_nmod**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A, *ulong* c)

Sets B to A multiplied entrywise by the coefficient c, which is assumed to be reduced modulo the modulus.

void **nmod_poly_mat_add**(*nmod_poly_mat_t* C, const *nmod_poly_mat_t* A, const *nmod_poly_mat_t* B)

Sets C to the sum of A and B. All matrices must have the same shape. Aliasing is allowed.

void **nmod_poly_mat_sub**(*nmod_poly_mat_t* C, const *nmod_poly_mat_t* A, const *nmod_poly_mat_t* B)

Sets C to the sum of A and B. All matrices must have the same shape. Aliasing is allowed.

void **nmod_poly_mat_neg**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A)

Sets B to the negation of A. The matrices must have the same shape. Aliasing is allowed.

void **nmod_poly_mat_mul**(*nmod_poly_mat_t* C, const *nmod_poly_mat_t* A, const *nmod_poly_mat_t* B)

Sets C to the matrix product of A and B. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical, KS and evaluation-interpolation multiplication.

void **nmod_poly_mat_mul_classical**(*nmod_poly_mat_t* C, const *nmod_poly_mat_t* A, const *nmod_poly_mat_t* B)

Sets C to the matrix product of A and B, computed using the classical algorithm. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void **nmod_poly_mat_mul_KS**(*nmod_poly_mat_t* C, const *nmod_poly_mat_t* A, const *nmod_poly_mat_t* B)

Sets C to the matrix product of A and B, computed using Kronecker segmentation. The matrices must have compatible dimensions for matrix multiplication. Aliasing is allowed.

void **nmod_poly_mat_mul_interpolate**(*nmod_poly_mat_t* C, const *nmod_poly_mat_t* A, const *nmod_poly_mat_t* B)

Sets C to the matrix product of A and B, computed through evaluation and interpolation. The matrices must have compatible dimensions for matrix multiplication. For interpolation to be well-defined, we require that the modulus is a prime at least as large as $m + n - 1$ where m and n are the maximum lengths of polynomials in the input matrices. Aliasing is allowed.

void **nmod_poly_mat_sqr**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function automatically chooses between classical and KS squaring.

void **nmod_poly_mat_sqr_classical**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses direct formulas for very small matrices, and otherwise classical matrix multiplication.

void **nmod_poly_mat_sqr_KS**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix. Aliasing is allowed. This function uses Kronecker segmentation.

void **nmod_poly_mat_sqr_interpolate**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A)

Sets B to the square of A, which must be a square matrix, computed through evaluation and interpolation. For interpolation to be well-defined, we require that the modulus is a prime at least as large as $2n - 1$ where n is the maximum length of polynomials in the input matrix. Aliasing is allowed.

void **nmod_poly_mat_pow**(*nmod_poly_mat_t* B, const *nmod_poly_mat_t* A, *ulong* exp)

Sets B to A raised to the power `exp`, where A is a square matrix. Uses exponentiation by squaring. Aliasing is allowed.

6.5.16 Row reduction

slong **nmod_poly_mat_find_pivot_any**(const *nmod_poly_mat_t* mat, *slong* start_row, *slong* end_row, *slong* c)

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r , or returns -1 if no such entry exists.

This implementation simply chooses the first nonzero entry from it encounters. This is likely to be a nearly optimal choice if all entries in the matrix have roughly the same size, but can lead to unnecessary coefficient growth if the entries vary in size.

slong **nmod_poly_mat_find_pivot_partial**(const *nmod_poly_mat_t* mat, *slong* start_row, *slong* end_row, *slong* c)

Attempts to find a pivot entry for row reduction. Returns a row index r between `start_row` (inclusive) and `stop_row` (exclusive) such that column c in `mat` has a nonzero entry on row r , or returns -1 if no such entry exists.

This implementation searches all the rows in the column and chooses the nonzero entry of smallest degree. This heuristic typically reduces coefficient growth when the matrix entries vary in size.

slong **nmod_poly_mat_fflu**(*nmod_poly_mat_t* B, *nmod_poly_t* den, *slong* *perm, const *nmod_poly_mat_t* A, int rank_check)

Uses fraction-free Gaussian elimination to set (B, den) to a fraction-free LU decomposition of A and returns the rank of A. Aliasing of A and B is allowed.

Pivot elements are chosen with `nmod_poly_mat_find_pivot_partial`. If `perm` is non-NULL, the permutation of rows in the matrix will also be applied to `perm`.

If `rank_check` is set, the function aborts and returns 0 if the matrix is detected not to have full rank without completing the elimination.

The denominator `den` is set to $\pm \det(A)$, where the sign is decided by the parity of the permutation. Note that the determinant is not generally the minimal denominator.

slong **nmod_poly_mat_rref**(*nmod_poly_mat_t* B, *nmod_poly_t* den, const *nmod_poly_mat_t* A)

Sets (B, den) to the reduced row echelon form of A and returns the rank of A. Aliasing of A and B is allowed.

The denominator `den` is set to $\pm \det(A)$. Note that the determinant is not generally the minimal denominator.

6.5.17 Trace

void `nmod_poly_mat_trace`(*nmod_poly_t* trace, const *nmod_poly_mat_t* mat)

Computes the trace of the matrix, i.e. the sum of the entries on the main diagonal. The matrix is required to be square.

6.5.18 Determinant and rank

void `nmod_poly_mat_det`(*nmod_poly_t* det, const *nmod_poly_mat_t* A)

Sets `det` to the determinant of the square matrix `A`. Uses a direct formula, fraction-free LU decomposition, or interpolation, depending on the size of the matrix.

void `nmod_poly_mat_det_fflu`(*nmod_poly_t* det, const *nmod_poly_mat_t* A)

Sets `det` to the determinant of the square matrix `A`. The determinant is computed by performing a fraction-free LU decomposition on a copy of `A`.

void `nmod_poly_mat_det_interpolate`(*nmod_poly_t* det, const *nmod_poly_mat_t* A)

Sets `det` to the determinant of the square matrix `A`. The determinant is computed by determining a bound n for its length, evaluating the matrix at n distinct points, computing the determinant of each coefficient matrix, and forming the interpolating polynomial.

If the coefficient ring does not contain n distinct points (that is, if working over $\mathbf{Z}/p\mathbf{Z}$ where $p < n$), this function automatically falls back to `nmod_poly_mat_det_fflu`.

slong `nmod_poly_mat_rank`(const *nmod_poly_mat_t* A)

Returns the rank of `A`. Performs fraction-free LU decomposition on a copy of `A`.

6.5.19 Inverse

int `nmod_poly_mat_inv`(*nmod_poly_mat_t* Ainv, *nmod_poly_t* den, const *nmod_poly_mat_t* A)

Sets (`Ainv`, `den`) to the inverse matrix of `A`. Returns 1 if `A` is nonsingular and 0 if `A` is singular. Aliasing of `Ainv` and `A` is allowed.

More precisely, `det` will be set to the determinant of `A` and `Ainv` will be set to the adjugate matrix of `A`. Note that the determinant is not necessarily the minimal denominator.

Uses fraction-free LU decomposition, followed by solving for the identity matrix.

6.5.20 Nullspace

slong `nmod_poly_mat_nullspace`(*nmod_poly_mat_t* res, const *nmod_poly_mat_t* mat)

Computes the right rational nullspace of the matrix `mat` and returns the nullity.

More precisely, assume that `mat` has rank r and nullity n . Then this function sets the first n columns of `res` to linearly independent vectors spanning the nullspace of `mat`. As a result, we always have $\text{rank}(\text{res}) = n$, and `mat` $\times$ `res` is the zero matrix.

The computed basis vectors will not generally be in a reduced form. In general, the polynomials in each column vector in the result will have a nontrivial common GCD.

6.5.21 Solving

```
int nmod_poly_mat_solve(nmod_poly_mat_t X, nmod_poly_t den, const nmod_poly_mat_t A,
                       const nmod_poly_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
int nmod_poly_mat_solve_fflu(nmod_poly_mat_t X, nmod_poly_t den, const nmod_poly_mat_t
                             A, const nmod_poly_mat_t B)
```

Solves the equation $AX = B$ for nonsingular A . More precisely, computes (X, den) such that $AX = B \times \text{den}$. Returns 1 if A is nonsingular and 0 if A is singular. The computed denominator will not generally be minimal.

Uses fraction-free LU decomposition followed by fraction-free forward and back substitution.

```
void nmod_poly_mat_solve_fflu_precomp(nmod_poly_mat_t X, const slong *perm, const
                                       nmod_poly_mat_t FFLU, const nmod_poly_mat_t B)
```

Performs fraction-free forward and back substitution given a precomputed fraction-free LU decomposition and corresponding permutation.

6.6 nmod_poly_factor.h – factorisation of univariate polynomials over integers mod n (word-size n)

6.6.1 Types, macros and constants

```
type nmod_poly_factor_struct
```

```
type nmod_poly_factor_t
```

6.6.2 Factorisation

```
void nmod_poly_factor_init(nmod_poly_factor_t fac)
```

Initialises `fac` for use. An `nmod_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void nmod_poly_factor_clear(nmod_poly_factor_t fac)
```

Frees all memory associated with `fac`.

```
void nmod_poly_factor_realloc(nmod_poly_factor_t fac, slong alloc)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void nmod_poly_factor_fit_length(nmod_poly_factor_t fac, slong len)
```

Ensures that the factor structure has space for at least `len` factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

```
void nmod_poly_factor_set(nmod_poly_factor_t res, const nmod_poly_factor_t fac)
```

Sets `res` to the same factorisation as `fac`.

```
void nmod_poly_factor_print(const nmod_poly_factor_t fac)
```

Prints the entries of `fac` to standard output.

void **nmod_poly_factor_insert**(*nmod_poly_factor_t* fac, const *nmod_poly_t* poly, *slong* exp)
 Inserts the factor *poly* with multiplicity *exp* into the factorisation *fac*.
 If *fac* already contains *poly*, then *exp* simply gets added to the exponent of the existing entry.

void **nmod_poly_factor_concat**(*nmod_poly_factor_t* res, const *nmod_poly_factor_t* fac)
 Concatenates two factorisations.
 This is equivalent to calling *nmod_poly_factor_insert()* repeatedly with the individual factors of *fac*.
 Does not support aliasing between *res* and *fac*.

void **nmod_poly_factor_pow**(*nmod_poly_factor_t* fac, *slong* exp)
 Raises *fac* to the power *exp*.

int **nmod_poly_is_irreducible**(const *nmod_poly_t* f)
 Returns 1 if the polynomial *f* is irreducible, otherwise returns 0.

int **nmod_poly_is_irreducible_ddf**(const *nmod_poly_t* f)
 Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int **nmod_poly_is_irreducible_rabin**(const *nmod_poly_t* f)
 Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses Rabin irreducibility test.

int **_nmod_poly_is_squarefree**(*nn_srcptr* f, *slong* len, *nmod_t* mod)
 Returns 1 if (*f*, *len*) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int **nmod_poly_is_squarefree**(const *nmod_poly_t* f)
 Returns 1 if *f* is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

void **nmod_poly_factor_squarefree**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)
 Sets *res* to a square-free factorization of *f*.

int **nmod_poly_factor_equal_deg_prob**(*nmod_poly_t* factor, *flint_rand_t* state, const *nmod_poly_t* pol, *slong* d)
 Probabilistic equal degree factorisation of *pol* into irreducible factors of degree *d*. If it passes, a factor is placed in *factor* and 1 is returned, otherwise 0 is returned and the value of *factor* is undetermined.
 Requires that *pol* be monic, non-constant and squarefree.

void **nmod_poly_factor_equal_deg**(*nmod_poly_factor_t* factors, const *nmod_poly_t* pol, *slong* d)
 Assuming *pol* is a product of irreducible factors all of degree *d*, finds all those factors and places them in *factors*. Requires that *pol* be monic, non-constant and squarefree.

void **nmod_poly_factor_distinct_deg**(*nmod_poly_factor_t* res, const *nmod_poly_t* poly, *slong* *const *degs)
 Factorises a monic non-constant squarefree polynomial *poly* of degree *n* into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of *poly* of degree *d*. Factors $f[d]$ are stored in *res*, and the degree *d* of the irreducible factors is stored in *degs* in the same order as the factors.
 Requires that *degs* has enough space for $(n/2)+1 * \text{sizeof}(\text{slong})$.

void **nmod_poly_factor_distinct_deg_threaded**(*nmod_poly_factor_t* res, const *nmod_poly_t* poly, *slong* *const *degs)
 Multithreaded version of *nmod_poly_factor_distinct_deg()*.

void **nmod_poly_factor_cantor_zassenhaus**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)

Factorises a non-constant polynomial **f** into monic irreducible factors using the Cantor-Zassenhaus algorithm.

void **nmod_poly_factor_berlekamp**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)

Factorises a non-constant, squarefree polynomial **f** into monic irreducible factors using the Berlekamp algorithm.

void **nmod_poly_factor_kaltofen_shoup**(*nmod_poly_factor_t* res, const *nmod_poly_t* poly)

Factorises a non-constant polynomial **f** into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step. If *flint_get_num_threads()* is greater than one *nmod_poly_factor_distinct_deg_threaded()* is used.

ulong **nmod_poly_factor_with_berlekamp**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)

Factorises a general polynomial **f** into monic irreducible factors and returns the leading coefficient of **f**, or 0 if **f** is the zero polynomial.

This function first checks for small special cases, deflates **f** if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

ulong **nmod_poly_factor_with_cantor_zassenhaus**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)

Factorises a general polynomial **f** into monic irreducible factors and returns the leading coefficient of **f**, or 0 if **f** is the zero polynomial.

This function first checks for small special cases, deflates **f** if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

ulong **nmod_poly_factor_with_kaltofen_shoup**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)

Factorises a general polynomial **f** into monic irreducible factors and returns the leading coefficient of **f**, or 0 if **f** is the zero polynomial.

This function first checks for small special cases, deflates **f** if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

ulong **nmod_poly_factor**(*nmod_poly_factor_t* res, const *nmod_poly_t* f)

Factorises a general polynomial **f** into monic irreducible factors and returns the leading coefficient of **f**, or 0 if **f** is the zero polynomial.

This function first checks for small special cases, deflates **f** if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs either Cantor-Zassenhaus or Berlekamp on all the individual square-free factors. Currently Cantor-Zassenhaus is used by default unless the modulus is 2, in which case Berlekamp is used.

void **_nmod_poly_interval_poly_worker**(void *arg_ptr)

Worker function to compute interval polynomials in distinct degree factorisation. Input/output is stored in *nmod_poly_interval_poly_arg_t*.

6.7 `nmod_mpoly.h` – multivariate polynomials over integers mod n (word-size n)

The exponents follow the `mpoly` interface. A coefficient may be referenced as a `ulong *`.

6.7.1 Types, macros and constants

type `nmod_mpoly_struct`

A structure holding a multivariate polynomial over the integers modulo n for word-sized n .

type `nmod_mpoly_t`

An array of length 1 of `nmod_mpoly_struct`.

type `nmod_mpoly_ctx_struct`

Context structure representing the parent ring of an `nmod_mpoly`.

type `nmod_mpoly_ctx_t`

An array of length 1 of `nmod_mpoly_ctx_struct`.

6.7.2 Context object

void `nmod_mpoly_ctx_init`(*nmod_mpoly_ctx_t* ctx, *slong* nvars, const *ordering_t* ord, *ulong* n)

Initialise a context object for a polynomial ring with the given number of variables and the given ordering. It will have coefficients modulo n . Setting $n = 0$ will give undefined behavior. The possibilities for the ordering are `ORD_LEX`, `ORD_DEGLEX` and `ORD_DEGREVLEX`.

slong `nmod_mpoly_ctx_nvars`(const *nmod_mpoly_ctx_t* ctx)

Return the number of variables used to initialize the context.

ordering_t `nmod_mpoly_ctx_ord`(const *nmod_mpoly_ctx_t* ctx)

Return the ordering used to initialize the context.

ulong `nmod_mpoly_ctx_modulus`(const *nmod_mpoly_ctx_t* ctx)

Return the modulus used to initialize the context.

void `nmod_mpoly_ctx_clear`(*nmod_mpoly_ctx_t* ctx)

Release any space allocated by *ctx*.

6.7.3 Memory management

void `nmod_mpoly_init`(*nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Initialise *A* for use with the given an initialised context object. Its value is set to zero.

void `nmod_mpoly_init2`(*nmod_mpoly_t* A, *slong* alloc, const *nmod_mpoly_ctx_t* ctx)

Initialise *A* for use with the given an initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and at least `MPOLY_MIN_BITS` bits for the exponent widths.

void `nmod_mpoly_init3`(*nmod_mpoly_t* A, *slong* alloc, *flint_bitcnt_t* bits, const *nmod_mpoly_ctx_t* ctx)

Initialise *A* for use with the given an initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and *bits* bits for the exponents.

void `nmod_mpoly_fit_length`(*nmod_mpoly_t* A, *slong* len, const *nmod_mpoly_ctx_t* ctx)

Ensure that *A* has space for at least *len* terms.

void **nmod_mpoly_realloc**(*nmod_mpoly_t* A, *slong* alloc, const *nmod_mpoly_ctx_t* ctx)

Reallocate *A* to have space for *alloc* terms. Assumes the current length of the polynomial is not greater than *alloc*.

void **nmod_mpoly_clear**(*nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Release any space allocated for *A*.

6.7.4 Input/Output

The variable strings in *x* start with the variable of most significance at index 0. If *x* is NULL, the variables are named *x1*, *x2*, etc.

char ***nmod_mpoly_get_str_pretty**(const *nmod_mpoly_t* A, const char **x, const *nmod_mpoly_ctx_t* ctx)

Return a string, which the user is responsible for cleaning up, representing *A*, given an array of variable strings *x*.

int **nmod_mpoly_fprint_pretty**(FILE *file, const *nmod_mpoly_t* A, const char **x, const *nmod_mpoly_ctx_t* ctx)

Print a string representing *A* to *file*.

int **nmod_mpoly_print_pretty**(const *nmod_mpoly_t* A, const char **x, const *nmod_mpoly_ctx_t* ctx)

Print a string representing *A* to *stdout*.

int **nmod_mpoly_set_str_pretty**(*nmod_mpoly_t* A, const char *str, const char **x, const *nmod_mpoly_ctx_t* ctx)

Set *A* to the polynomial in the null-terminates string *str* given an array *x* of variable strings. If parsing *str* fails, *A* is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in *x*. The character $\wedge$ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

6.7.5 Basic manipulation

void **nmod_mpoly_gen**(*nmod_mpoly_t* A, *slong* var, const *nmod_mpoly_ctx_t* ctx)

Set *A* to the variable of index *var*, where *var* = 0 corresponds to the variable with the most significance with respect to the ordering.

int **nmod_mpoly_is_gen**(const *nmod_mpoly_t* A, *slong* var, const *nmod_mpoly_ctx_t* ctx)

If *var* ≥ 0 , return 1 if *A* is equal to the *var*-th generator, otherwise return 0 . If *var* < 0 , return 1 if the polynomial is equal to any generator, otherwise return 0 .

void **nmod_mpoly_set**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_ctx_t* ctx)

Set *A* to *B*.

int **nmod_mpoly_equal**(const *nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is equal to *B*, else return 0 .

void **nmod_mpoly_swap**(*nmod_mpoly_t* A, *nmod_mpoly_t* B, const *nmod_mpoly_ctx_t* ctx)

Efficiently swap *A* and *B*.

6.7.6 Constants

int `nmod_mpoly_is_ui`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is a constant, else return 0.

ulong `nmod_mpoly_get_ui`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Assuming that *A* is a constant, return this constant. This function throws if *A* is not a constant.

void `nmod_mpoly_set_ui`(*nmod_mpoly_t* A, ulong *c*, const *nmod_mpoly_ctx_t* ctx)

Set *A* to the constant *c*.

void `nmod_mpoly_zero`(*nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Set *A* to the constant 0.

void `nmod_mpoly_one`(*nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Set *A* to the constant 1.

int `nmod_mpoly_equal_ui`(const *nmod_mpoly_t* A, ulong *c*, const *nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is equal to the constant *c*, else return 0.

int `nmod_mpoly_is_zero`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is the constant 0, else return 0.

int `nmod_mpoly_is_one`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is the constant 1, else return 0.

6.7.7 Degrees

int `nmod_mpoly_degrees_fit_si`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return 1 if the degrees of *A* with respect to each variable fit into an `slong`, otherwise return 0.

void `nmod_mpoly_degrees_fmpz`(fmpz***degs*, const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

void `nmod_mpoly_degrees_si`(slong**degs*, const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Set *degs* to the degrees of *A* with respect to each variable. If *A* is zero, all degrees are set to -1 .

void `nmod_mpoly_degree_fmpz`(fmpz_t *deg*, const *nmod_mpoly_t* A, slong *var*, const *nmod_mpoly_ctx_t* ctx)

slong `nmod_mpoly_degree_si`(const *nmod_mpoly_t* A, slong *var*, const *nmod_mpoly_ctx_t* ctx)

Either return or set *deg* to the degree of *A* with respect to the variable of index *var*. If *A* is zero, the degree is defined to be -1 .

int `nmod_mpoly_total_degree_fits_si`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return 1 if the total degree of *A* fits into an `slong`, otherwise return 0.

void `nmod_mpoly_total_degree_fmpz`(fmpz_t *tdeg*, const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

slong `nmod_mpoly_total_degree_si`(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Either return or set *tdeg* to the total degree of *A*. If *A* is zero, the total degree is defined to be -1 .

void `nmod_mpoly_used_vars`(int**used*, const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

For each variable index *i*, set *used*[*i*] to nonzero if the variable of index *i* appears in *A* and to zero otherwise.

6.7.8 Coefficients

ulong **nmod_mpoly_get_coeff_ui_monomial**(const *nmod_mpoly_t* A, const *nmod_mpoly_t* M, const *nmod_mpoly_ctx_t* ctx)

Assuming that M is a monomial, return the coefficient of the corresponding monomial in A . This function throws if M is not a monomial.

void **nmod_mpoly_set_coeff_ui_monomial**(*nmod_mpoly_t* A, *ulong* c, const *nmod_mpoly_t* M, const *nmod_mpoly_ctx_t* ctx)

Assuming that M is a monomial, set the coefficient of the corresponding monomial in A to c . This function throws if M is not a monomial.

ulong **nmod_mpoly_get_coeff_ui_fmpz**(const *nmod_mpoly_t* A, *fmpz* *const *exp, const *nmod_mpoly_ctx_t* ctx)

ulong **nmod_mpoly_get_coeff_ui_ui**(const *nmod_mpoly_t* A, const *ulong* *exp, const *nmod_mpoly_ctx_t* ctx)

Return the coefficient of the monomial with exponent exp .

void **nmod_mpoly_set_coeff_ui_fmpz**(*nmod_mpoly_t* A, *ulong* c, *fmpz* *const *exp, const *nmod_mpoly_ctx_t* ctx)

void **nmod_mpoly_set_coeff_ui_ui**(*nmod_mpoly_t* A, *ulong* c, const *ulong* *exp, const *nmod_mpoly_ctx_t* ctx)

Set the coefficient of the monomial with exponent exp to c .

void **nmod_mpoly_get_coeff_vars_ui**(*nmod_mpoly_t* C, const *nmod_mpoly_t* A, const *slong* *vars, const *ulong* *exps, *slong* length, const *nmod_mpoly_ctx_t* ctx)

Set C to the coefficient of A with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \leq nvars(A)$ and that the variables in $vars$ are distinct.

6.7.9 Comparison

int **nmod_mpoly_cmp**(const *nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_ctx_t* ctx)

Return 1 (resp. -1 , or 0) if A is after (resp. before, same as) B in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when A and B are both monomials.

6.7.10 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

ulong ***nmod_mpoly_term_coeff_ref**(*nmod_mpoly_t* A, *slong* i, const *nmod_mpoly_ctx_t* ctx)

Return a reference to the coefficient of index i of A .

int **nmod_mpoly_is_canonical**(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return 1 if A is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficients, and the terms must be sorted from greatest to least.

slong **nmod_mpoly_length**(const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Return the number of terms in A . If the polynomial is in canonical form, this will be the number of nonzero coefficients.

void **nmod_mpoly_resize**(*nmod_mpoly_t* A, *slong* new_length, const *nmod_mpoly_ctx_t* ctx)

Set the length of A to new_length . Terms are either deleted from the end, or new zero terms are appended.

```
ulong nmod_mpoly_get_term_coeff_ui(const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

Return the coefficient of the term of index *i*.

```
void nmod_mpoly_set_term_coeff_ui(nmod_mpoly_t A, slong i, ulong c, const nmod_mpoly_ctx_t ctx)
```

Set the coefficient of the term of index *i* to *c*.

```
int nmod_mpoly_term_exp_fits_si(const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

```
int nmod_mpoly_term_exp_fits_ui(const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

Return 1 if all entries of the exponent vector of the term of index *i* fit into an *slong* (resp. a *ulong*). Otherwise, return 0.

```
void nmod_mpoly_get_term_exp_fmpz(fmpz **exp, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

```
void nmod_mpoly_get_term_exp_ui(ulong *exp, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

```
void nmod_mpoly_get_term_exp_si(slong *exp, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

Set *exp* to the exponent vector of the term of index *i*. The *_ui* (resp. *_si*) version throws if any entry does not fit into a *ulong* (resp. *slong*).

```
ulong nmod_mpoly_get_term_var_exp_ui(const nmod_mpoly_t A, slong i, slong var, const nmod_mpoly_ctx_t ctx)
```

```
slong nmod_mpoly_get_term_var_exp_si(const nmod_mpoly_t A, slong i, slong var, const nmod_mpoly_ctx_t ctx)
```

Return the exponent of the variable *var* of the term of index *i*. This function throws if the exponent does not fit into a *ulong* (resp. *slong*).

```
void nmod_mpoly_set_term_exp_fmpz(nmod_mpoly_t A, slong i, fmpz *const *exp, const nmod_mpoly_ctx_t ctx)
```

```
void nmod_mpoly_set_term_exp_ui(nmod_mpoly_t A, slong i, const ulong *exp, const nmod_mpoly_ctx_t ctx)
```

Set the exponent of the term of index *i* to *exp*.

```
void nmod_mpoly_get_term(nmod_mpoly_t M, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

Set *M* to the term of index *i* in *A*.

```
void nmod_mpoly_get_term_monomial(nmod_mpoly_t M, const nmod_mpoly_t A, slong i, const nmod_mpoly_ctx_t ctx)
```

Set *M* to the monomial of the term of index *i* in *A*. The coefficient of *M* will be one.

```
void nmod_mpoly_push_term_ui_fmpz(nmod_mpoly_t A, ulong c, fmpz *const *exp, const nmod_mpoly_ctx_t ctx)
```

```
void nmod_mpoly_push_term_ui_ffmpz(nmod_mpoly_t A, ulong c, const fmpz *exp, const nmod_mpoly_ctx_t ctx)
```

```
void nmod_mpoly_push_term_ui_ui(nmod_mpoly_t A, ulong c, const ulong *exp, const nmod_mpoly_ctx_t ctx)
```

Append a term to *A* with coefficient *c* and exponent vector *exp*. This function runs in constant average time.

```
void nmod_mpoly_sort_terms(nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
```

Sort the terms of *A* into the canonical ordering dictated by the ordering in *ctx*. This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the bit size of *A*.

void **nmod_mpoly_combine_like_terms**(*nmod_mpoly_t* A, const *nmod_mpoly_ctx_t* ctx)

Combine adjacent like terms in A and delete terms with coefficient zero. If the terms of A were sorted to begin with, the result will be in canonical form. This function runs in linear time in the bit size of A .

void **nmod_mpoly_reverse**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_ctx_t* ctx)

Set A to the reversal of B .

6.7.11 Random generation

void **nmod_mpoly_randtest_bound**(*nmod_mpoly_t* A, *flint_rand_t* state, *slong* length, *ulong* exp_bound, const *nmod_mpoly_ctx_t* ctx)

Generate a random polynomial with length up to $length$ and exponents in the range $[0, exp\_bound - 1]$. The exponents of each variable are generated by calls to `n_randint(state, exp_bound)`.

void **nmod_mpoly_randtest_bounds**(*nmod_mpoly_t* A, *flint_rand_t* state, *slong* length, *ulong* *exp_bounds, const *nmod_mpoly_ctx_t* ctx)

Generate a random polynomial with length up to $length$ and exponents in the range $[0, exp\_bounds[i] - 1]$. The exponents of the variable of index i are generated by calls to `n_randint(state, exp_bounds[i])`.

void **nmod_mpoly_randtest_bits**(*nmod_mpoly_t* A, *flint_rand_t* state, *slong* length, *ulong* exp_bits, const *nmod_mpoly_ctx_t* ctx)

Generate a random polynomial with length up to $length$ and exponents whose packed form does not exceed the given bit count.

6.7.12 Addition/Subtraction

void **nmod_mpoly_add_ui**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, *ulong* c, const *nmod_mpoly_ctx_t* ctx)

Set A to $B + c$.

void **nmod_mpoly_sub_ui**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, *ulong* c, const *nmod_mpoly_ctx_t* ctx)

Set A to $B - c$.

void **nmod_mpoly_add**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_t* C, const *nmod_mpoly_ctx_t* ctx)

Set A to $B + C$.

void **nmod_mpoly_sub**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_t* C, const *nmod_mpoly_ctx_t* ctx)

Set A to $B - C$.

6.7.13 Scalar operations

void **nmod_mpoly_neg**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, const *nmod_mpoly_ctx_t* ctx)

Set A to $-B$.

void **nmod_mpoly_scalar_mul_ui**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, *ulong* c, const *nmod_mpoly_ctx_t* ctx)

Set A to $B \times c$.

```
void nmod_mpoly_make_monic(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_ctx_t
                        ctx)
```

Set A to B divided by the leading coefficient of B . This throws if B is zero or the leading coefficient is not invertible.

6.7.14 Differentiation

```
void nmod_mpoly_derivative(nmod_mpoly_t A, const nmod_mpoly_t B, long var, const
                        nmod_mpoly_ctx_t ctx)
```

Set A to the derivative of B with respect to the variable of index var .

6.7.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```
ulong nmod_mpoly_evaluate_all_ui(const nmod_mpoly_t A, const ulong *vals, const
                        nmod_mpoly_ctx_t ctx)
```

Return the evaluation of A where the variables are replaced by the corresponding elements of the array $vals$.

```
void nmod_mpoly_evaluate_one_ui(nmod_mpoly_t A, const nmod_mpoly_t B, long var, ulong val,
                        const nmod_mpoly_ctx_t ctx)
```

Set A to the evaluation of B where the variable of index var is replaced by val .

```
int nmod_mpoly_compose_nmod_poly(nmod_poly_t A, const nmod_mpoly_t B, nmod_poly_struct
                        *const *C, const nmod_mpoly_ctx_t ctx)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . The context object of B is $ctxB$. Return 1 for success and 0 for failure.

```
int nmod_mpoly_compose_nmod_mpoly_geobucket(nmod_mpoly_t A, const nmod_mpoly_t B,
                        nmod_mpoly_struct *const *C, const
                        nmod_mpoly_ctx_t ctxB, const
                        nmod_mpoly_ctx_t ctxAC)
```

```
int nmod_mpoly_compose_nmod_mpoly_horner(nmod_mpoly_t A, const nmod_mpoly_t B,
                        nmod_mpoly_struct *const *C, const
                        nmod_mpoly_ctx_t ctxB, const nmod_mpoly_ctx_t
                        ctxAC)
```

```
int nmod_mpoly_compose_nmod_mpoly(nmod_mpoly_t A, const nmod_mpoly_t B,
                        nmod_mpoly_struct *const *C, const nmod_mpoly_ctx_t
                        ctxB, const nmod_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . Both A and the elements of C have context object $ctxAC$, while B has context object $ctxB$. Neither of A and B is allowed to alias any other polynomial. Return 1 for success and 0 for failure. The main method attempts to perform the calculation using matrices and chooses heuristically between the `geobucket` and `horner` methods if needed.

```
void nmod_mpoly_compose_nmod_mpoly_gen(nmod_mpoly_t A, const nmod_mpoly_t B, const long
                        *c, const nmod_mpoly_ctx_t ctxB, const
                        nmod_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variable of index i in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array C is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.

6.7.16 Multiplication

```
void nmod_mpoly_mul(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const
                  nmod_mpoly_ctx_t ctx)
```

Set A to $B \times C$.

```
void nmod_mpoly_mul_johnson(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C,
                           const nmod_mpoly_ctx_t ctx)
```

```
void nmod_mpoly_mul_heap_threaded(nmod_mpoly_t A, const nmod_mpoly_t B, const
                                  nmod_mpoly_t C, const nmod_mpoly_ctx_t ctx)
```

Set A to $B \times C$ using Johnson's heap-based method. The first version always uses one thread.

```
int nmod_mpoly_mul_array(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const
                        nmod_mpoly_ctx_t ctx)
```

```
int nmod_mpoly_mul_array_threaded(nmod_mpoly_t A, const nmod_mpoly_t B, const
                                  nmod_mpoly_t C, const nmod_mpoly_ctx_t ctx)
```

Try to set A to $B \times C$ using arrays. If the return is 0, the operation was unsuccessful. Otherwise, it was successful, and the return is 1. The first version always uses one thread.

```
int nmod_mpoly_mul_dense(nmod_mpoly_t A, const nmod_mpoly_t B, const nmod_mpoly_t C, const
                        nmod_mpoly_ctx_t ctx)
```

Try to set A to $B \times C$ using univariate arithmetic. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1.

6.7.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int nmod_mpoly_pow_fmpz(nmod_mpoly_t A, const nmod_mpoly_t B, const fmpz_t k, const
                       nmod_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

```
int nmod_mpoly_pow_ui(nmod_mpoly_t A, const nmod_mpoly_t B, ulong k, const
                     nmod_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

6.7.18 Division

The division functions assume that the modulus is prime.

```
int nmod_mpoly_divides(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B, const
                      nmod_mpoly_ctx_t ctx)
```

If A is divisible by B , set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0. Note that the function `nmod_mpoly_div` below may be faster if the quotient is known to be exact.

```
void nmod_mpoly_div(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B, const
                   nmod_mpoly_ctx_t ctx)
```

Set Q to the quotient of A by B , discarding the remainder.

```
void nmod_mpoly_divrem(nmod_mpoly_t Q, nmod_mpoly_t R, const nmod_mpoly_t A, const
                      nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
```

Set Q and R to the quotient and remainder of A divided by B .

```
void nmod_mpoly_divrem_ideal(nmod_mpoly_struct **Q, nmod_mpoly_t R, const nmod_mpoly_t
                             A, nmod_mpoly_struct *const *B, slong len, const
                             nmod_mpoly_ctx_t ctx)
```

This function is as per `nmod_mpoly_divrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q . The number of divisor (and hence quotient) polynomials, is given by len .

```
int nmod_mpoly_divides_dense(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t B,
                             const nmod_mpoly_ctx_t ctx)
```

Try to do the operation of `nmod_mpoly_divides` using univariate arithmetic. If the return is -1 , the operation was unsuccessful. Otherwise, it was successful and the return is 0 or 1.

```
int nmod_mpoly_divides_monagan_pearce(nmod_mpoly_t Q, const nmod_mpoly_t A, const
                                       nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
```

Do the operation of `nmod_mpoly_divides` using the algorithm of Michael Monagan and Roman Pearce.

```
int nmod_mpoly_divides_heap_threaded(nmod_mpoly_t Q, const nmod_mpoly_t A, const
                                       nmod_mpoly_t B, const nmod_mpoly_ctx_t ctx)
```

Do the operation of `nmod_mpoly_divides` using the heap and multiple threads. This function should only be called once `global_thread_pool` has been initialized.

Note: This function is only defined if the machine is known to be strongly ordered during the configuration. To check whether this function is defined during compilation-time, use the C preprocessor macro `#ifdef nmod_mpoly_divides_heap_threaded`.

Note that, if the system is known to be strongly ordered, the underlying algorithm for this function is utilized in `nmod_mpoly_divides()`. Hence, you may find it easier to use this function instead if the C preprocessor is not available.

6.7.19 Greatest Common Divisor

The greatest common divisor functions assume that the modulus is prime.

```
void nmod_mpoly_term_content(nmod_mpoly_t M, const nmod_mpoly_t A, const
                             nmod_mpoly_ctx_t ctx)
```

Set M to the GCD of the terms of A . If A is zero, M will be zero. Otherwise, M will be a monomial with coefficient one.

```
int nmod_mpoly_content_vars(nmod_mpoly_t g, const nmod_mpoly_t A, slong *vars, slong
                             vars_length, const nmod_mpoly_ctx_t ctx)
```

Set g to the GCD of the coefficients of A when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, g will be independent of the variables $vars$.

```
int nmod_mpoly_gcd(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B, const
                   nmod_mpoly_ctx_t ctx)
```

Try to set G to the monic GCD of A and B . The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and G is left untouched.

```
int nmod_mpoly_gcd_cofactors(nmod_mpoly_t G, nmod_mpoly_t Abar, nmod_mpoly_t Bbar, const
                             nmod_mpoly_t A, const nmod_mpoly_t B, const
                             nmod_mpoly_ctx_t ctx)
```

Do the operation of `nmod_mpoly_gcd()` and also compute $Abar = A/G$ and $Bbar = B/G$ if successful.

```
int nmod_mpoly_gcd_brown(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B,
                         const nmod_mpoly_ctx_t ctx)
```

```
int nmod_mpoly_gcd_hensel(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B,
                          const nmod_mpoly_ctx_t ctx)
```

```
int nmod_mpoly_gcd_zippel(nmod_mpoly_t G, const nmod_mpoly_t A, const nmod_mpoly_t B,
                          const nmod_mpoly_ctx_t ctx)
```

Try to set G to the GCD of A and B using various algorithms.

```
int nmod_mpoly_resultant(nmod_mpoly_t R, const nmod_mpoly_t A, const nmod_mpoly_t B, slong
                        var, const nmod_mpoly_ctx_t ctx)
```

Try to set R to the resultant of A and B with respect to the variable of index var .

```
int nmod_mpoly_discriminant(nmod_mpoly_t D, const nmod_mpoly_t A, slong var, const
                           nmod_mpoly_ctx_t ctx)
```

Try to set D to the discriminant of A with respect to the variable of index var .

6.7.20 Square Root

The square root functions assume that the modulus is prime for correct operation.

```
int nmod_mpoly_sqrt(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
```

If $Q^2 = A$ has a solution, set Q to a solution and return 1, otherwise return 0 and set Q to zero.

```
int nmod_mpoly_is_square(const nmod_mpoly_t A, const nmod_mpoly_ctx_t ctx)
```

Return 1 if A is a perfect square, otherwise return 0.

```
int nmod_mpoly_quadratic_root(nmod_mpoly_t Q, const nmod_mpoly_t A, const nmod_mpoly_t
                              B, const nmod_mpoly_ctx_t ctx)
```

If $Q^2 + AQ = B$ has a solution, set Q to a solution and return 1, otherwise return 0.

6.7.21 Univariate Functions

An `nmod_mpoly_univar_t` holds a univariate polynomial in some main variable with `nmod_mpoly_t` coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Z}/n\mathbb{Z}[x_1, \dots, x_m]$ as an element of $(\mathbb{Z}/n\mathbb{Z}[x_1, \dots, x_{v-1}, x_{v+1}, \dots, x_m])[x_v]$ and vice versa.

```
void nmod_mpoly_univar_init(nmod_mpoly_univar_t A, const nmod_mpoly_ctx_t ctx)
```

Initialize A .

```
void nmod_mpoly_univar_clear(nmod_mpoly_univar_t A, const nmod_mpoly_ctx_t ctx)
```

Clear A .

```
void nmod_mpoly_univar_swap(nmod_mpoly_univar_t A, nmod_mpoly_univar_t B, const
                           nmod_mpoly_ctx_t ctx)
```

Swap A and B .

```
void nmod_mpoly_to_univar(nmod_mpoly_univar_t A, const nmod_mpoly_t B, slong var, const
                          nmod_mpoly_ctx_t ctx)
```

Set A to a univariate form of B by pulling out the variable of index var . The coefficients of A will still belong to the content ctx but will not depend on the variable of index var .

```
void nmod_mpoly_from_univar(nmod_mpoly_t A, const nmod_mpoly_univar_t B, slong var, const
                           nmod_mpoly_ctx_t ctx)
```

Set A to the normal form of B by putting in the variable of index var . This function is undefined if the coefficients of B depend on the variable of index var .

```
int nmod_mpoly_univar_degree_fits_si(const nmod_mpoly_univar_t A, const nmod_mpoly_ctx_t
                                     ctx)
```

Return 1 if the degree of A with respect to the main variable fits an `slong`. Otherwise, return 0.

slong **nmod_mpoly_univar_length**(const *nmod_mpoly_univar_t* A, const *nmod_mpoly_ctx_t* ctx)

Return the number of terms in *A* with respect to the main variable.

slong **nmod_mpoly_univar_get_term_exp_si**(*nmod_mpoly_univar_t* A, *slong* i, const *nmod_mpoly_ctx_t* ctx)

Return the exponent of the term of index *i* of *A*.

void **nmod_mpoly_univar_get_term_coeff**(*nmod_mpoly_t* c, const *nmod_mpoly_univar_t* A, *slong* i, const *nmod_mpoly_ctx_t* ctx)

void **nmod_mpoly_univar_swap_term_coeff**(*nmod_mpoly_t* c, *nmod_mpoly_univar_t* A, *slong* i, const *nmod_mpoly_ctx_t* ctx)

Set (resp. swap) *c* to (resp. with) the coefficient of the term of index *i* of *A*.

6.7.22 Internal Functions

void **nmod_mpoly_pow_rmul**(*nmod_mpoly_t* A, const *nmod_mpoly_t* B, *ulong* k, const *nmod_mpoly_ctx_t* ctx)

Set *A* to *B* raised to the *k*-th power using repeated multiplications.

void **nmod_mpoly_div_monagan_pearce**(*nmod_mpoly_t* polyq, const *nmod_mpoly_t* poly2, const *nmod_mpoly_t* poly3, const *nmod_mpoly_ctx_t* ctx)

Set *polyq* to the quotient of *poly2* by *poly3*, discarding the remainder (with notional remainder coefficients reduced modulo the leading coefficient of *poly3*). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce. This function is exceptionally efficient if the division is known to be exact.

void **nmod_mpoly_divrem_monagan_pearce**(*nmod_mpoly_t* q, *nmod_mpoly_t* r, const *nmod_mpoly_t* poly2, const *nmod_mpoly_t* poly3, const *nmod_mpoly_ctx_t* ctx)

Set *polyq* and *polyr* to the quotient and remainder of *poly2* divided by *poly3*, (with remainder coefficients reduced modulo the leading coefficient of *poly3*). Implements “Polynomial division using dynamic arrays, heaps and packed exponents” by Michael Monagan and Roman Pearce.

void **nmod_mpoly_divrem_ideal_monagan_pearce**(*nmod_mpoly_struct* **q, *nmod_mpoly_t* r, const *nmod_mpoly_t* poly2, *nmod_mpoly_struct* *const *poly3, *slong* len, const *nmod_mpoly_ctx_t* ctx)

This function is as per **nmod_mpoly_divrem_monagan_pearce** except that it takes an array of divisor polynomials *poly3*, and it returns an array of quotient polynomials *q*. The number of divisor (and hence quotient) polynomials, is given by *len*. The function computes polynomials $q_i = q[i]$ such that poly2 is $r + \sum_{i=0}^{\text{len}-1} q_i b_i$, where $b_i = \text{poly3}[i]$.

6.8 nmod_mpoly_factor.h – factorisation of multivariate polynomials over integers mod n (word-size n)

6.8.1 Types, macros and constants

type **nmod_mpoly_factor_struct**

A struct for holding a factored polynomial. There is a single constant and a product of bases to corresponding exponents.

type **nmod_mpoly_factor_t**

An array of length 1 of **nmod_mpoly_factor_struct**.

6.8.2 Memory management

void `nmod_mpoly_factor_init`(*nmod_mpoly_factor_t* f, const *nmod_mpoly_ctx_t* ctx)
Initialise *f*.

void `nmod_mpoly_factor_clear`(*nmod_mpoly_factor_t* f, const *nmod_mpoly_ctx_t* ctx)
Clear *f*.

6.8.3 Basic manipulation

void `nmod_mpoly_factor_swap`(*nmod_mpoly_factor_t* f, *nmod_mpoly_factor_t* g, const
nmod_mpoly_ctx_t ctx)

Efficiently swap *f* and *g*.

slong `nmod_mpoly_factor_length`(const *nmod_mpoly_factor_t* f, const *nmod_mpoly_ctx_t* ctx)
Return the length of the product in *f*.

ulong `nmod_mpoly_factor_get_constant_ui`(const *nmod_mpoly_factor_t* f, const
nmod_mpoly_ctx_t ctx)

Return the constant of *f*.

void `nmod_mpoly_factor_get_base`(*nmod_mpoly_t* p, const *nmod_mpoly_factor_t* f, *slong* i, const
nmod_mpoly_ctx_t ctx)

void `nmod_mpoly_factor_swap_base`(*nmod_mpoly_t* p, *nmod_mpoly_factor_t* f, *slong* i, const
nmod_mpoly_ctx_t ctx)

Set (resp. swap) *B* to (resp. with) the base of the term of index *i* in *A*.

slong `nmod_mpoly_factor_get_exp_si`(*nmod_mpoly_factor_t* f, *slong* i, const *nmod_mpoly_ctx_t*
ctx)

Return the exponent of the term of index *i* in *A*. It is assumed to fit an *slong*.

void `nmod_mpoly_factor_sort`(*nmod_mpoly_factor_t* f, const *nmod_mpoly_ctx_t* ctx)
Sort the product of *f* first by exponent and then by base.

6.8.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and *f* is undefined. None of these functions multiply *f* by *A*: *f* is simply set to a factorisation of *A*, and thus these functions should not depend on the initial value of the output *f*.

int `nmod_mpoly_factor_squarefree`(*nmod_mpoly_factor_t* f, const *nmod_mpoly_t* A, const
nmod_mpoly_ctx_t ctx)

Set *f* to a factorization of *A* where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call `nmod_mpoly_factor_sort()` and then multiply the bases with the desired exponent.

int `nmod_mpoly_factor`(*nmod_mpoly_factor_t* f, const *nmod_mpoly_t* A, const *nmod_mpoly_ctx_t*
ctx)

Set *f* to a factorization of *A* where the bases are irreducible.

6.9 mpn_mod.h – integers mod n (packed multi-word n)

This module provides efficient arithmetic in rings $R = \mathbb{Z}/n\mathbb{Z}$ for medium-sized n . Given an ℓ -limb modulus $2^{\beta(\ell-1)} \leq n < 2^{\beta\ell}$ where β is `FLINT_BITS` (32 or 64), elements are represented as ℓ -limb arrays (i.e. `ulong[1]`), zero-padded for values that happen to fit in less than ℓ limbs, which can be stack-allocated and packed consecutively without indirection or memory allocation overhead.

This module is designed to use the *generics* interface. As such, the ring is represented by a *gr_ctx_t* context object, methods return status flags (`GR_SUCCESS`, `GR_UNABLE`, `GR_DOMAIN`), and one can use generic structures such as *gr_poly_t* for polynomials and *gr_mat_t* for matrices.

6.9.1 Types, macros and constants

`MPN_MOD_MIN_LIMBS`

`MPN_MOD_MAX_LIMBS`

The number of limbs ℓ permitted in a modulus. The current limits are $2 \leq \ell \leq 16$, permitting moduli up to 512 bits on 32-bit machines and 1024 bits on 64-bit machines. We exclude single-limb moduli since these are covered by *nmod* arithmetic, and this allows not bothering with various degenerate cases. The upper limit exists so that elements and temporary buffers are safe to allocate on the stack and so that simple operations like swapping or zeroing elements are not too expensive compared to a pointer-and-size representation. A second reason is that the algorithms in this module have been tuned only for moduli in a certain range. For larger moduli, one should use *fmpz_mod* instead. The upper limit might be increased in the future.

6.9.2 Context objects

`int gr_ctx_init_mpn_mod(gr_ctx_t ctx, const fmpz_t n)`

`int _gr_ctx_init_mpn_mod(gr_ctx_t ctx, nn_srcptr n, slong nlimbs)`

Initializes *ctx* to the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n where elements are `ulong` arrays with the same number of limbs as n . This constructor does no initialization and returns `GR_DOMAIN` if the modulus is nonpositive, or `GR_UNABLE` if the modulus is not in bounds.

`void gr_ctx_init_mpn_mod_randtest(gr_ctx_t ctx, flint_rand_t state)`

Initializes *ctx* to a ring with a random modulus.

`MPN_MOD_CTX_NLIMBS(ctx)`

Retrieves the number of limbs ℓ of the modulus.

`MPN_MOD_CTX_MODULUS_BITS`

Retrieves the number of bits of the modulus.

`MPN_MOD_CTX_MODULUS(ctx)`

Pointer to the limbs of the modulus.

`MPN_MOD_CTX_NORM(ctx)`

An integer indicating the number of leading zero bits in the most significant limb of the modulus.

`MPN_MOD_CTX_MODULUS_NORMED(ctx)`

Pointer to a copy of the modulus left-shifted so that the most significant bit is in a limb boundary.

`MPN_MOD_CTX_MODULUS_PREINV(ctx)`

Pointer to a precomputed inverse of the (normed) modulus.

`MPN_MOD_CTX_IS_PRIME(ctx)`

A *truth_t* flag indicating whether n is prime.

void `mpn_mod_ctx_set_is_field`(*gr_ctx_t* ctx, *truth_t* is_prime)

Set the flag indicating whether n is prime. Setting this to `T_TRUE` speeds up some algorithms which can assume that the ring is actually a field.

6.9.3 Basic operations and arithmetic

```
int mpn_mod_ctx_write(gr_stream_t out, gr_ctx_t ctx)
void mpn_mod_ctx_clear(gr_ctx_t ctx)
truth_t mpn_mod_ctx_is_field(gr_ctx_t ctx)
void mpn_mod_init(nn_ptr x, gr_ctx_t ctx)
void mpn_mod_clear(nn_ptr x, gr_ctx_t ctx)
void mpn_mod_swap(nn_ptr x, nn_ptr y, gr_ctx_t ctx)
int mpn_mod_set(nn_ptr res, nn_srcptr x, gr_ctx_t ctx)
int mpn_mod_zero(nn_ptr res, gr_ctx_t ctx)
int mpn_mod_one(nn_ptr res, gr_ctx_t ctx)
int mpn_mod_set_ui(nn_ptr res, ulong x, gr_ctx_t ctx)
int mpn_mod_set_si(nn_ptr res, slong x, gr_ctx_t ctx)
int mpn_mod_neg_one(nn_ptr res, gr_ctx_t ctx)
int mpn_mod_set_mpn(nn_ptr res, nn_srcptr x, slong xn, gr_ctx_t ctx)
int mpn_mod_set_fmpz(nn_ptr res, const fmpz_t x, gr_ctx_t ctx)
int mpn_mod_set_other(nn_ptr res, gr_ptr v, gr_ctx_t v_ctx, gr_ctx_t ctx)
int mpn_mod_randtest(nn_ptr res, flint_rand_t state, gr_ctx_t ctx)
int mpn_mod_write(gr_stream_t out, nn_srcptr x, gr_ctx_t ctx)
int mpn_mod_get_fmpz(fmpz_t res, nn_srcptr x, gr_ctx_t ctx)
truth_t mpn_mod_is_zero(nn_srcptr x, gr_ctx_t ctx)
truth_t mpn_mod_is_one(nn_srcptr x, gr_ctx_t ctx)
truth_t mpn_mod_is_neg_one(gr_srcptr x, gr_ctx_t ctx)
truth_t mpn_mod_equal(nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)
int mpn_mod_neg(nn_ptr res, nn_srcptr x, gr_ctx_t ctx)
int mpn_mod_add(nn_ptr res, nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)
int mpn_mod_sub(nn_ptr res, nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)
int mpn_mod_add_ui(nn_ptr res, nn_srcptr x, ulong y, gr_ctx_t ctx)
int mpn_mod_sub_ui(nn_ptr res, nn_srcptr x, ulong y, gr_ctx_t ctx)
int mpn_mod_add_si(nn_ptr res, nn_srcptr x, slong y, gr_ctx_t ctx)
int mpn_mod_sub_si(nn_ptr res, nn_srcptr x, slong y, gr_ctx_t ctx)
int mpn_mod_add_fmpz(nn_ptr res, nn_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int mpn_mod_sub_fmpz(nn_ptr res, nn_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int mpn_mod_mul(nn_ptr res, nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)
int mpn_mod_mul_ui(nn_ptr res, nn_srcptr x, ulong y, gr_ctx_t ctx)
int mpn_mod_mul_si(nn_ptr res, nn_srcptr x, slong y, gr_ctx_t ctx)
int mpn_mod_mul_fmpz(nn_ptr res, nn_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int mpn_mod_addmul(nn_ptr res, nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)
int mpn_mod_addmul_ui(nn_ptr res, nn_srcptr x, ulong y, gr_ctx_t ctx)
int mpn_mod_addmul_si(nn_ptr res, nn_srcptr x, slong y, gr_ctx_t ctx)
int mpn_mod_addmul_fmpz(nn_ptr res, nn_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int mpn_mod_submul(nn_ptr res, nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)
int mpn_mod_submul_ui(nn_ptr res, nn_srcptr x, ulong y, gr_ctx_t ctx)
```

```

int mpn_mod_submul_si(nn_ptr res, nn_srcptr x, slong y, gr_ctx_t ctx)
int mpn_mod_submul_fmpz(nn_ptr res, nn_srcptr x, const fmpz_t y, gr_ctx_t ctx)
int mpn_mod_fmma(nn_ptr res, nn_srcptr x1, nn_srcptr y1, nn_srcptr x2, nn_srcptr y2, gr_ctx_t
    ctx)
int mpn_mod_sqr(nn_ptr res, nn_srcptr x, gr_ctx_t ctx)
int mpn_mod_inv(nn_ptr res, nn_srcptr x, gr_ctx_t ctx)
int mpn_mod_div(nn_ptr res, nn_srcptr x, nn_srcptr y, gr_ctx_t ctx)

```

Basic functionality for the `gr` method table. These methods are interchangeable with their `gr` counterparts. For example, `mpn_mod_add(res, x, y, ctx)` is equivalent to `gr_add(res, x, y, ctx)`. The former can be slightly faster as it avoids the indirection of the method table lookup.

6.9.4 Vector functions

```

int _mpn_mod_vec_zero(nn_ptr res, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_clear(nn_ptr res, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_set(nn_ptr res, nn_srcptr x, slong len, gr_ctx_t ctx)
void _mpn_mod_vec_swap(nn_ptr vec1, nn_ptr vec2, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_neg(nn_ptr res, nn_srcptr x, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_add(nn_ptr res, nn_srcptr x, nn_srcptr y, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_sub(nn_ptr res, nn_srcptr x, nn_srcptr y, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_mul(nn_ptr res, nn_srcptr x, nn_srcptr y, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_mul_scalar(nn_ptr res, nn_srcptr x, slong len, nn_srcptr y, gr_ctx_t ctx)
int _mpn_mod_scalar_mul_vec(nn_ptr res, nn_srcptr y, nn_srcptr x, slong len, gr_ctx_t ctx)
int _mpn_mod_vec_addmul_scalar(nn_ptr res, nn_srcptr x, slong len, nn_srcptr y, gr_ctx_t ctx)
int _mpn_mod_vec_submul_scalar(nn_ptr res, nn_srcptr x, slong len, nn_srcptr y, gr_ctx_t ctx);
int _mpn_mod_vec_dot(nn_ptr res, nn_srcptr initial, int subtract, nn_srcptr vec1, nn_srcptr vec2,
    slong len, gr_ctx_t ctx)
int _mpn_mod_vec_dot_rev(nn_ptr res, nn_srcptr initial, int subtract, nn_srcptr vec1, nn_srcptr
    vec2, slong len, gr_ctx_t ctx)

```

Overrides for generic `gr` vector operations with inlined or partially inlined code for reduced overhead.

6.9.5 Matrix algorithms

All `gr_mat_t` functionality is supported by this ring. The following methods implement optimized basic operation overrides used by higher-level generic routines.

```

int mpn_mod_mat_mul_waksman(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
    Waksman's matrix multiplication algorithm using  $n^3/2 + O(n^2)$  scalar multiplications. The operations are done with delayed reduction.
int mpn_mod_mat_mul_multi_mod(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
    Reduces matrix multiplication to several nmod_mat matrix multiplications followed by CRT reconstruction. Supports multithreading.
int mpn_mod_mat_mul(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
    Dispatches among classical, Waksman and multimodular matrix multiplication according to which method is expected to perform better for the given dimensions and modulus. Strassen is currently not used as the other methods were determined to perform better.
int mpn_mod_mat_nonsingular_solve_tril(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int
    unit, gr_ctx_t ctx)

```

```
int mpn_mod_mat_nonsingular_solve_triu(gr_mat_t X, const gr_mat_t U, const gr_mat_t B, int
                                         unit, gr_ctx_t ctx)
```

Dispatches to an appropriate generic algorithm (classical or block recursive) for triangular solving.

```
int mpn_mod_mat_lu_classical_delayed(slong *res_rank, slong *P, gr_mat_t A, const gr_mat_t
                                         A_in, int rank_check, gr_ctx_t ctx)
```

Classical LU factorization with delayed modular reductions.

```
int mpn_mod_mat_lu(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check,
                    gr_ctx_t ctx)
```

Dispatches between classical, delayed-reduction and recursive LU factorization.

```
int mpn_mod_mat_det(nn_ptr res, const gr_mat_t A, gr_ctx_t ctx)
```

Dispatches to an appropriate generic algorithm for computing the determinant.

6.9.6 Polynomial algorithms

All *gr_poly_t* functionality is supported by this ring. The following methods implement optimized basic operation overrides used by higher-level generic routines.

Multiplication

All multiplication algorithms optimize for squaring.

```
int _mpn_mod_poly_mulow_classical(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,
                                   slong len2, slong len, gr_ctx_t ctx)
```

```
int _mpn_mod_poly_mulmid_classical(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,
                                    slong len2, slong nlo, slong nhi, gr_ctx_t ctx)
```

Polynomial multiplication using the schoolbook algorithm.

```
int _mpn_mod_poly_mulow_KS(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                           slong len, gr_ctx_t ctx)
```

```
int _mpn_mod_poly_mulmid_KS(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                            slong nlo, slong nhi, gr_ctx_t ctx)
```

Polynomial multiplication using Kronecker substitution (bit packing).

```
int _mpn_mod_poly_mulow_karatsuba(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,
                                   slong len2, slong len, slong cutoff, gr_ctx_t ctx)
```

```
int _mpn_mod_poly_mulmid_karatsuba(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,
                                    slong len2, slong nlo, slong nhi, slong cutoff, gr_ctx_t ctx)
```

Polynomial multiplication using the Karatsuba algorithm, implemented without intermediate modular reductions. This algorithm calls itself recursively, switching to basecase multiplication (also without intermediate reductions) when either *len1* or *len2* is smaller than *cutoff*.

Currently a full product is computed internally regardless of the output length; truncation only skips the modular reductions.

```
int _mpn_mod_poly_mulow_fft_small(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,
                                   slong len2, slong len, gr_ctx_t ctx)
```

```
int _mpn_mod_poly_mulmid_fft_small(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2,
                                    slong len2, slong nlo, slong nhi, gr_ctx_t ctx)
```

Polynomial multiplication using the small-prime FFT. Returns `GR_UNABLE` if the small-prime FFT is not available or if the coefficients are too large to use this implementation.

```
int _mpn_mod_poly_mulow(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                        slong len, gr_ctx_t ctx)
```

```
int _mpn_mod_poly_mulmid(nn_ptr res, nn_srcptr poly1, slong len1, nn_srcptr poly2, slong len2,
                        slong nlo, slong nhi, gr_ctx_t ctx)
```

Polynomial multiplication with automatic algorithm selection.

Division

```
int _mpn_mod_poly_inv_series(nn_ptr Q, nn_srcptr B, slong lenB, slong len, gr_ctx_t ctx)
int _mpn_mod_poly_div_series(nn_ptr Q, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB, slong
                            len, gr_ctx_t ctx)
```

Power series inversion and division with automatic selection between basecase and Newton algorithms.

```
int _mpn_mod_poly_divrem_basecase_preinv1(nn_ptr Q, nn_ptr R, nn_srcptr A, slong lenA,
                                          nn_srcptr B, slong lenB, nn_srcptr invL, gr_ctx_t
                                          ctx)
```

```
int _mpn_mod_poly_divrem_basecase(nn_ptr Q, nn_ptr R, nn_srcptr A, slong lenA, nn_srcptr B,
                                  slong lenB, gr_ctx_t ctx)
```

Polynomial division with remainder implemented using the basecase algorithm with delayed reductions.

```
int _mpn_mod_poly_divrem_q1_preinv1_fmma(nn_ptr Q, nn_ptr R, nn_srcptr A, slong lenA,
                                          nn_srcptr B, slong lenB, nn_srcptr invL, gr_ctx_t
                                          ctx);
```

```
int _mpn_mod_poly_divrem_q1_preinv1_fmma_precond(nn_ptr Q, nn_ptr R, nn_srcptr A, slong
                                                  lenA, nn_srcptr B, slong lenB, nn_srcptr
                                                  invL, gr_ctx_t ctx);
```

```
int _mpn_mod_poly_divrem_q1_preinv1_karatsuba_precond(nn_ptr Q, nn_ptr R, nn_srcptr A,
                                                      slong lenA, nn_srcptr B, slong lenB,
                                                      nn_srcptr invL, gr_ctx_t ctx);
```

```
int _mpn_mod_poly_divrem_q1_preinv1(nn_ptr Q, nn_ptr R, nn_srcptr A, slong lenA, nn_srcptr B,
                                     slong lenB, nn_srcptr invL, gr_ctx_t ctx);
```

Algorithms for polynomial division in the special case where $lenA = lenB + 1$. Require $lenB \geq 2$.

```
int _mpn_mod_poly_divrem(nn_ptr Q, nn_ptr R, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB,
                        gr_ctx_t ctx)
```

```
int _mpn_mod_poly_div(nn_ptr Q, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB, gr_ctx_t ctx)
```

Polynomial division with remainder with automatic selection between basecase and Newton algorithms.

GCD

```
int _mpn_mod_poly_gcd(nn_ptr G, slong *lenG, nn_srcptr A, slong lenA, nn_srcptr B, slong lenB,
                     gr_ctx_t ctx)
```

Polynomial GCD with automatic selection between basecase and HGCD algorithms.

```
int _mpn_mod_poly_xgcd(slong *lenG, nn_ptr G, nn_ptr S, nn_ptr T, nn_srcptr A, slong lenA,
                      nn_srcptr B, slong lenB, gr_ctx_t ctx);
```

Polynomial extended GCD with automatic selection between basecase and HGCD algorithms.

6.10 fmpz_mod.h – arithmetic modulo integers

6.10.1 Types, macros and constants

type `fmpz_mod_ctx_struct`

type `fmpz_mod_ctx_t`

The context object for arithmetic modulo integers.

6.10.2 Context object

void `fmpz_mod_ctx_init`(*fmpz_mod_ctx_t* ctx, const *fmpz_t* n)

Initialise ctx for arithmetic modulo n, which is expected to be positive.

void `fmpz_mod_ctx_clear`(*fmpz_mod_ctx_t* ctx)

Free any memory used by ctx.

void `fmpz_mod_ctx_set_modulus`(*fmpz_mod_ctx_t* ctx, const *fmpz_t* n)

Reconfigure ctx for arithmetic modulo n.

6.10.3 Conversions

void `fmpz_mod_set_fmpz`(*fmpz_t* a, const *fmpz_t* b, const *fmpz_mod_ctx_t* ctx)

Set a to b after reduction modulo the modulus.

6.10.4 Arithmetic

Unless specified otherwise all functions here expect their relevant arguments to be in the canonical range $[0, n)$. Comparison of elements against each other or against zero can be accomplished with `fmpz_equal()` or `fmpz_is_zero()` without a context.

int `fmpz_mod_is_canonical`(const *fmpz_t* a, const *fmpz_mod_ctx_t* ctx)

Return 1 if a is in the canonical range $[0, n)$ and 0 otherwise.

int `fmpz_mod_is_one`(const *fmpz_t* a, const *fmpz_mod_ctx_t* ctx)

Return 1 if a is 1 modulo n and return 0 otherwise.

void `fmpz_mod_add`(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)

Set a to $b + c$ modulo n.

void `fmpz_mod_add_fmpz`(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)

void `fmpz_mod_add_ui`(*fmpz_t* a, const *fmpz_t* b, *ulong* c, const *fmpz_mod_ctx_t* ctx)

void `fmpz_mod_add_si`(*fmpz_t* a, const *fmpz_t* b, *slong* c, const *fmpz_mod_ctx_t* ctx)

Set a to $b + c$ modulo n where only b is assumed to be canonical.

void `fmpz_mod_sub`(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)

Set a to $b - c$ modulo n.

void `fmpz_mod_sub_fmpz`(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)

void `fmpz_mod_sub_ui`(*fmpz_t* a, const *fmpz_t* b, *ulong* c, const *fmpz_mod_ctx_t* ctx)

void `fmpz_mod_sub_si`(*fmpz_t* a, const *fmpz_t* b, *slong* c, const *fmpz_mod_ctx_t* ctx)

Set a to $b - c$ modulo n where only b is assumed to be canonical.

void `fmpz_mod_fmpz_sub`(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)

void `fmpz_mod_ui_sub`(*fmpz_t* a, *ulong* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)

void **fmpz_mod_si_sub**(*fmpz_t* a, *slong* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)
 Set a to $b - c$ modulo n where only c is assumed to be canonical.

void **fmpz_mod_neg**(*fmpz_t* a, const *fmpz_t* b, const *fmpz_mod_ctx_t* ctx)
 Set a to $-b$ modulo n .

void **fmpz_mod_mul**(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)
 Set a to $b \cdot c$ modulo n .

void **fmpz_mod_inv**(*fmpz_t* a, const *fmpz_t* b, const *fmpz_mod_ctx_t* ctx)
 Set a to b^{-1} modulo n . This function expects that b is invertible modulo n and throws if this not the case. Invertibility may be tested with *fmpz_mod_pow_fmpz()* or *fmpz_mod_divides()*.

int **fmpz_mod_divides**(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)
 If $a \cdot c = b \pmod n$ has a solution for a return 1 and set a to such a solution. Otherwise return 0 and leave a undefined.

void **fmpz_mod_pow_ui**(*fmpz_t* a, const *fmpz_t* b, *ulong* e, const *fmpz_mod_ctx_t* ctx)
 Set a to b^e modulo n .

int **fmpz_mod_pow_fmpz**(*fmpz_t* a, const *fmpz_t* b, const *fmpz_t* e, const *fmpz_mod_ctx_t* ctx)
 Try to set a to b^e modulo n . If $e < 0$ and b is not invertible modulo n , the return is 0. Otherwise, the return is 1.

6.10.5 Discrete Logarithms via Pohlig-Hellman

void **fmpz_mod_discrete_log_pohlig_hellman_init**(*fmpz_mod_discrete_log_pohlig_hellman_t* L)
 Initialize L. Upon initialization L is not ready for computation.

void **fmpz_mod_discrete_log_pohlig_hellman_clear**(*fmpz_mod_discrete_log_pohlig_hellman_t* L)
 Free any space used by L.

double **fmpz_mod_discrete_log_pohlig_hellman_precompute_prime**(*fmpz_mod_discrete_log_pohlig_hellman_t* L, const *fmpz_t* p)
 Configure L for discrete logarithms modulo p to an internally chosen base. It is assumed that p is prime. The return is an estimate on the number of multiplications needed for one run.

const *fmpz_t* ***fmpz_mod_discrete_log_pohlig_hellman_primitive_root**(*fmpz_mod_discrete_log_pohlig_hellman_t* L)
 Return the internally stored base.

void **fmpz_mod_discrete_log_pohlig_hellman_run**(*fmpz_t* x, const *fmpz_mod_discrete_log_pohlig_hellman_t* L, const *fmpz_t* y)
 Set x to the logarithm of y with respect to the internally stored base. y is expected to be reduced modulo the p . The function is undefined if the logarithm does not exist.

int **fmpz_next_smooth_prime**(*fmpz_t* a, const *fmpz_t* b)
 Either return 1 and set a to a smooth prime strictly greater than b , or return 0 and set a to 0. The smooth primes returned by this function currently have no prime factor of $a - 1$ greater than 23, but this should not be relied upon.

6.11 fmpz_mod_vec.h – vectors over integers mod n

6.11.1 Conversions

void `_fmpz_mod_vec_set_fmpz_vec`(*fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_mod_ctx_t* ctx)
Set the *fmpz_mod_vec* (A, len) to the *fmpz_vec* (B, len) after reduction of each entry modulo the modulus..

6.11.2 Arithmetic

void `_fmpz_mod_vec_neg`(*fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_mod_ctx_t* ctx)
Set (A, len) to $-(B, len)$.

void `_fmpz_mod_vec_add`(*fmpz* *a, const *fmpz* *b, const *fmpz* *c, *slong* n, const *fmpz_mod_ctx_t* ctx)
Set (A, len) to $(B, len) + (C, len)$.

void `_fmpz_mod_vec_sub`(*fmpz* *a, const *fmpz* *b, const *fmpz* *c, *slong* n, const *fmpz_mod_ctx_t* ctx)
Set (A, len) to $(B, len) - (C, len)$.

6.11.3 Scalar Multiplication

void `_fmpz_mod_vec_scalar_mul_fmpz_mod`(*fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)
Set (A, len) to $(B, len) * c$.

void `_fmpz_mod_vec_scalar_addmul_fmpz_mod`(*fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)
Set (A, len) to $(A, len) + (B, len) * c$.

void `_fmpz_mod_vec_scalar_div_fmpz_mod`(*fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_t* c, const *fmpz_mod_ctx_t* ctx)
Set (A, len) to $(B, len)/c$ assuming c is nonzero.

6.11.4 Dot Product

void `_fmpz_mod_vec_dot`(*fmpz_t* d, const *fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_mod_ctx_t* ctx)
Set d to the dot product of (A, len) with (B, len) .

void `_fmpz_mod_vec_dot_rev`(*fmpz_t* d, const *fmpz* *A, const *fmpz* *B, *slong* len, const *fmpz_mod_ctx_t* ctx)
Set d to the dot product of (A, len) with the reverse of the vector (B, len) .

6.11.5 Multiplication

void `_fmpz_mod_vec_mul`(*fmpz* *A, const *fmpz* *B, const *fmpz* *C, *slong* len, const *fmpz_mod_ctx_t* ctx)
Set (A, len) the pointwise multiplication of (B, len) and (C, len) .

6.12 fmpz_mod_mat.h – matrices over integers mod n

Unless indicated, functions may assume that the stored values are in the range $[0, n)$. The user can manipulate matrix entries directly, but must assume responsibility for normalising all values to the range $[0, n)$.

6.12.1 Types, macros and constants

type `fmpz_mod_mat_struct`

type `fmpz_mod_mat_t`

6.12.2 Element access

fmpz *`fmpz_mod_mat_entry`(const *fmpz_mod_mat_t* mat, *slong* i, *slong* j)

Return a reference to the element at row *i* and column *j* of *mat*.

void `fmpz_mod_mat_set_entry`(*fmpz_mod_mat_t* mat, *slong* i, *slong* j, const *fmpz_t* val, const *fmpz_mod_ctx_t* ctx)

Set the entry at row *i* and column *j* of *mat* to *val*.

6.12.3 Memory management

void `fmpz_mod_mat_init`(*fmpz_mod_mat_t* mat, *slong* rows, *slong* cols, const *fmpz_mod_ctx_t* ctx)

Initialise *mat* as a matrix with the given number of *rows* and *cols* and modulus defined by *ctx*.

void `fmpz_mod_mat_init_set`(*fmpz_mod_mat_t* mat, const *fmpz_mod_mat_t* src, const *fmpz_mod_ctx_t* ctx)

Initialise *mat* and set it equal to the matrix *src*, including the number of rows and columns and the modulus.

void `fmpz_mod_mat_clear`(*fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Clear *mat* and release any memory it used.

Basic manipulation

slong `fmpz_mod_mat_nrows`(const *fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Return the number of rows of *mat*.

slong `fmpz_mod_mat_ncols`(const *fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Return the number of columns of *mat*.

void `_fmpz_mod_mat_set_mod`(*fmpz_mod_mat_t* mat, const *fmpz_t* n, const *fmpz_mod_ctx_t* ctx)

Set the modulus of the matrix *mat* to *n*.

void `fmpz_mod_mat_one`(*fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Set *mat* to the identity matrix (ones down the diagonal).

void `fmpz_mod_mat_zero`(*fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Set *mat* to the zero matrix.

void `fmpz_mod_mat_swap`(*fmpz_mod_mat_t* mat1, *fmpz_mod_mat_t* mat2, const *fmpz_mod_ctx_t* ctx)

Efficiently swap the matrices *mat1* and *mat2*.

```
void fmpz_mod_mat_swap_entrywise(fmpz_mod_mat_t mat1, fmpz_mod_mat_t mat2, const  
                                fmpz_mod_ctx_t ctx)
```

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

```
int fmpz_mod_mat_is_empty(const fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Return 1 if *mat* has either zero rows or columns.

```
int fmpz_mod_mat_is_square(const fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Return 1 if *mat* has the same number of rows and columns.

```
void _fmpz_mod_mat_reduce(fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Reduce all the entries of *mat* by the modulus *n*. This function is only needed internally.

6.12.4 Random generation

```
void fmpz_mod_mat_randtest(fmpz_mod_mat_t mat, flint_rand_t state, const fmpz_mod_ctx_t  
                           ctx)
```

Generate a random matrix with the existing dimensions and entries in $[0, n)$ where *n* is the modulus. A sparse matrix is generated with increased probability.

```
void fmpz_mod_mat_randfull(fmpz_mod_mat_t mat, flint_rand_t state, const fmpz_mod_ctx_t  
                           ctx)
```

Sets the element to random numbers in $[0, n)$, likely to be close to the modulus *n* of the matrix. This is used to test potential overflow-related bugs.

```
void fmpz_mod_mat_rand(fmpz_mod_mat_t mat, flint_rand_t state, const fmpz_mod_ctx_t ctx)
```

Sets the element to random numbers in $[0, n)$, likely to be close to the modulus *n* of the matrix.

6.12.5 Windows and concatenation

```
void fmpz_mod_mat_window_init(fmpz_mod_mat_t window, const fmpz_mod_mat_t mat, slong r1,  
                             slong c1, slong r2, slong c2, const fmpz_mod_ctx_t ctx)
```

Initializes the matrix *window* to be an $r2 - r1$ by $c2 - c1$ submatrix of *mat* whose (0, 0) entry is the (*r1*, *c1*) entry of *mat*. The memory for the elements of *window* is shared with *mat*.

```
void fmpz_mod_mat_window_clear(fmpz_mod_mat_t window, const fmpz_mod_ctx_t ctx)
```

Clears the matrix *window* and releases any memory that it uses. Note that the memory to the underlying matrix that *window* points to is not freed.

```
void fmpz_mod_mat_concat_horizontal(fmpz_mod_mat_t res, const fmpz_mod_mat_t mat1, const  
                                   fmpz_mod_mat_t mat2, const fmpz_mod_ctx_t ctx)
```

Sets *res* to vertical concatenation of (*mat1*, *mat2*) in that order. Matrix dimensions : *mat1* : $m \times n$, *mat2* : $k \times n$, *res* : $(m + k) \times n$.

```
void fmpz_mod_mat_concat_vertical(fmpz_mod_mat_t res, const fmpz_mod_mat_t mat1, const  
                                 fmpz_mod_mat_t mat2, const fmpz_mod_ctx_t ctx)
```

Sets *res* to horizontal concatenation of (*mat1*, *mat2*) in that order. Matrix dimensions : *mat1* : $m \times n$, *mat2* : $m \times k$, *res* : $m \times (n + k)$.

6.12.6 Input and output

void **fmpz_mod_mat_print_pretty**(const *fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Prints the given matrix to `stdout`. The format is an opening square bracket then on each line a row of the matrix, followed by a closing square bracket. Each row is written as an opening square bracket followed by a space separated list of coefficients followed by a closing square bracket.

6.12.7 Comparison

int **fmpz_mod_mat_equal**(const *fmpz_mod_mat_t* mat1, const *fmpz_mod_mat_t* mat2, const *fmpz_mod_ctx_t* ctx)

Returns nonzero if `mat1` and `mat2` have the same dimensions and elements, and zero otherwise. The moduli are ignored.

int **fmpz_mod_mat_is_zero**(const *fmpz_mod_mat_t* mat, const *fmpz_mod_ctx_t* ctx)

Return 1 if `mat` is the zero matrix.

6.12.8 Set and transpose

void **fmpz_mod_mat_set**(*fmpz_mod_mat_t* B, const *fmpz_mod_mat_t* A, const *fmpz_mod_ctx_t* ctx)

Set B to equal A.

void **fmpz_mod_mat_transpose**(*fmpz_mod_mat_t* B, const *fmpz_mod_mat_t* A, const *fmpz_mod_ctx_t* ctx)

Sets B to the transpose of A. Dimensions must be compatible. Aliasing is allowed for square matrices.

6.12.9 Conversions

void **fmpz_mod_mat_set_fmpz_mat**(*fmpz_mod_mat_t* A, const *fmpz_mat_t* B, const *fmpz_mod_ctx_t* ctx)

Set A to the matrix B reducing modulo the modulus of A.

void **fmpz_mod_mat_get_fmpz_mat**(*fmpz_mat_t* A, const *fmpz_mod_mat_t* B, const *fmpz_mod_ctx_t* ctx)

Set A to a lift of B.

6.12.10 Addition and subtraction

void **fmpz_mod_mat_add**(*fmpz_mod_mat_t* C, const *fmpz_mod_mat_t* A, const *fmpz_mod_mat_t* B, const *fmpz_mod_ctx_t* ctx)

Set C to $A + B$.

void **fmpz_mod_mat_sub**(*fmpz_mod_mat_t* C, const *fmpz_mod_mat_t* A, const *fmpz_mod_mat_t* B, const *fmpz_mod_ctx_t* ctx)

Set C to $A - B$.

void **fmpz_mod_mat_neg**(*fmpz_mod_mat_t* B, const *fmpz_mod_mat_t* A, const *fmpz_mod_ctx_t* ctx)

Set B to $-A$.

6.12.11 Scalar arithmetic

```
void fmpz_mod_mat_scalar_mul_si(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, slong c, const  
                                fmpz_mod_ctx_t ctx)
```

Set B to cA where c is a constant.

```
void fmpz_mod_mat_scalar_mul_ui(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, ulong c, const  
                                fmpz_mod_ctx_t ctx)
```

Set B to cA where c is a constant.

```
void fmpz_mod_mat_scalar_mul_fmpz(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, fmpz_t c,  
                                  const fmpz_mod_ctx_t ctx)
```

Set B to cA where c is a constant.

6.12.12 Matrix multiplication

```
void fmpz_mod_mat_mul(fmpz_mod_mat_t C, const fmpz_mod_mat_t A, const fmpz_mod_mat_t B,  
                     const fmpz_mod_ctx_t ctx)
```

Set C to $A \times B$. The number of rows of B must match the number of columns of A .

```
void _fmpz_mod_mat_mul_classical_threaded_pool_op(fmpz_mod_mat_t D, const  
                                                  fmpz_mod_mat_t C, const  
                                                  fmpz_mod_mat_t A, const  
                                                  fmpz_mod_mat_t B, int op,  
                                                  thread_pool_handle *threads, slong  
                                                  num_threads, const fmpz_mod_ctx_t ctx)
```

Set D to $A \times B + op \times C$ where op is $+1$, -1 or 0 .

```
void _fmpz_mod_mat_mul_classical_threaded_op(fmpz_mod_mat_t D, const fmpz_mod_mat_t C,  
                                             const fmpz_mod_mat_t A, const  
                                             fmpz_mod_mat_t B, int op, const  
                                             fmpz_mod_ctx_t ctx)
```

Set D to $A \times B + op \times C$ where op is $+1$, -1 or 0 .

```
void fmpz_mod_mat_mul_classical_threaded(fmpz_mod_mat_t C, const fmpz_mod_mat_t A,  
                                         const fmpz_mod_mat_t B, const fmpz_mod_ctx_t  
                                         ctx)
```

Set C to $A \times B$. The number of rows of B must match the number of columns of A .

```
void fmpz_mod_mat_sqr(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, const fmpz_mod_ctx_t  
                     ctx)
```

Set B to A^2 . The matrix A must be square.

```
void fmpz_mod_mat_pow_ui(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, ulong e, const  
                        fmpz_mod_ctx_t ctx)
```

Sets B to the matrix A raised to the power e , where A must be a square matrix. Aliasing is allowed.

```
void fmpz_mod_mat_mul_fmpz_vec(fmpz *c, const fmpz_mod_mat_t A, const fmpz *b, slong blen,  
                              const fmpz_mod_ctx_t ctx)
```

```
void fmpz_mod_mat_mul_fmpz_vec_ptr(fmpz *const *c, const fmpz_mod_mat_t A, const fmpz *const  
                                  *b, slong blen, const fmpz_mod_ctx_t ctx)
```

Compute a matrix-vector product of A and $(b, blen)$ and store the result in c . The vector $(b, blen)$ is either truncated or zero-extended to the number of columns of A . The number entries written to c is always equal to the number of rows of A .

```
void fmpz_mod_mat_fmpz_vec_mul(fmpz *c, const fmpz *a, slong alen, const fmpz_mod_mat_t B,  
                              const fmpz_mod_ctx_t ctx)
```



```
void fmpz_mod_mat_fmpz_vec_mul_ptr(fmpz *const *c, const fmpz *const *a, slong alen, const
    fmpz_mod_mat_t B, const fmpz_mod_ctx_t ctx)
```

Compute a vector-matrix product of $(a, alen)$ and B and store the result in c . The vector $(a, alen)$ is either truncated or zero-extended to the number of rows of B . The number entries written to c is always equal to the number of columns of B .

6.12.13 Trace

```
void fmpz_mod_mat_trace(fmpz_t trace, const fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Set `trace` to the trace of the matrix `mat`.

6.12.14 Gaussian elimination

```
void fmpz_mod_mat_det(fmpz_t res, const fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Set `res` to the determinant of the matrix `mat`.

```
slong fmpz_mod_mat_rref(fmpz_mod_mat_t res, const fmpz_mod_mat_t mat, const
    fmpz_mod_ctx_t ctx)
```

Sets `res` to the reduced row echelon form of `mat` and returns the rank.

The modulus is assumed to be prime.

6.12.15 Strong echelon form and Howell form

```
void fmpz_mod_mat_strong_echelon_form(fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Transforms `mat` into the strong echelon form of `mat`. The Howell form and the strong echelon form are equal up to permutation of the rows, see [FieHof2014] for a definition of the strong echelon form and the algorithm used here.

`mat` must have at least as many rows as columns.

```
slong fmpz_mod_mat_howell_form(fmpz_mod_mat_t mat, const fmpz_mod_ctx_t ctx)
```

Transforms `mat` into the Howell form of `mat`. For a definition of the Howell form see [StoMul1998]. The Howell form is computed by first putting `mat` into strong echelon form and then ordering the rows.

`mat` must have at least as many rows as columns.

6.12.16 Inverse

```
int fmpz_mod_mat_inv(fmpz_mod_mat_t B, const fmpz_mod_mat_t A, const fmpz_mod_ctx_t ctx)
```

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of B to undefined values.

A and B must be square matrices with the same dimensions.

The modulus is assumed to be prime.

6.12.17 LU decomposition

slong **fmpz_mod_mat_lu**(*slong* *P, *fmpz_mod_mat_t* A, int rank_check, const *fmpz_mod_ctx_t* ctx)

Computes a generalised LU decomposition $PLU = A$ of a given matrix A , returning the rank of A .

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for **rank_check** is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

The modulus is assumed to be prime.

6.12.18 Triangular solving

void **fmpz_mod_mat_solve_tril**(*fmpz_mod_mat_t* X, const *fmpz_mod_mat_t* L, const *fmpz_mod_mat_t* B, int unit, const *fmpz_mod_ctx_t* ctx)

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If **unit** = 1, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

The modulus is assumed to be prime.

void **fmpz_mod_mat_solve_triu**(*fmpz_mod_mat_t* X, const *fmpz_mod_mat_t* U, const *fmpz_mod_mat_t* B, int unit, const *fmpz_mod_ctx_t* ctx)

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If **unit** = 1, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

The modulus is assumed to be prime.

6.12.19 Solving

int **fmpz_mod_mat_solve**(*fmpz_mod_mat_t* X, const *fmpz_mod_mat_t* A, const *fmpz_mod_mat_t* B, const *fmpz_mod_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

The matrix A must be square.

The modulus is assumed to be prime.

int **fmpz_mod_mat_can_solve**(*fmpz_mod_mat_t* X, const *fmpz_mod_mat_t* A, const *fmpz_mod_mat_t* B, const *fmpz_mod_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$ over Fp .

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of X to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of A and it may be singular.

The modulus is assumed to be prime.

6.12.20 Transforms

void **fmpz_mod_mat_similarity**(*fmpz_mod_mat_t* M, *slong* r, *fmpz_t* d, const *fmpz_mod_ctx_t* ctx)

Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value d is required to be reduced modulo the modulus of the entries in the matrix.

The modulus is assumed to be prime.

6.12.21 Characteristic polynomial

void **fmpz_mod_mat_charpoly**(*fmpz_mod_poly_t* p, const *fmpz_mod_mat_t* M, const *fmpz_mod_ctx_t* ctx)

Compute the characteristic polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

6.12.22 Minimal polynomial

void **fmpz_mod_mat_minpoly**(*fmpz_mod_poly_t* p, const *fmpz_mod_mat_t* M, const *fmpz_mod_ctx_t* ctx)

Compute the minimal polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

The modulus is assumed to be prime.

6.13 fmpz_mod_poly.h – polynomials over integers mod n

The *fmpz_mod_poly_t* data type represents elements of $\mathbb{Z}/n\mathbb{Z}[x]$ for a fixed modulus n . The **fmpz_mod_poly** module provides routines for memory management, basic arithmetic and some higher level functions such as GCD, etc.

Each coefficient of an *fmpz_mod_poly_t* is of type *fmpz* and represents an integer reduced modulo the fixed modulus n in the range $[0, n)$.

Unless otherwise specified, all functions in this section permit aliasing between their input arguments and between their input and output arguments.

The *fmpz_mod_poly_t* type is a typedef for an array of length 1 of *fmpz_mod_poly_struct*'s. This permits passing parameters of type *fmpz_mod_poly_t* by reference.

In reality one never deals directly with the **struct** and simply deals with objects of type *fmpz_mod_poly_t*. For simplicity we will think of an *fmpz_mod_poly_t* as a **struct**, though in practice to access fields of this **struct**, one needs to dereference first, e.g. to access the **length** field of an *fmpz_mod_poly_t* called **poly1** one writes **poly1->length**.

An *fmpz_mod_poly_t* is said to be *normalised* if either **length** is zero, or if the leading coefficient of the polynomial is non-zero. All **fmpz_mod_poly** functions expect their inputs to be normalised and all coefficients to be reduced modulo n , and unless otherwise specified they produce output that is normalised with coefficients reduced modulo n .

6.13.1 Simple example

The following example computes the square of the polynomial $5x^3 + 6$ in $\mathbb{Z}/7\mathbb{Z}[x]$.

```
#include "flint/fmpz_mod.h"
#include "flint/fmpz_mod_poly.h"
int main(void)
{
    fmpz_mod_ctx_t ctx;
    fmpz_mod_poly_t x, y;

    fmpz_mod_ctx_init_ui(ctx, 7);
    fmpz_mod_poly_init(x, ctx);
    fmpz_mod_poly_init(y, ctx);
    fmpz_mod_poly_set_coeff_ui(x, 3, 5, ctx);
    fmpz_mod_poly_set_coeff_ui(x, 0, 6, ctx);
    fmpz_mod_poly_sqr(y, x, ctx);

    flint_printf("x = {%fmpz_mod_poly} ({%fmpz_mod_ctx})\n"
                "x^2 = {%fmpz_mod_poly} ({%fmpz_mod_ctx})\n",
                x, ctx, y, ctx);

    fmpz_mod_poly_clear(x, ctx);
    fmpz_mod_poly_clear(y, ctx);
    fmpz_mod_ctx_clear(ctx);
}
```

The output is:

```
4 7 6 0 0 5
7 7 1 0 0 4 0 0 4
```

6.13.2 Types, macros and constants

type **fmpz_mod_poly_struct**

A structure holding a polynomial over the integers modulo n .

type **fmpz_mod_poly_t**

An array of length 1 of **fmpz_mod_poly_struct**.

6.13.3 Memory management

void **fmpz_mod_poly_init**(*fmpz_mod_poly_t* poly, const *fmpz_mod_ctx_t* ctx)

Initialises *poly* for use with context *ctx* and set it to zero. A corresponding call to *fmpz_mod_poly_clear()* must be made to free the memory used by the polynomial.

void **fmpz_mod_poly_init2**(*fmpz_mod_poly_t* poly, *slong* alloc, const *fmpz_mod_ctx_t* ctx)

Initialises *poly* with space for at least *alloc* coefficients and sets the length to zero. The allocated coefficients are all set to zero.

void **fmpz_mod_poly_clear**(*fmpz_mod_poly_t* poly, const *fmpz_mod_ctx_t* ctx)

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void **fmpr_mod_poly_realloc**(*fmpr_mod_poly_t* poly, *slong* alloc, const *fmpr_mod_ctx_t* ctx)

Reallocates the given polynomial to have space for *alloc* coefficients. If *alloc* is zero the polynomial is cleared and then reinitialised. If the current length is greater than *alloc* the polynomial is first truncated to length *alloc*.

void **fmpr_mod_poly_fit_length**(*fmpr_mod_poly_t* poly, *slong* len, const *fmpr_mod_ctx_t* ctx)

If *len* is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least *len* coefficients. No data is lost when calling this function.

The function efficiently deals with the case where it is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void **_fmpr_mod_poly_normalise**(*fmpr_mod_poly_t* poly)

Sets the length of *poly* so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **_fmpr_mod_poly_set_length**(*fmpr_mod_poly_t* poly, *slong* len)

Demotes the coefficients of *poly* beyond *len* and sets the length of *poly* to *len*.

void **fmpr_mod_poly_truncate**(*fmpr_mod_poly_t* poly, *slong* len, const *fmpr_mod_ctx_t* ctx)

If the current length of *poly* is greater than *len*, it is truncated to have the given length. Discarded coefficients are not necessarily set to zero.

void **fmpr_mod_poly_set_trunc**(*fmpr_mod_poly_t* res, const *fmpr_mod_poly_t* poly, *slong* n, const *fmpr_mod_ctx_t* ctx)

Notionally truncate *poly* to length *n* and set *res* to the result. The result is normalised.

6.13.4 Randomisation

void **fmpr_mod_poly_randtest**(*fmpr_mod_poly_t* f, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Sets the polynomial~`f` to a random polynomial of length up~``len``.

void **fmpr_mod_poly_randtest_irreducible**(*fmpr_mod_poly_t* f, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Sets the polynomial~`f` to a random irreducible polynomial of length up~``len``, assuming *len* is positive.

void **fmpr_mod_poly_randtest_not_zero**(*fmpr_mod_poly_t* f, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Sets the polynomial~`f` to a random polynomial of length up~``len``, assuming *len* is positive.

void **fmpr_mod_poly_randtest_monic**(*fmpr_mod_poly_t* poly, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Generates a random monic polynomial with length *len*.

void **fmpr_mod_poly_randtest_monic_irreducible**(*fmpr_mod_poly_t* poly, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Generates a random monic irreducible polynomial with length *len*.

void **fmpr_mod_poly_randtest_monic_primitive**(*fmpr_mod_poly_t* poly, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Generates a random monic irreducible primitive polynomial with length *len*.

void **fmpr_mod_poly_randtest_trinomial**(*fmpr_mod_poly_t* poly, *flint_rand_t* state, *slong* len, const *fmpr_mod_ctx_t* ctx)

Generates a random monic trinomial of length *len*.

```
int fmpz_mod_poly_randtest_trinomial_irreducible(fmpz_mod_poly_t poly, flint_rand_t state,
                                                slong len, slong max_attempts, const
                                                fmpz_mod_ctx_t ctx)
```

Attempts to set `poly` to a monic irreducible trinomial of length `len`. It will generate up to `max_attempts` trinomials in attempt to find an irreducible one. If `max_attempts` is 0, then it will keep generating trinomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

```
void fmpz_mod_poly_randtest_pentomial(fmpz_mod_poly_t poly, flint_rand_t state, slong len,
                                      const fmpz_mod_ctx_t ctx)
```

Generates a random monic pentomial of length `len`.

```
int fmpz_mod_poly_randtest_pentomial_irreducible(fmpz_mod_poly_t poly, flint_rand_t state,
                                                slong len, slong max_attempts, const
                                                fmpz_mod_ctx_t ctx)
```

Attempts to set `poly` to a monic irreducible pentomial of length `len`. It will generate up to `max_attempts` pentomials in attempt to find an irreducible one. If `max_attempts` is 0, then it will keep generating pentomials until an irreducible one is found. Returns 1 if one is found and 0 otherwise.

```
void fmpz_mod_poly_randtest_sparse_irreducible(fmpz_mod_poly_t poly, flint_rand_t state,
                                              slong len, const fmpz_mod_ctx_t ctx)
```

Attempts to set `poly` to a sparse, monic irreducible polynomial with length `len`. It attempts to find an irreducible trinomial. If that does not succeed, it attempts to find an irreducible pentomial. If that fails, then `poly` is just set to a random monic irreducible polynomial.

6.13.5 Attributes

```
slong fmpz_mod_poly_degree(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Returns the degree of the polynomial. The degree of the zero polynomial is defined to be -1 .

```
slong fmpz_mod_poly_length(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Returns the length of the polynomial, which is one more than its degree.

```
fmpz *fmpz_mod_poly_lead(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Returns a pointer to the first leading coefficient of `poly` if this is non-zero, otherwise returns `NULL`.

6.13.6 Assignment and basic manipulation

```
void fmpz_mod_poly_set(fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const
                     fmpz_mod_ctx_t ctx)
```

Sets the polynomial `poly1` to the value of `poly2`.

```
void fmpz_mod_poly_swap(fmpz_mod_poly_t poly1, fmpz_mod_poly_t poly2, const
                      fmpz_mod_ctx_t ctx)
```

Swaps the two polynomials. This is done efficiently by swapping pointers rather than individual coefficients.

```
void fmpz_mod_poly_zero(fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Sets `poly` to the zero polynomial.

```
void fmpz_mod_poly_one(fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Sets `poly` to the constant polynomial 1.

```
void fmpz_mod_poly_zero_coeffs(fmpz_mod_poly_t poly, slong i, slong j, const fmpz_mod_ctx_t
                             ctx)
```

Sets the coefficients of X^k for $k \in [i, j)$ in the polynomial to zero.

```
void fmpz_mod_poly_reverse(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, slong n, const
                           fmpz_mod_ctx_t ctx)
```

This function considers the polynomial `poly` to be of length n , notionally truncating and zero padding if required, and reverses the result. Since the function normalises its result `res` may be of length less than n .

6.13.7 Conversion

```
void fmpz_mod_poly_set_ui(fmpz_mod_poly_t f, ulong c, const fmpz_mod_ctx_t ctx)
```

Sets the polynomial f to the constant c reduced modulo p .

```
void fmpz_mod_poly_set_fmpz(fmpz_mod_poly_t f, const fmpz_t c, const fmpz_mod_ctx_t ctx)
```

Sets the polynomial f to the constant c reduced modulo p .

```
void fmpz_mod_poly_set_fmpz_poly(fmpz_mod_poly_t f, const fmpz_poly_t g, const
                                 fmpz_mod_ctx_t ctx)
```

Sets f to g reduced modulo p , where p is the modulus that is part of the data structure of f .

```
void fmpz_mod_poly_get_fmpz_poly(fmpz_poly_t f, const fmpz_mod_poly_t g, const
                                 fmpz_mod_ctx_t ctx)
```

Sets f to g . This is done simply by lifting the coefficients of g taking representatives $[0, p) \subset \mathbf{Z}$.

```
void fmpz_mod_poly_get_nmod_poly(nmod_poly_t f, const fmpz_mod_poly_t g)
```

Sets f to g assuming the modulus of both polynomials is the same (no checking is performed).

```
void fmpz_mod_poly_set_nmod_poly(fmpz_mod_poly_t f, const nmod_poly_t g)
```

Sets f to g assuming the modulus of both polynomials is the same (no checking is performed).

6.13.8 Comparison

```
int fmpz_mod_poly_equal(const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2, const
                        fmpz_mod_ctx_t ctx)
```

Returns non-zero if the two polynomials are equal, otherwise returns zero.

```
int fmpz_mod_poly_equal_trunc(const fmpz_mod_poly_t poly1, const fmpz_mod_poly_t poly2,
                              slong n, const fmpz_mod_ctx_t ctx)
```

Notionally truncates the two polynomials to length n and returns non-zero if the two polynomials are equal, otherwise returns zero.

```
int fmpz_mod_poly_is_zero(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Returns non-zero if the polynomial is zero.

```
int fmpz_mod_poly_is_one(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Returns non-zero if the polynomial is the constant 1.

```
int fmpz_mod_poly_is_gen(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Returns non-zero if the polynomial is the degree 1 polynomial x .

6.13.9 Getting and setting coefficients

void `fmpz_mod_poly_set_coeff_fmpz`(*fmpz_mod_poly_t* poly, *slong* n, const *fmpz_t* x, const *fmpz_mod_ctx_t* ctx)

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

void `fmpz_mod_poly_set_coeff_ui`(*fmpz_mod_poly_t* poly, *slong* n, *ulong* x, const *fmpz_mod_ctx_t* ctx)

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

void `fmpz_mod_poly_get_coeff_fmpz`(*fmpz_t* x, const *fmpz_mod_poly_t* poly, *slong* n, const *fmpz_mod_ctx_t* ctx)

Sets x to the coefficient of X^n in the polynomial, assuming $n \geq 0$.

void `fmpz_mod_poly_set_coeff_mpz`(*fmpz_mod_poly_t* poly, *slong* n, const *mpz_t* x, const *fmpz_mod_ctx_t* ctx)

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

void `fmpz_mod_poly_get_coeff_mpz`(*mpz_t* x, const *fmpz_mod_poly_t* poly, *slong* n, const *fmpz_mod_ctx_t* ctx)

Sets x to the coefficient of X^n in the polynomial, assuming $n \geq 0$.

6.13.10 Shifting

void `_fmpz_mod_poly_shift_left`(*fmpz_t* *res, const *fmpz_t* *poly, *slong* len, *slong* n)

Sets (res, len + n) to (poly, len) shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that len and n are positive, and that res fits len + n elements. Supports aliasing between res and poly.

void `fmpz_mod_poly_shift_left`(*fmpz_mod_poly_t* f, const *fmpz_mod_poly_t* g, *slong* n, const *fmpz_mod_ctx_t* ctx)

Sets res to poly shifted left by n coeffs. Zero coefficients are inserted.

void `_fmpz_mod_poly_shift_right`(*fmpz_t* *res, const *fmpz_t* *poly, *slong* len, *slong* n)

Sets (res, len - n) to (poly, len) shifted right by n coefficients.

Assumes that len and n are positive, that len > n , and that res fits len - n elements. Supports aliasing between res and poly, although in this case the top coefficients of poly are not set to zero.

void `fmpz_mod_poly_shift_right`(*fmpz_mod_poly_t* f, const *fmpz_mod_poly_t* g, *slong* n, const *fmpz_mod_ctx_t* ctx)

Sets res to poly shifted right by n coefficients. If n is equal to or greater than the current length of poly, res is set to the zero polynomial.

6.13.11 Addition and subtraction

void `_fmpz_mod_poly_add`(*fmpz_t* *res, const *fmpz_t* *poly1, *slong* len1, const *fmpz_t* *poly2, *slong* len2, const *fmpz_mod_ctx_t* ctx)

Sets res to the sum of (poly1, len1) and (poly2, len2). It is assumed that res has sufficient space for the longer of the two polynomials.

void `fmpz_mod_poly_add`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly1, const *fmpz_mod_poly_t* poly2, const *fmpz_mod_ctx_t* ctx)

Sets res to the sum of poly1 and poly2.

```
void fmpz_mod_poly_add_series(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const
                             fmpz_mod_poly_t poly2, slong n, const fmpz_mod_ctx_t ctx)
```

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the sum.

```
void _fmpz_mod_poly_sub(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong len2,
                       const fmpz_mod_ctx_t ctx)
```

Sets `res` to $(\text{poly1}, \text{len1})$ minus $(\text{poly2}, \text{len2})$. It is assumed that `res` has sufficient space for the longer of the two polynomials.

```
void fmpz_mod_poly_sub(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const
                      fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly1` minus `poly2`.

```
void fmpz_mod_poly_sub_series(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const
                              fmpz_mod_poly_t poly2, slong n, const fmpz_mod_ctx_t ctx)
```

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the difference.

```
void _fmpz_mod_poly_neg(fmpz *res, const fmpz *poly, slong len, const fmpz_mod_ctx_t ctx)
```

Sets (res, len) to the negative of $(\text{poly}, \text{len})$ modulo p .

```
void fmpz_mod_poly_neg(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const
                      fmpz_mod_ctx_t ctx)
```

Sets `res` to the negative of `poly` modulo p .

6.13.12 Scalar multiplication and division

```
void _fmpz_mod_poly_scalar_mul_fmpz(fmpz *res, const fmpz *poly, slong len, const fmpz_t x, const
                                    fmpz_mod_ctx_t ctx)
```

```
void _fmpz_mod_poly_scalar_mul_ui(fmpz *res, const fmpz *poly, slong len, ulong x, const
                                  fmpz_mod_ctx_t ctx)
```

Sets (res, len) to $(\text{poly}, \text{len})$ multiplied by x , reduced modulo p .

```
void fmpz_mod_poly_scalar_mul_fmpz(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const
                                   fmpz_t x, const fmpz_mod_ctx_t ctx)
```

```
void fmpz_mod_poly_scalar_mul_ui(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong x,
                                 const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` multiplied by x .

```
void fmpz_mod_poly_scalar_addmul_fmpz(fmpz_mod_poly_t rop, const fmpz_mod_poly_t op, const
                                      fmpz_t x, const fmpz_mod_ctx_t ctx)
```

Adds to `rop` the product of `op` by the scalar x .

```
void _fmpz_mod_poly_scalar_div_fmpz(fmpz *res, const fmpz *poly, slong len, const fmpz_t x, const
                                    fmpz_mod_ctx_t ctx)
```

Sets (res, len) to $(\text{poly}, \text{len})$ divided by x (i.e. multiplied by the inverse of $x \pmod{p}$). The result is reduced modulo p .

```
void fmpz_mod_poly_scalar_div_fmpz(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const
                                   fmpz_t x, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` divided by x , (i.e. multiplied by the inverse of $x \pmod{p}$). The result is reduced modulo p .

6.13.13 Multiplication

void `_fmpz_mod_poly_mul`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2, const *fmpz_mod_ctx_t* ctx)

Sets (res, len1 + len2 - 1) to the product of (poly1, len1) and (poly2, len2). Assumes len1 >= len2 > 0. Allows zero-padding of the two input polynomials.

void `fmpz_mod_poly_mul`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly1, const *fmpz_mod_poly_t* poly2, const *fmpz_mod_ctx_t* ctx)

Sets res to the product of poly1 and poly2.

void `_fmpz_mod_poly_mullo`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2, *slong* n, const *fmpz_mod_ctx_t* ctx)

Sets (res, n) to the lowest n coefficients of the product of (poly1, len1) and (poly2, len2).

Assumes len1 >= len2 > 0 and 0 < n <= len1 + len2 - 1. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

void `fmpz_mod_poly_mullo`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly1, const *fmpz_mod_poly_t* poly2, *slong* n, const *fmpz_mod_ctx_t* ctx)

Sets res to the lowest n coefficients of the product of poly1 and poly2.

void `_fmpz_mod_poly_mulmid`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2, *slong* nlo, *slong* nhi, const *fmpz_mod_ctx_t* ctx)

Sets (res, nhi - nlo) to the coefficients at indices [nlo, nhi) in the full product of (poly1, len1) and (poly2, len2). Assumes that len1 and len2 are positive and that 0 ≤ nlo < nhi ≤ len1 + len2 - 1. Does not support aliasing between the inputs and the output.

void `fmpz_mod_poly_mulmid`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly1, const *fmpz_mod_poly_t* poly2, *slong* nlo, *slong* nhi, const *fmpz_mod_ctx_t* ctx)

Sets res to the polynomial formed by the coefficients at indices [nlo, nhi) in the product of poly1 and poly2. Equivalently, compute $[(poly1 \cdot poly2) \bmod x^{nhi}] / x^{nlo}$.

void `_fmpz_mod_poly_sqr`(*fmpz* *res, const *fmpz* *poly, *slong* len, const *fmpz_mod_ctx_t* ctx)

Sets res to the square of poly.

void `fmpz_mod_poly_sqr`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly, const *fmpz_mod_ctx_t* ctx)

Computes res as the square of poly.

void `fmpz_mod_poly_mulhigh`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly1, const *fmpz_mod_poly_t* poly2, *slong* start, const *fmpz_mod_ctx_t* ctx)

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary.

void `_fmpz_mod_poly_mulmod`(*fmpz* *res, const *fmpz* *poly1, *slong* len1, const *fmpz* *poly2, *slong* len2, const *fmpz* *f, *slong* lenf, const *fmpz_mod_ctx_t* ctx)

Sets res, len1 + len2 - 1 to the remainder of the product of poly1 and poly2 upon polynomial division by f.

It is required that len1 + len2 - lenf > 0, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fmpz_mod_poly_mul` instead.

Aliasing of f and res is not permitted.

void `fmpz_mod_poly_mulmod`(*fmpz_mod_poly_t* res, const *fmpz_mod_poly_t* poly1, const *fmpz_mod_poly_t* poly2, const *fmpz_mod_poly_t* f, const *fmpz_mod_ctx_t* ctx)

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f.

```
void _fmpz_mod_poly_mulmod_preinv(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2,
                                slong len2, const fmpz *f, slong lenf, const fmpz *finv, slong
                                lenfinv, const fmpz_mod_ctx_t ctx)
```

Sets `res`, `len1 + len2 - 1` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f mod xlenf`. It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. It is required that `len1 < lenf` and `len2 < lenf`. Otherwise, simply use `_fmpz_mod_poly_mul` instead.

Aliasing of `f` or `finv` and `res` is not permitted.

```
void fmpz_mod_poly_mulmod_preinv(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const
                                fmpz_mod_poly_t poly2, const fmpz_mod_poly_t f, const
                                fmpz_mod_poly_t finv, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`. It is required that `poly1` and `poly2` are reduced modulo `f`.

6.13.14 Products

```
void _fmpz_mod_poly_product_roots_fmpz_vec(fmpz *poly, const fmpz *xs, slong n, const
                                           fmpz_mod_ctx_t ctx)
```

Sets `(poly, n + 1)` to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by `xs`. It is required that the roots are canonical.

Aliasing of the input and output is not allowed.

```
void fmpz_mod_poly_product_roots_fmpz_vec(fmpz_mod_poly_t poly, const fmpz *xs, slong n,
                                           const fmpz_mod_ctx_t ctx)
```

Sets `poly` to the monic polynomial which is the product of $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$, the roots x_i being given by `xs`. It is required that the roots are canonical.

```
int fmpz_mod_poly_find_distinct_nonzero_roots(fmpz *roots, const fmpz_mod_poly_t A, const
                                              fmpz_mod_ctx_t ctx)
```

If `A` has `deg(A)` distinct nonzero roots in $\mathbb{F}_p$, write these roots out to `roots[0]` to `roots[deg(A) - 1]` and return 1. Otherwise, return 0. It is assumed that `A` is nonzero and that the modulus of `A` is prime. This function uses Rabin's probabilistic method via gcd's with $(x + \delta)^{\frac{p-1}{2}} - 1$.

Powering

```
void _fmpz_mod_poly_pow(fmpz *rop, const fmpz *op, slong len, ulong e, const fmpz_mod_ctx_t ctx)
    Sets rop = polye, assuming that  $e > 1$  and elen > 0, and that res has space for  $e * (\text{len} - 1) + 1$  coefficients. Does not support aliasing.
```

```
void fmpz_mod_poly_pow(fmpz_mod_poly_t rop, const fmpz_mod_poly_t op, ulong e, const
                      fmpz_mod_ctx_t ctx)
```

Computes `rop = polye`. If `e` is zero, returns one, so that in particular `00 = 1`.

```
void _fmpz_mod_poly_pow_trunc(fmpz *res, const fmpz *poly, ulong e, slong trunc, const
                             fmpz_mod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that $e > 1$. Aliasing is not permitted.

```
void fmpz_mod_poly_pow_trunc(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong e, slong
                           trunc, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.

```
void _fmpz_mod_poly_pow_trunc_binexp(fmpz *res, const fmpz *poly, ulong e, slong trunc, const
                                   fmpz_mod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void fmpz_mod_poly_pow_trunc_binexp(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong
                                   e, slong trunc, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```
void _fmpz_mod_poly_powmod_ui_binexp(fmpz *res, const fmpz *poly, ulong e, const fmpz *f, slong
                                   lenf, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_ui_binexp(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, ulong
                                   e, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fmpz_mod_poly_powmod_ui_binexp_preinv(fmpz *res, const fmpz *poly, ulong e, const fmpz *f,
                                             slong lenf, const fmpz *finv, slong lenfinv, const
                                             fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_ui_binexp_preinv(fmpz_mod_poly_t res, const fmpz_mod_poly_t
                                           poly, ulong e, const fmpz_mod_poly_t f, const
                                           fmpz_mod_poly_t finv, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fmpz_mod_poly_powmod_fmpz_binexp(fmpz *res, const fmpz *poly, const fmpz_t e, const fmpz
                                       *f, slong lenf, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_fmpz_binexp(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly,
                                       const fmpz_t e, const fmpz_mod_poly_t f, const
                                       fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fmpz_mod_poly_powmod_fmpz_binexp_preinv(fmpz *res, const fmpz *poly, const fmpz_t e,
                                              const fmpz *f, slong lenf, const fmpz *finv, slong
                                              lenfinv, const fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_fmpz_binexp_preinv(fmpz_mod_poly_t res, const fmpz_mod_poly_t
                                              poly, const fmpz_t e, const fmpz_mod_poly_t f,
                                              const fmpz_mod_poly_t finv, const
                                              fmpz_mod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fmpz_mod_poly_powmod_x_fmpz_preinv(fmpz *res, const fmpz_t e, const fmpz *f, slong lenf,
                                          const fmpz *finv, slong lenfinv, const fmpz_mod_ctx_t
                                          ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fmpz_mod_poly_powmod_x_fmpz_preinv(fmpz_mod_poly_t res, const fmpz_t e, const
                                          fmpz_mod_poly_t f, const fmpz_mod_poly_t finv,
                                          const fmpz_mod_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of ``

```
void _fmpz_mod_poly_powers_mod_preinv_naive(fmpz **res, const fmpz *f, slong flen, slong n, const
                                             fmpz *g, slong glen, const fmpz *ginv, slong
                                             ginvlen, const fmpz_mod_ctx_t ctx)
```

Compute $f^0, f^1, \dots, f^{(n-1)} \bmod g$, where `g` has length `glen` and `f` is reduced mod `g` and has length `flen` (possibly zero spaced). Assumes `res` is an array of `n` arrays each with space for at least `glen - 1` coefficients and that `flen > 0`. We require that `ginv` of length `ginvlen` is set to the power series inverse of the reverse of `g`.

```
void fmpz_mod_poly_powers_mod_naive(fmpz_mod_poly_struct *res, const fmpz_mod_poly_t f,
                                     slong n, const fmpz_mod_poly_t g, const fmpz_mod_ctx_t
                                     ctx)
```

Set the entries of the array `res` to $f^0, f^1, \dots, f^{(n-1)} \bmod g$. No aliasing is permitted between the entries of `res` and either of the inputs.

```
void _fmpz_mod_poly_powers_mod_preinv_threaded_pool(fmpz **res, const fmpz *f, slong flen,
                                                    slong n, const fmpz *g, slong glen, const
                                                    fmpz *ginv, slong ginvlen, const
                                                    fmpz_mod_ctx_t p, thread_pool_handle
                                                    *threads, slong num_threads)
```

Compute $f^0, f^1, \dots, f^{(n-1)} \bmod g$, where `g` has length `glen` and `f` is reduced mod `g` and has length `flen` (possibly zero spaced). Assumes `res` is an array of `n` arrays each with space for at least `glen - 1` coefficients and that `flen > 0`. We require that `ginv` of length `ginvlen` is set to the power series inverse of the reverse of `g`.

```
void fmpz_mod_poly_powers_mod_bsigs(fmpz_mod_poly_struct *res, const fmpz_mod_poly_t f, slong
                                     n, const fmpz_mod_poly_t g, const fmpz_mod_ctx_t ctx)
```

Set the entries of the array `res` to $f^0, f^1, \dots, f^{(n-1)} \bmod g$. No aliasing is permitted between the entries of `res` and either of the inputs.


```
void fmpz_mod_poly_frobenius_powers_2exp_precomp(fmpz_mod_poly_frobenius_powers_2exp_t
                                                  pow, const fmpz_mod_poly_t f, const
                                                  fmpz_mod_poly_t finv, ulong m, const
                                                  fmpz_mod_ctx_t ctx)
```

If $p = f \rightarrow p$, compute $x^{(p^1)}, x^{(p^2)}, x^{(p^4)}, \dots, x^{(p^{2^l})} \pmod{f}$ where 2^l is the greatest power of 2 less than or equal to m .

Allows construction of $x^{(p^k)}$ for $k = 0, 1, \dots, x^{(p^m)} \pmod{f}$ using `fmpz_mod_poly_frobenius_power()`.

Requires precomputed inverse of f , i.e. newton inverse.

```
void fmpz_mod_poly_frobenius_powers_2exp_clear(fmpz_mod_poly_frobenius_powers_2exp_t
                                                pow, const fmpz_mod_ctx_t ctx)
```

Clear resources used by the `fmpz_mod_poly_frobenius_powers_2exp_t` struct.

```
void fmpz_mod_poly_frobenius_power(fmpz_mod_poly_t res,
                                   fmpz_mod_poly_frobenius_powers_2exp_t pow, const
                                   fmpz_mod_poly_t f, ulong m, const fmpz_mod_ctx_t ctx)
```

If $p = f \rightarrow p$, compute $x^{(p^m)} \pmod{f}$.

Requires precomputed frobenius powers supplied by `fmpz_mod_poly_frobenius_powers_2exp_precomp`.

If $m == 0$ and f has degree 0 or 1, this performs a division. However an impossible inverse by the leading coefficient of f will have been caught by `fmpz_mod_poly_frobenius_powers_2exp_precomp`.

```
void fmpz_mod_poly_frobenius_powers_precomp(fmpz_mod_poly_frobenius_powers_t pow, const
                                              fmpz_mod_poly_t f, const fmpz_mod_poly_t finv,
                                              ulong m, const fmpz_mod_ctx_t ctx)
```

If $p = f \rightarrow p$, compute $x^{(p^0)}, x^{(p^1)}, x^{(p^2)}, x^{(p^3)}, \dots, x^{(p^m)} \pmod{f}$.

Requires precomputed inverse of f , i.e. newton inverse.

```
void fmpz_mod_poly_frobenius_powers_clear(fmpz_mod_poly_frobenius_powers_t pow, const
                                           fmpz_mod_ctx_t ctx)
```

Clear resources used by the `fmpz_mod_poly_frobenius_powers_t` struct.

6.13.15 Division

```
void _fmpz_mod_poly_divrem_basecase(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const fmpz
                                   *B, slong lenB, const fmpz_t invB, const fmpz_mod_ctx_t
                                   ctx)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1), (R, \text{lenA})$ such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible modulo p , and that `invB` is the inverse.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_divrem_basecase(fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const
                                   fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
                                   fmpz_mod_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible modulo p .

```
void _fmpz_mod_poly_divrem_newton_n_preinv(fmpz *Q, fmpz *R, const fmpz *A, slong lenA, const
                                           fmpz *B, slong lenB, const fmpz *Binv, slong
                                           lenBinv, const fmpz_mod_ctx_t ctx)
```


Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

```
void fmpz_mod_poly_divrem_newton_n_preinv(fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const
                                         fmpz_mod_poly_t A, const fmpz_mod_poly_t B,
                                         const fmpz_mod_poly_t Binv, const
                                         fmpz_mod_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \times \text{length of } B - 2$.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

```
void _fmpz_mod_poly_div_newton_n_preinv(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B,
                                       slong lenB, const fmpz *Binv, slong lenBinv, const
                                       fmpz_mod_ctx_t ctx)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but return only Q .

We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fmpz_mod_poly_div_newton_n_preinv(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const
                                       fmpz_mod_poly_t B, const fmpz_mod_poly_t Binv,
                                       const fmpz_mod_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q .

We assume that the leading coefficient of B is a unit and that B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \times \text{length of } B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
ulong fmpz_mod_poly_remove(fmpz_mod_poly_t f, const fmpz_mod_poly_t g, const
                           fmpz_mod_ctx_t ctx)
```

Removes the highest possible power of g from f and returns the exponent.

```
void _fmpz_mod_poly_rem_basecase(fmpz *R, const fmpz *A, slong lenA, const fmpz *B, slong lenB,
                                const fmpz_t invB, const fmpz_mod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(R, \text{len}B - 1)$.

Allows aliasing only between A and R . Allows zero-padding in A but not in B . Assumes that the leading coefficient of B is a unit modulo p .

```
void fmpz_mod_poly_rem_basecase(fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const
                                fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ assuming that the leading term of B is a unit.

```
void _fmpz_mod_poly_div(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const
                       fmpz_t invB, const fmpz_mod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(Q, \text{len}A - \text{len}B + 1)$.

Assumes that the leading coefficient of B is a unit modulo p .

```
void fmpz_mod_poly_div(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ assuming that the leading term of B is a unit.

```
void _fmpz_mod_poly_divrem(fmpz_t *Q, fmpz_t *R, const fmpz_t *A, slong lenA, const fmpz_t *B, slong lenB, const fmpz_t invB, const fmpz_mod_ctx_t ctx)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1), (R, \text{lenB} - 1)$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that B is non-zero, that the leading coefficient of B is invertible modulo p and that invB is the inverse.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . No aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_divrem(fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that B is non-zero and that the leading coefficient of B is invertible modulo p .

```
void fmpz_mod_poly_divrem_f(fmpz_t f, fmpz_mod_poly_t Q, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Either finds a non-trivial factor f of the modulus p , or computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

If the leading coefficient of B is invertible in $\mathbf{Z}/(p)$, the division with remainder operation is carried out, Q and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of p and Q and R are not touched.

Assumes that B is non-zero.

```
void _fmpz_mod_poly_rem(fmpz_t *R, const fmpz_t *A, slong lenA, const fmpz_t *B, slong lenB, const fmpz_t invB, const fmpz_mod_ctx_t ctx)
```

Notationally, computes $(Q, \text{lenA} - \text{lenB} + 1), (R, \text{lenB} - 1)$ such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$, returning only the remainder part.

Assumes that B is non-zero, that the leading coefficient of B is invertible modulo p and that invB is the inverse.

Assumes $\text{len}(A) \geq \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . No aliasing of input and output operands is allowed.

```
void fmpz_mod_poly_rem_f(fmpz_t f, fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

If f returns with the value 1 then the function operates as `_fmpz_mod_poly_rem`, otherwise f will be set to a nontrivial factor of p .

```
void fmpz_mod_poly_rem(fmpz_mod_poly_t R, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$, returning only the remainder part.

Assumes that B is non-zero and that the leading coefficient of B is invertible modulo p .

6.13.16 Divisibility testing

`int _fmpz_mod_poly_divides_classical(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)`

Returns 1 if $(B, \text{len}B)$ divides $(A, \text{len}A)$ and sets $(Q, \text{len}A - \text{len}B + 1)$ to the quotient. Otherwise, returns 0 and sets $(Q, \text{len}A - \text{len}B + 1)$ to zero. We require that $\text{len}A \geq \text{len}B > 0$.

`int fmpz_mod_poly_divides_classical(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)`

Returns 1 if B divides A and sets Q to the quotient. Otherwise returns 0 and sets Q to zero.

`int _fmpz_mod_poly_divides(fmpz *Q, const fmpz *A, slong lenA, const fmpz *B, slong lenB, const fmpz_mod_ctx_t ctx)`

Returns 1 if $(B, \text{len}B)$ divides $(A, \text{len}A)$ and sets $(Q, \text{len}A - \text{len}B + 1)$ to the quotient. Otherwise, returns 0 and sets $(Q, \text{len}A - \text{len}B + 1)$ to zero. We require that $\text{len}A \geq \text{len}B > 0$.

`int fmpz_mod_poly_divides(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)`

Returns 1 if B divides A and sets Q to the quotient. Otherwise returns 0 and sets Q to zero.

6.13.17 Power series inversion

`void _fmpz_mod_poly_inv_series(fmpz *Qinv, const fmpz *Q, slong Qlen, slong n, const fmpz_mod_ctx_t ctx)`

Sets $(Qinv, n)$ to the inverse of (Q, n) modulo x^n , where $n \geq 1$, assuming that the bottom coefficient of Q is invertible modulo p and that its inverse is `cinv`.

`void fmpz_mod_poly_inv_series(fmpz_mod_poly_t Qinv, const fmpz_mod_poly_t Q, slong n, const fmpz_mod_ctx_t ctx)`

Sets $Qinv$ to the inverse of Q modulo x^n , where $n \geq 1$, assuming that the bottom coefficient of Q is a unit.

`void fmpz_mod_poly_inv_series_f(fmpz_t f, fmpz_mod_poly_t Qinv, const fmpz_mod_poly_t Q, slong n, const fmpz_mod_ctx_t ctx)`

Either sets f to a nontrivial factor of p with the value of $Qinv$ undefined, or sets $Qinv$ to the inverse of Q modulo x^n , where $n \geq 1$.

6.13.18 Power series division

`void _fmpz_mod_poly_div_series(fmpz *Q, const fmpz *A, slong Alen, const fmpz *B, slong Blen, slong n, const fmpz_mod_ctx_t ctx)`

Set (Q, n) to the quotient of the series $(A, A\text{len})$ and $(B, B\text{len})$ assuming $A\text{len}, B\text{len} \leq n$. We assume the bottom coefficient of B is invertible modulo p .

`void fmpz_mod_poly_div_series(fmpz_mod_poly_t Q, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, slong n, const fmpz_mod_ctx_t ctx)`

Set Q to the quotient of the series A by B , thinking of the series as though they were of length n . We assume that the bottom coefficient of B is a unit.

6.13.19 Greatest common divisor

```
void fmpz_mod_poly_make_monic(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const  
                             fmpz_mod_ctx_t ctx)
```

If `poly` is non-zero, sets `res` to `poly` divided by its leading coefficient. This assumes that the leading coefficient of `poly` is invertible modulo p .

Otherwise, if `poly` is zero, sets `res` to zero.

```
void fmpz_mod_poly_make_monic_f(fmpz_t f, fmpz_mod_poly_t res, const fmpz_mod_poly_t poly,  
                               const fmpz_mod_ctx_t ctx)
```

Either set f to 1 and `res` to `poly` divided by its leading coefficient or set f to a nontrivial factor of p and leave `res` undefined.

```
slong _fmpz_mod_poly_gcd(fmpz_t G, const fmpz_t A, slong lenA, const fmpz_t B, slong lenB, const  
                        fmpz_mod_ctx_t ctx)
```

Sets G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Assumes that `invB` is the inverse of the leading coefficients of B modulo the prime number p .

```
void fmpz_mod_poly_gcd(fmpz_mod_poly_t G, const fmpz_mod_poly_t A, const fmpz_mod_poly_t  
                      B, const fmpz_mod_ctx_t ctx)
```

Sets G to the greatest common divisor of A and B .

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number. Thus, this function assumes that p is prime.

```
slong _fmpz_mod_poly_gcd_euclidean_f(fmpz_t f, fmpz_t G, const fmpz_t A, slong lenA, const fmpz_t  
                                     B, slong lenB, const fmpz_mod_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length, or sets $f \in (1, p)$ to a non-trivial factor of p and leaves the contents of the vector $(G, \text{len}B)$ undefined.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Does not support aliasing of any of the input arguments with any of the output argument.

```
void fmpz_mod_poly_gcd_euclidean_f(fmpz_t f, fmpz_mod_poly_t G, const fmpz_mod_poly_t A,  
                                   const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of A and B , or $f \in (1, p)$ to a non-trivial factor of p .

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number.

```
slong _fmpz_mod_poly_gcd_f(fmpz_t f, fmpz_t G, const fmpz_t A, slong lenA, const fmpz_t B, slong  
                          lenB, const fmpz_mod_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length, or sets $f \in (1, p)$ to a non-trivial factor of p and leaves the contents of the vector $(G, \text{len}B)$ undefined.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

Does not support aliasing of any of the input arguments with any of the output arguments.

```
void fmpz_mod_poly_gcd_f(fmpz_t f, fmpz_mod_poly_t G, const fmpz_mod_poly_t A, const  
                        fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of A and B , or $f \in (1, p)$ to a non-trivial factor of p .

In general, the greatest common divisor is defined in the polynomial ring $(\mathbf{Z}/(p\mathbf{Z}))[X]$ if and only if p is a prime number.

```
long _fmpz_mod_poly_hgcd(fmpz **M, long *lenM, fmpz *A, long *lenA, fmpz *B, long *lenB,
                        const fmpz *a, long lena, const fmpz *b, long lenb, const
                        fmpz_mod_ctx_t ctx)
```

Computes the HGCD of a and b , that is, a matrix M , a sign σ and two polynomials A and B such that

$$(A, B)^t = \sigma M^{-1}(a, b)^t.$$

Assumes that $\text{len}(a) > \text{len}(b) > 0$.

Assumes that A and B have space of size at least $\text{len}(a)$ and $\text{len}(b)$, respectively. On exit, $*lenA$ and $*lenB$ will contain the correct lengths of A and B .

Assumes that $M[0]$, $M[1]$, $M[2]$, and $M[3]$ each point to a vector of size at least $\text{len}(a)$.

```
long _fmpz_mod_poly_xgcd_euclidean_f(fmpz_t f, fmpz *G, fmpz *S, fmpz *T, const fmpz *A,
                                     long lenA, const fmpz *B, long lenB, const fmpz_t invB,
                                     const fmpz_mod_ctx_t ctx)
```

If f returns with the value 1 then the function operates as per `_fmpz_mod_poly_xgcd_euclidean`, otherwise f is set to a nontrivial factor of p .

```
void fmpz_mod_poly_xgcd_euclidean_f(fmpz_t f, fmpz_mod_poly_t G, fmpz_mod_poly_t S,
                                    fmpz_mod_poly_t T, const fmpz_mod_poly_t A, const
                                    fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

If f returns with the value 1 then the function operates as per `fmpz_mod_poly_xgcd_euclidean`, otherwise f is set to a nontrivial factor of p .

```
long _fmpz_mod_poly_xgcd(fmpz *G, fmpz *S, fmpz *T, const fmpz *A, long lenA, const fmpz *B,
                        long lenB, const fmpz_t invB, const fmpz_mod_ctx_t ctx)
```

Computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fmpz_mod_poly_xgcd(fmpz_mod_poly_t G, fmpz_mod_poly_t S, fmpz_mod_poly_t T, const
                      fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const fmpz_mod_ctx_t
                      ctx)
```

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S*A + T*B = G$. The length of S will be at most $\text{len}B$ and the length of T will be at most $\text{len}A$.

```
void fmpz_mod_poly_xgcd_f(fmpz_t f, fmpz_mod_poly_t G, fmpz_mod_poly_t S, fmpz_mod_poly_t
                        T, const fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
                        fmpz_mod_ctx_t ctx)
```

If f returns with the value 1 then the function operates as per `fmpz_mod_poly_xgcd`, otherwise f is set to a nontrivial factor of p .

```
long _fmpz_mod_poly_gcdinv_euclidean(fmpz *G, fmpz *S, const fmpz *A, long lenA, const fmpz
                                     *B, long lenB, const fmpz_t invA, const
                                     fmpz_mod_ctx_t ctx)
```

Computes $(G, \text{len}A)$, $(S, \text{len}B-1)$ such that $G \cong SA \pmod{B}$, returning the actual length of G .

Assumes that $0 < \text{len}(A) < \text{len}(B)$.

```
void fmpz_mod_poly_gcdinv_euclidean(fmpz_mod_poly_t G, fmpz_mod_poly_t S, const
                                   fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
                                   fmpz_mod_ctx_t ctx)
```

Computes polynomials G and S , both reduced modulo B , such that $G \cong SA \pmod{B}$, where B is assumed to have $\text{len}(B) \geq 2$.

In the case that $A \equiv 0 \pmod{B}$, returns $G = S = 0$.

```
slong _fmpz_mod_poly_gcdinv_euclidean_f(fmpz_t f, fmpz *G, fmpz *S, const fmpz *A, slong lenA,
                                         const fmpz *B, slong lenB, const fmpz_t invA, const
                                         fmpz_mod_ctx_t ctx)
```

If f returns with value 1 then the function operates as per `_fmpz_mod_poly_gcdinv_euclidean()`, otherwise f is set to a nontrivial factor of p .

```
void fmpz_mod_poly_gcdinv_euclidean_f(fmpz_t f, fmpz_mod_poly_t G, fmpz_mod_poly_t S,
                                       const fmpz_mod_poly_t A, const fmpz_mod_poly_t B,
                                       const fmpz_mod_ctx_t ctx)
```

If f returns with value 1 then the function operates as per `fmpz_mod_poly_gcdinv_euclidean()`, otherwise f is set to a nontrivial factor of the modulus of A .

```
slong _fmpz_mod_poly_gcdinv(fmpz *G, fmpz *S, const fmpz *A, slong lenA, const fmpz *B, slong
                           lenB, const fmpz_mod_ctx_t ctx)
```

Computes (G, lenA) , $(S, \text{lenB}-1)$ such that $G \cong SA \pmod{B}$, returning the actual length of G .

Assumes that $0 < \text{len}(A) < \text{len}(B)$.

```
slong _fmpz_mod_poly_gcdinv_f(fmpz_t f, fmpz *G, fmpz *S, const fmpz *A, slong lenA, const fmpz
                             *B, slong lenB, const fmpz_mod_ctx_t ctx)
```

If f returns with value 1 then the function operates as per `_fmpz_mod_poly_gcdinv()`, otherwise f will be set to a nontrivial factor of p .

```
void fmpz_mod_poly_gcdinv(fmpz_mod_poly_t G, fmpz_mod_poly_t S, const fmpz_mod_poly_t A,
                          const fmpz_mod_poly_t B, const fmpz_mod_ctx_t ctx)
```

Computes polynomials G and S , both reduced modulo B , such that $G \cong SA \pmod{B}$, where B is assumed to have $\text{len}(B) \geq 2$.

In the case that $A \equiv 0 \pmod{B}$, returns $G = S = 0$.

```
void fmpz_mod_poly_gcdinv_f(fmpz_t f, fmpz_mod_poly_t G, fmpz_mod_poly_t S, const
                            fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
                            fmpz_mod_ctx_t ctx)
```

If f returns with value 1 then the function operates as per `fmpz_mod_poly_gcdinv()`, otherwise f will be set to a nontrivial factor of p .

```
int _fmpz_mod_poly_invmod(fmpz *A, const fmpz *B, slong lenB, const fmpz *P, slong lenP, const
                          fmpz_mod_ctx_t ctx)
```

Attempts to set $(A, \text{lenP}-1)$ to the inverse of (B, lenB) modulo the polynomial (P, lenP) . Returns 1 if (B, lenB) is invertible and 0 otherwise.

Assumes that $0 < \text{len}(B) < \text{len}(P)$, and hence also $\text{len}(P) \geq 2$, but supports zero-padding in (B, lenB) .

Does not support aliasing.

Assumes that p is a prime number.

```
int _fmpz_mod_poly_invmod_f(fmpz_t f, fmpz *A, const fmpz *B, slong lenB, const fmpz *P, slong
                           lenP, const fmpz_mod_ctx_t ctx)
```

If f returns with the value 1, then the function operates as per `_fmpz_mod_poly_invmod()`. Otherwise f is set to a nontrivial factor of p .


```
int fmpz_mod_poly_invmod(fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
                        fmpz_mod_poly_t P, const fmpz_mod_ctx_t ctx)
```

Attempts to set A to the inverse of B modulo P in the polynomial ring $(\mathbf{Z}/p\mathbf{Z})[X]$, where we assume that p is a prime number.

If $\deg(P) < 2$, raises an exception.

If the greatest common divisor of B and P is ~ 1 , returns ~ 1 and sets A to the inverse of B . Otherwise, returns ~ 0 and the value of A on exit is undefined.

```
int fmpz_mod_poly_invmod_f(fmpz_t f, fmpz_mod_poly_t A, const fmpz_mod_poly_t B, const
                        fmpz_mod_poly_t P, const fmpz_mod_ctx_t ctx)
```

If f returns with the value 1, then the function operates as per `fmpz_mod_poly_invmod()`. Otherwise f is set to a nontrivial factor of p .

6.13.20 Minpoly

```
slong _fmpz_mod_poly_minpoly_bm(fmpz *poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t
                                ctx)
```

Sets `poly` to the coefficients of a minimal generating polynomial for sequence `(seq, len)` modulo p .

The return value equals the length of `poly`.

It is assumed that p is prime and `poly` has space for at least $len + 1$ coefficients. No aliasing between inputs and outputs is allowed.

```
void fmpz_mod_poly_minpoly_bm(fmpz_mod_poly_t poly, const fmpz *seq, slong len, const
                              fmpz_mod_ctx_t ctx)
```

Sets `poly` to a minimal generating polynomial for sequence `seq` of length `len`.

Assumes that the modulus is prime.

This version uses the Berlekamp-Massey algorithm, whose running time is proportional to `len` times the size of the minimal generator.

```
slong _fmpz_mod_poly_minpoly_hgcd(fmpz *poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t
                                  ctx)
```

Sets `poly` to the coefficients of a minimal generating polynomial for sequence `(seq, len)` modulo p .

The return value equals the length of `poly`.

It is assumed that p is prime and `poly` has space for at least $len + 1$ coefficients. No aliasing between inputs and outputs is allowed.

```
void fmpz_mod_poly_minpoly_hgcd(fmpz_mod_poly_t poly, const fmpz *seq, slong len, const
                                fmpz_mod_ctx_t ctx)
```

Sets `poly` to a minimal generating polynomial for sequence `seq` of length `len`.

Assumes that the modulus is prime.

This version uses the HGCD algorithm, whose running time is $O(n \log^2 n)$ field operations, regardless of the actual size of the minimal generator.

```
slong _fmpz_mod_poly_minpoly(fmpz *poly, const fmpz *seq, slong len, const fmpz_mod_ctx_t ctx)
```

Sets `poly` to the coefficients of a minimal generating polynomial for sequence `(seq, len)` modulo p .

The return value equals the length of `poly`.

It is assumed that p is prime and `poly` has space for at least $len + 1$ coefficients. No aliasing between inputs and outputs is allowed.


```
void fmpz_mod_poly_minpoly(fmpz_mod_poly_t poly, const fmpz *seq, slong len, const
                           fmpz_mod_ctx_t ctx)
```

Sets `poly` to a minimal generating polynomial for sequence `seq` of length `len`.

A minimal generating polynomial is a monic polynomial $f = x^d + c_{d-1}x^{d-1} + \dots + c_1x + c_0$, of minimal degree d , that annihilates any consecutive $d + 1$ terms in `seq`. That is, for any $i < \text{len} - d$,

$$\text{seq}_i = -\sum_{j=0}^{d-1} \text{seq}_{i+j} * f_j.$$

Assumes that the modulus is prime.

This version automatically chooses the fastest underlying implementation based on `len` and the size of the modulus.

6.13.21 Resultant

```
void _fmpz_mod_poly_resultant(fmpz_t res, const fmpz *poly1, slong len1, const fmpz *poly2, slong
                              len2, const fmpz_mod_ctx_t ctx)
```

Returns the resultant of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `len1 >= len2 > 0`.

The complexity is only guaranteed to be quasilinear if the modulus is prime.

```
void fmpz_mod_poly_resultant(fmpz_t res, const fmpz_mod_poly_t f, const fmpz_mod_poly_t g,
                             const fmpz_mod_ctx_t ctx)
```

Computes the resultant of `f` and `g`.

6.13.22 Discriminant

```
void _fmpz_mod_poly_discriminant(fmpz_t d, const fmpz *poly, slong len, const fmpz_mod_ctx_t
                                 ctx)
```

Set `d` to the discriminant of `(poly, len)`. Assumes `len > 1`.

```
void fmpz_mod_poly_discriminant(fmpz_t d, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t
                                ctx)
```

Set `d` to the discriminant of `f`. We normalise the discriminant so that $\text{disc}(f) = (-1)^{n(n-1)/2} \text{res}(f, f') / \text{lc}(f)^{(n-m-2)}$, where $n = \text{len}(f)$ and $m = \text{len}(f')$. Thus $\text{disc}(f) = \text{lc}(f)^{(2n-2)} \prod_{i < j} (r_i - r_j)^2$, where $\text{lc}(f)$ is the leading coefficient of `f` and r_i are the roots of `f`.

6.13.23 Derivative

```
void _fmpz_mod_poly_derivative(fmpz *res, const fmpz *poly, slong len, const fmpz_mod_ctx_t
                               ctx)
```

Sets `(res, len - 1)` to the derivative of `(poly, len)`. Also handles the cases where `len` is 0 or 1 correctly. Supports aliasing of `res` and `poly`.

```
void fmpz_mod_poly_derivative(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly, const
                              fmpz_mod_ctx_t ctx)
```

Sets `res` to the derivative of `poly`.

6.13.24 Evaluation

```
void _fmpz_mod_poly_evaluate_fmpz(fmpz_t res, const fmpz_t *poly, slong len, const fmpz_t a, const
fmpz_mod_ctx_t ctx)
```

Evaluates the polynomial (poly, len) at the integer *a* and sets *res* to the result. Aliasing between *res* and *a* or any of the coefficients of *poly* is not supported.

```
void fmpz_mod_poly_evaluate_fmpz(fmpz_t res, const fmpz_mod_poly_t poly, const fmpz_t a, const
fmpz_mod_ctx_t ctx)
```

Evaluates the polynomial *poly* at the integer *a* and sets *res* to the result.

As expected, aliasing between *res* and *a* is supported. However, *res* may not be aliased with a coefficient of *poly*.

6.13.25 Multipoint evaluation

```
void _fmpz_mod_poly_evaluate_fmpz_vec_iter(fmpz_t *ys, const fmpz_t *coeffs, slong len, const fmpz_t
*xs, slong n, const fmpz_mod_ctx_t ctx)
```

Evaluates (coeffs, len) at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void fmpz_mod_poly_evaluate_fmpz_vec_iter(fmpz_t *ys, const fmpz_mod_poly_t poly, const fmpz_t
*xs, slong n, const fmpz_mod_ctx_t ctx)
```

Evaluates *poly* at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

Uses Horner's method iteratively.

```
void _fmpz_mod_poly_evaluate_fmpz_vec_fast_precomp(fmpz_t *vs, const fmpz_t *poly, slong plen,
fmpz_poly_struct *const *tree, slong len,
const fmpz_mod_ctx_t ctx)
```

Evaluates (poly, plen) at the *len* values given by the precomputed subproduct tree *tree*.

```
void _fmpz_mod_poly_evaluate_fmpz_vec_fast(fmpz_t *ys, const fmpz_t *poly, slong plen, const fmpz_t
*xs, slong n, const fmpz_mod_ctx_t ctx)
```

Evaluates (coeffs, len) at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void fmpz_mod_poly_evaluate_fmpz_vec_fast(fmpz_t *ys, const fmpz_mod_poly_t poly, const fmpz_t
*xs, slong n, const fmpz_mod_ctx_t ctx)
```

Evaluates *poly* at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

Uses fast multipoint evaluation, building a temporary subproduct tree.

```
void _fmpz_mod_poly_evaluate_fmpz_vec(fmpz_t *ys, const fmpz_t *coeffs, slong len, const fmpz_t *xs,
slong n, const fmpz_mod_ctx_t ctx)
```

Evaluates (coeffs, len) at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

```
void fmpz_mod_poly_evaluate_fmpz_vec(fmpz_t *ys, const fmpz_mod_poly_t poly, const fmpz_t *xs,
slong n, const fmpz_mod_ctx_t ctx)
```

Evaluates *poly* at the *n* values given in the vector *xs*, writing the output values to *ys*. The values in *xs* should be reduced modulo the modulus.

6.13.26 Composition

```
void _fmpz_mod_poly_compose(fmpz *res, const fmpz *poly1, slong len1, const fmpz *poly2, slong
                           len2, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition of `(poly1, len1)` and `(poly2, len2)`.

Assumes that `res` has space for $(len1-1)*(len2-1) + 1$ coefficients, although in $\mathbf{Z}_p[X]$ this might not actually be the length of the resulting polynomial when p is not a prime.

Assumes that `poly1` and `poly2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fmpz_mod_poly_compose(fmpz_mod_poly_t res, const fmpz_mod_poly_t poly1, const
                           fmpz_mod_poly_t poly2, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition of `poly1` and `poly2`.

To be precise about the order of composition, denoting `res`, `poly1`, and `poly2` by f , g , and h , respectively, sets $f(t) = g(h(t))$.

6.13.27 Square roots

The series expansions for $\sqrt{h}$ and $1/\sqrt{h}$ are defined by means of the generalised binomial theorem $(1+y)^r = \sum_{k=0}^{\infty} \binom{r}{k} y^k$. It is assumed that h has constant term 1 and that the coefficients 2^{-k} exist in the coefficient ring (i.e. 2 must be invertible).

```
void _fmpz_mod_poly_invsqrt_series(fmpz *g, const fmpz *h, slong hlen, slong n, const
                                   fmpz_mod_ctx_t ctx)
```

Set the first n terms of g to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$ and $h > 0$. Aliasing is not permitted.

```
void fmpz_mod_poly_invsqrt_series(fmpz_mod_poly_t g, const fmpz_mod_poly_t h, slong n, const
                                   fmpz_mod_ctx_t ctx)
```

Set g to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
void _fmpz_mod_poly_sqrt_series(fmpz *g, const fmpz *h, slong hlen, slong n, const
                                fmpz_mod_ctx_t ctx)
```

Set the first n terms of g to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$ and $h > 0$. Aliasing is not permitted.

```
void fmpz_mod_poly_sqrt_series(fmpz_mod_poly_t g, const fmpz_mod_poly_t h, slong n, const
                                fmpz_mod_ctx_t ctx)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
int _fmpz_mod_poly_sqrt(fmpz *s, const fmpz *p, slong n, const fmpz_mod_ctx_t ctx)
```

If (p, n) is a perfect square, sets $(s, n / 2 + 1)$ to a square root of p and returns 1. Otherwise returns 0.

```
int fmpz_mod_poly_sqrt(fmpz_mod_poly_t s, const fmpz_mod_poly_t p, const fmpz_mod_ctx_t
                       ctx)
```

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

6.13.28 Modular composition

```
void _fmpz_mod_poly_compose_mod(fmpz *res, const fmpz *f, slong lenf, const fmpz *g, const fmpz
                                *h, slong lenh, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fmpz_mod_poly_compose_mod(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const
                                fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const
                                fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero.

```
void _fmpz_mod_poly_compose_mod_horner(fmpz *res, const fmpz *f, slong lenf, const fmpz *g, const
                                        fmpz *h, slong lenh, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fmpz_mod_poly_compose_mod_horner(fmpz_mod_poly_t res, const fmpz_mod_poly_t f, const
                                        fmpz_mod_poly_t g, const fmpz_mod_poly_t h, const
                                        fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fmpz_mod_poly_compose_mod_brent_kung(fmpz *res, const fmpz *f, slong len1, const fmpz *g,
                                            const fmpz *h, slong len3, const fmpz_mod_ctx_t
                                            ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fmpz_mod_poly_compose_mod_brent_kung(fmpz_mod_poly_t res, const fmpz_mod_poly_t f,
                                            const fmpz_mod_poly_t g, const fmpz_mod_poly_t
                                            h, const fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fmpz_mod_poly_reduce_matrix_mod_poly(fmpz_mat_t A, const fmpz_mat_t B, const
                                            fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Sets the i th row of A to the reduction of the i th row of B modulo f for $i = 1, \dots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fmpz_mod_poly_precompute_matrix_worker(void *arg_ptr)
```

Worker function version of `_fmpz_mod_poly_precompute_matrix`. Input/output is stored in `fmpz_mod_poly_matrix_precompute_arg_t`.

```
void _fmpz_mod_poly_precompute_matrix(fmpz_mat_t A, const fmpz *f, const fmpz *g, slong leng,
                                      const fmpz *ginv, slong lenginv, const fmpz_mod_ctx_t
                                      ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g and g to be nonzero. f has to be reduced modulo g and of length one less than `leng` (possibly with zero padding).

```
void fmpz_mod_poly_precompute_matrix(fmpz_mat_t A, const fmpz_mod_poly_t f, const
                                      fmpz_mod_poly_t g, const fmpz_mod_poly_t ginv, const
                                      fmpz_mod_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require ginv to be the inverse of the reverse of g .

```
void _fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv_worker(void *arg_ptr)
```

Worker function version of `_fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv()`. Input/output is stored in `fmpz_mod_poly_compose_mod_precomp_preinv_arg_t`.

```
void _fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(fmpz *res, const fmpz *f, slong
lenf, const fmpz_mat_t A, const
fmpz *h, slong lenh, const fmpz
*hinv, slong lenhin, const
fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv(fmpz_mod_poly_t res, const
fmpz_mod_poly_t f, const
fmpz_mat_t A, const
fmpz_mod_poly_t h, const
fmpz_mod_poly_t hinv, const
fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h .

```
void _fmpz_mod_poly_compose_mod_brent_kung_preinv(fmpz *res, const fmpz *f, slong lenf, const
fmpz *g, const fmpz *h, slong lenh, const
fmpz *hinv, slong lenhin, const
fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fmpz_mod_poly_compose_mod_brent_kung_preinv(fmpz_mod_poly_t res, const
fmpz_mod_poly_t f, const
fmpz_mod_poly_t g, const
fmpz_mod_poly_t h, const
fmpz_mod_poly_t hinv, const
fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fmpz_mod_poly_compose_mod_brent_kung_vec_preinv(fmpz_mod_poly_struct *res, const
fmpz_mod_poly_struct *polys, slong
len1, slong l, const fmpz *g, slong glen,
const fmpz *h, slong lenh, const fmpz
*hinv, slong lenhin, const
fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f_i(g)$ modulo h for $1 \leq i \leq l$, where f_i are the l elements of `polys`.

We require that h is nonzero and that the length of g is less than the length of h . We also require that the length of f_i is less than the length of h . We require `res` to have enough memory allocated to hold `len1` `fmpz_mod_poly_struct`'s. The entries of `res` need to be initialised and `len1` needs to be less than `len1`. Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fmpz_mod_poly_compose_mod_brent_kung_vec_preinv(fmpz_mod_poly_struct *res, const
                                                    fmpz_mod_poly_struct *polys, slong
                                                    len1, slong n, const fmpz_mod_poly_t
                                                    g, const fmpz_mod_poly_t h, const
                                                    fmpz_mod_poly_t hinv, const
                                                    fmpz_mod_ctx_t ctx)
```

Sets `res` to the composition $f_i(g)$ modulo h for $1 \leq i \leq n$ where f_i are the `n` elements of `polys`. We require `res` to have enough memory allocated to hold `n` `fmpz_mod_poly_struct`'s. The entries of `res` need to be initialised and `n` needs to be less than `len1`. We require that h is nonzero and that f_i and g have smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . No aliasing of `res` and `polys` is allowed. The algorithm used is the Brent-Kung matrix algorithm.

```
void _fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(fmpz_mod_poly_struct
                                                                    *res, const
                                                                    fmpz_mod_poly_struct
                                                                    *polys, slong lenpolys,
                                                                    slong l, const fmpz *g,
                                                                    slong glen, const fmpz
                                                                    *poly, slong len, const
                                                                    fmpz *polyinv, slong
                                                                    leninv, const
                                                                    fmpz_mod_ctx_t ctx,
                                                                    thread_pool_handle
                                                                    *threads, slong
                                                                    num_threads)
```

Multithreaded version of `_fmpz_mod_poly_compose_mod_brent_kung_vec_preinv()`. Distributing the Horner evaluations across `flint_get_num_threads()` threads.

```
void fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool(fmpz_mod_poly_struct
                                                                    *res, const
                                                                    fmpz_mod_poly_struct
                                                                    *polys, slong len1, slong
                                                                    n, const
                                                                    fmpz_mod_poly_t g,
                                                                    const
                                                                    fmpz_mod_poly_t poly,
                                                                    const
                                                                    fmpz_mod_poly_t
                                                                    polyinv, const
                                                                    fmpz_mod_ctx_t ctx,
                                                                    thread_pool_handle
                                                                    *threads, slong
                                                                    num_threads)
```

Multithreaded version of `fmpz_mod_poly_compose_mod_brent_kung_vec_preinv()`. Distributing the Horner evaluations across `flint_get_num_threads()` threads.

```
void fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded(fmpz_mod_poly_struct *res,
                                                             const fmpz_mod_poly_struct
                                                             *polys, slong len1, slong n,
                                                             const fmpz_mod_poly_t g,
                                                             const fmpz_mod_poly_t poly,
                                                             const fmpz_mod_poly_t
                                                             polyinv, const
                                                             fmpz_mod_ctx_t ctx)
```

Multithreaded version of `fmpz_mod_poly_compose_mod_brent_kung_vec_preinv()`. Distributing the Horner evaluations across `flint_get_num_threads()` threads.

6.13.29 Subproduct trees

```
fmpz_poly_struct **_fmpz_mod_poly_tree_alloc(slong len)
```

Allocates space for a subproduct tree of the given length, having linear factors at the lowest level.

```
void _fmpz_mod_poly_tree_free(fmpz_poly_struct **tree, slong len)
```

Free the allocated space for the subproduct.

```
void _fmpz_mod_poly_tree_build(fmpz_poly_struct **tree, const fmpz *roots, slong len, const
                              fmpz_mod_ctx_t ctx)
```

Builds a subproduct tree in the preallocated space from the `len` monic linear factors $(x - r_i)$ where r_i are given by `roots`. The top level product is not computed.

6.13.30 Radix conversion

The following functions provide the functionality to solve the radix conversion problems for polynomials, which is to express a polynomial $f(X)$ with respect to a given radix $r(X)$ as

$$f(X) = \sum_{i=0}^N b_i(X) r(X)^i$$

where $N = \lfloor \deg(f)/\deg(r) \rfloor$. The algorithm implemented here is a recursive one, which performs Euclidean divisions by powers of r of the form r^{2^i} , and it has time complexity $\Theta(\deg(f) \log \deg(f))$. It facilitates the repeated use of precomputed data, namely the powers of r and their power series inverses. This data is stored in objects of type `fmpz_mod_poly_radix_t` and it is computed using the function `fmpz_mod_poly_radix_init()`, which only depends on r and an upper bound on the degree of f .

```
void _fmpz_mod_poly_radix_init(fmpz **Rpow, fmpz **Rinv, const fmpz *R, slong lenR, slong k,
                              const fmpz_t invL, const fmpz_mod_ctx_t ctx)
```

Computes powers of R of the form R^{2^i} and their Newton inverses modulo $x^{2^i \deg(R)}$ for $i = 0, \dots, k-1$.

Assumes that the vectors `Rpow[i]` and `Rinv[i]` have space for $2^i \deg(R) + 1$ and $2^i \deg(R)$ coefficients, respectively.

Assumes that the polynomial R is non-constant, i.e. $\deg(R) \geq 1$.

Assumes that the leading coefficient of R is a unit and that the argument `invL` is the inverse of the coefficient modulo p .

The argument p is the modulus, which in p -adic applications is typically a prime power, although this is not necessary. Here, we only assume that $p \geq 2$.

Note that this precomputed data can be used for any F such that $\deg(F) \leq 2^k \deg(R)$.


```
void fmpz_mod_poly_radix_init(fmpz_mod_poly_radix_t D, const fmpz_mod_poly_t R, slong
                             degF, const fmpz_mod_ctx_t ctx)
```

Carries out the precomputation necessary to perform radix conversion to radix- R for polynomials of degree at most $\deg F$.

Assumes that R is non-constant, i.e. $\deg(R) \geq 1$, and that the leading coefficient is a unit.

```
void _fmpz_mod_poly_radix(fmpz **B, const fmpz *F, fmpz **Rpow, fmpz **Rinv, slong degR, slong
                          k, slong i, fmpz *W, const fmpz_mod_ctx_t ctx)
```

This is the main recursive function used by the function `fmpz_mod_poly_radix()`.

Assumes that, for all $i = 0, \dots, N$, the vector `B[i]` has space for $\deg(R)$ coefficients.

The variable k denotes the factors of r that have previously been counted for the polynomial F , which is assumed to have length $2^{i+1} \deg(R)$, possibly including zero-padding.

Assumes that W is a vector providing temporary space of length $\text{len}(F) = 2^{i+1} \deg(R)$.

The entire computation takes place over $\mathbf{Z}/p\mathbf{Z}$, where $p \geq 2$ is a natural number.

Thus, the top level call will have F as in the original problem, and $k = 0$.

```
void fmpz_mod_poly_radix(fmpz_mod_poly_struct **B, const fmpz_mod_poly_t F, const
                        fmpz_mod_poly_radix_t D, const fmpz_mod_ctx_t ctx)
```

Given a polynomial F and the precomputed data D for the radix R , computes polynomials $B_0, \dots, B_N$ of degree less than $\deg(R)$ such that

$$F = B_0 + B_1 R + \dots + B_N R^N,$$

where necessarily $N = \lfloor \deg(F) / \deg(R) \rfloor$.

Assumes that R is non-constant, i.e., $\deg(R) \geq 1$, and that the leading coefficient is a unit.

6.13.31 Input and output

The printing options supported by this module are very similar to what can be found in the two related modules `fmpz_poly` and `nmod_poly`. Consider, for example, the polynomial $f(x) = 5x^3 + 2x + 1$ in $(\mathbf{Z}/6\mathbf{Z})[x]$. Its simple string representation is "4 6 1 2 0 5", where the first two numbers denote the length of the polynomial and the modulus. The pretty string representation is "5*x^3+2*x+1".

```
int _fmpz_mod_poly_fprint(FILE *file, const fmpz *poly, slong len, const fmpz_t p)
```

Prints the polynomial `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_fprint(FILE *file, const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Prints the polynomial to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_fprint_pretty(FILE *file, const fmpz_mod_poly_t poly, const char *x, const
                                fmpz_mod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_print(const fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Prints the polynomial to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fmpz_mod_poly_print_pretty(const fmpz_mod_poly_t poly, const char *x, const
                               fmpz_mod_ctx_t ctx)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

6.13.32 Inflation and deflation

```
void fmpz_mod_poly_inflate(fmpz_mod_poly_t result, const fmpz_mod_poly_t input, ulong
                           inflation, const fmpz_mod_ctx_t ctx)
```

Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `inflation`.

```
void fmpz_mod_poly_deflate(fmpz_mod_poly_t result, const fmpz_mod_poly_t input, ulong
                           deflation, const fmpz_mod_ctx_t ctx)
```

Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

```
ulong fmpz_mod_poly_deflation(const fmpz_mod_poly_t input, const fmpz_mod_ctx_t ctx)
```

Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 if `input` is a constant polynomial.

6.13.33 Berlekamp-Massey Algorithm

The `fmpz_mod_berlekamp_massey_t` manages an unlimited stream of points $a_1, a_2, \dots$. At any point in time, after, say, n points have been added, a call to `fmpz_mod_berlekamp_massey_reduce()` will calculate the polynomials U , V and R in the extended euclidean remainder sequence with

$$U * x^n + V * (a_1 * x^{n-1} + \dots + a_{n-1} * x + a_n) = R, \quad \deg(U) < \deg(V) \leq n/2, \quad \deg(R) < n/2.$$

The polynomials V and R may be obtained with `fmpz_mod_berlekamp_massey_V_poly()` and `fmpz_mod_berlekamp_massey_R_poly()`. This class differs from `fmpz_mod_poly_minpoly()` in the following respect. Let v_i denote the coefficient of x^i in V . `fmpz_mod_poly_minpoly()` will return a polynomial V of lowest degree that annihilates the whole sequence $a_1, \dots, a_n$ as

$$\sum_i v_i a_{j+i} = 0, \quad 1 \leq j \leq n - \deg(V).$$

The cost is that a polynomial of degree $n-1$ might be returned and the return is not generally uniquely determined by the input sequence. For the `fmpz_mod_berlekamp_massey_t` we have

$$\sum_{i,j} v_i a_{j+i} x^{-j} = -U + \frac{R}{x^n},$$

and it can be seen that $\sum_i v_i a_{j+i}$ is zero for $1 \leq j < n - \deg(R)$. Thus whether or not V has annihilated the whole sequence may be checked by comparing the degrees of V and R .

```
void fmpz_mod_berlekamp_massey_init(fmpz_mod_berlekamp_massey_t B, const
                                    fmpz_mod_ctx_t ctx)
```

Initialize `B` with an empty stream.

```
void fmpz_mod_berlekamp_massey_clear(fmpz_mod_berlekamp_massey_t B, const
                                     fmpz_mod_ctx_t ctx)
```

Free any space used by `B`.

```
void fmpz_mod_berlekamp_massey_start_over(fmpz_mod_berlekamp_massey_t B, const
                                         fmpz_mod_ctx_t ctx)
```

Empty the stream of points in B.

```
void fmpz_mod_berlekamp_massey_add_points(fmpz_mod_berlekamp_massey_t B, const fmpz *a,
                                         slong count, const fmpz_mod_ctx_t ctx)
```

```
void fmpz_mod_berlekamp_massey_add_zeros(fmpz_mod_berlekamp_massey_t B, slong count,
                                         const fmpz_mod_ctx_t ctx)
```

```
void fmpz_mod_berlekamp_massey_add_point(fmpz_mod_berlekamp_massey_t B, const fmpz_t a,
                                         const fmpz_mod_ctx_t ctx)
```

Add point(s) to the stream processed by B. The addition of any number of points will not update the V and R polynomial.

```
int fmpz_mod_berlekamp_massey_reduce(fmpz_mod_berlekamp_massey_t B, const
                                     fmpz_mod_ctx_t ctx)
```

Ensure that the polynomials V and R are up to date. The return value is 1 if this function changed V and 0 otherwise. For example, if this function is called twice in a row without adding any points in between, the return of the second call should be 0. As another example, suppose the object is emptied, the points 1, 1, 2, 3 are added, then reduce is called. This reduce should return 1 with $\deg(R) < \deg(V) = 2$ because the Fibonacci sequence has been recognized. The further addition of the two points 5, 8 and a reduce will result in a return value of 0.

```
slong fmpz_mod_berlekamp_massey_point_count(const fmpz_mod_berlekamp_massey_t B)
```

Return the number of points stored in B.

```
const fmpz *fmpz_mod_berlekamp_massey_points(const fmpz_mod_berlekamp_massey_t B)
```

Return a pointer to the array of points stored in B. This may be NULL if `fmpz_mod_berlekamp_massey_point_count()` returns 0.

```
const fmpz_mod_poly_struct *fmpz_mod_berlekamp_massey_V_poly(const
                                                             fmpz_mod_berlekamp_massey_t
                                                             B)
```

Return the polynomial V in B.

```
const fmpz_mod_poly_struct *fmpz_mod_berlekamp_massey_R_poly(const
                                                             fmpz_mod_berlekamp_massey_t
                                                             B)
```

Return the polynomial R in B.

6.14 fmpz_mod_poly_factor.h – factorisation of polynomials over integers mod n

6.14.1 Types, macros and constants

```
type fmpz_mod_poly_factor_struct
```

A structure representing a polynomial in factorised form as a product of polynomials with associated exponents.

```
type fmpz_mod_poly_factor_t
```

An array of length 1 of `fmpz_mod_poly_factor_struct`.

6.14.2 Factorisation

void **fmpz_mod_poly_factor_init**(*fmpz_mod_poly_factor_t* fac, const *fmpz_mod_ctx_t* ctx)
Initialises *fac* for use.

void **fmpz_mod_poly_factor_clear**(*fmpz_mod_poly_factor_t* fac, const *fmpz_mod_ctx_t* ctx)
Frees all memory associated with *fac*.

void **fmpz_mod_poly_factor_realloc**(*fmpz_mod_poly_factor_t* fac, *slong* alloc, const *fmpz_mod_ctx_t* ctx)

Reallocates the factor structure to provide space for precisely *alloc* factors.

void **fmpz_mod_poly_factor_fit_length**(*fmpz_mod_poly_factor_t* fac, *slong* len, const *fmpz_mod_ctx_t* ctx)

Ensures that the factor structure has space for at least *len* factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

void **fmpz_mod_poly_factor_set**(*fmpz_mod_poly_factor_t* res, const *fmpz_mod_poly_factor_t* fac, const *fmpz_mod_ctx_t* ctx)

Sets *res* to the same factorisation as *fac*.

void **fmpz_mod_poly_factor_print**(const *fmpz_mod_poly_factor_t* fac, const *fmpz_mod_ctx_t* ctx)
Prints the entries of *fac* to standard output.

void **fmpz_mod_poly_factor_insert**(*fmpz_mod_poly_factor_t* fac, const *fmpz_mod_poly_t* poly, *slong* exp, const *fmpz_mod_ctx_t* ctx)

Inserts the factor *poly* with multiplicity *exp* into the factorisation *fac*.

If *fac* already contains *poly*, then *exp* simply gets added to the exponent of the existing entry.

void **fmpz_mod_poly_factor_concat**(*fmpz_mod_poly_factor_t* res, const *fmpz_mod_poly_factor_t* fac, const *fmpz_mod_ctx_t* ctx)

Concatenates two factorisations.

This is equivalent to calling *fmpz_mod_poly_factor_insert()* repeatedly with the individual factors of *fac*.

Does not support aliasing between *res* and *fac*.

void **fmpz_mod_poly_factor_pow**(*fmpz_mod_poly_factor_t* fac, *slong* exp, const *fmpz_mod_ctx_t* ctx)

Raises *fac* to the power *exp*.

int **fmpz_mod_poly_is_irreducible**(const *fmpz_mod_poly_t* f, const *fmpz_mod_ctx_t* ctx)
Returns 1 if the polynomial *f* is irreducible, otherwise returns 0.

int **fmpz_mod_poly_is_irreducible_ddf**(const *fmpz_mod_poly_t* f, const *fmpz_mod_ctx_t* ctx)
Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int **fmpz_mod_poly_is_irreducible_rabin**(const *fmpz_mod_poly_t* f, const *fmpz_mod_ctx_t* ctx)
Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses Rabin irreducibility test.

int **fmpz_mod_poly_is_irreducible_rabin_f**(*fmpz_t* r, const *fmpz_mod_poly_t* f, const *fmpz_mod_ctx_t* ctx)

Either sets *r* to 1 and returns 1 if the polynomial *f* is irreducible or 0 otherwise, or sets *r* to a nontrivial factor of *p*.

This algorithm correctly determines whether *f* is irreducible over $\mathbb{Z}/p\mathbb{Z}$, even for composite *f*, or it finds a factor of *p*.

```
int _fmpz_mod_poly_is_squarefree(const fmpz *f, slong len, const fmpz_mod_ctx_t ctx)
```

Returns 1 if (f, len) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int _fmpz_mod_poly_is_squarefree_f(fmpz_t fac, const fmpz *f, slong len, const fmpz_mod_ctx_t ctx)
```

If *fac* returns with the value 1 then the function operates as per *_fmpz_mod_poly_is_squarefree()*, otherwise *f* is set to a nontrivial factor of *p*.

```
int fmpz_mod_poly_is_squarefree(const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Returns 1 if *f* is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

```
int fmpz_mod_poly_is_squarefree_f(fmpz_t fac, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

If *fac* returns with the value 1 then the function operates as per *fmpz_mod_poly_is_squarefree()*, otherwise *f* is set to a nontrivial factor of *p*.

```
int fmpz_mod_poly_factor_equal_deg_prob(fmpz_mod_poly_t factor, flint_rand_t state, const fmpz_mod_poly_t pol, slong d, const fmpz_mod_ctx_t ctx)
```

Probabilistic equal degree factorisation of *pol* into irreducible factors of degree *d*. If it passes, a factor is placed in *factor* and 1 is returned, otherwise 0 is returned and the value of *factor* is undetermined.

Requires that *pol* be monic, non-constant and squarefree.

```
void fmpz_mod_poly_factor_equal_deg(fmpz_mod_poly_factor_t factors, const fmpz_mod_poly_t pol, slong d, const fmpz_mod_ctx_t ctx)
```

Assuming *pol* is a product of irreducible factors all of degree *d*, finds all those factors and places them in *factors*. Requires that *pol* be monic, non-constant and squarefree.

```
void fmpz_mod_poly_factor_distinct_deg(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t poly, slong *const *degs, const fmpz_mod_ctx_t ctx)
```

Factorises a monic non-constant squarefree polynomial *poly* of degree *n* into factors *f*[*d*] such that for $1 \leq d \leq n$ *f*[*d*] is the product of the monic irreducible factors of *poly* of degree *d*. Factors *f*[*d*] are stored in *res*, and the degree *d* of the irreducible factors is stored in *degs* in the same order as the factors.

Requires that *degs* has enough space for $(n/2) + 1 * \text{sizeof}(\text{slong})$.

```
void fmpz_mod_poly_factor_distinct_deg_threaded(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t poly, slong *const *degs, const fmpz_mod_ctx_t ctx)
```

Multithreaded version of *fmpz_mod_poly_factor_distinct_deg()*.

```
void fmpz_mod_poly_factor_squarefree(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Sets *res* to a squarefree factorization of *f*.

```
void fmpz_mod_poly_factor(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Factorises a non-constant polynomial *f* into monic irreducible factors choosing the best algorithm for given modulo and degree. Choice is based on heuristic measurements.

```
void fmpz_mod_poly_factor_cantor_zassenhaus(fmpz_mod_poly_factor_t res, const fmpz_mod_poly_t f, const fmpz_mod_ctx_t ctx)
```

Factorises a non-constant polynomial *f* into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void fmpz_mod_poly_factor_kaltofen_shoup(fmpz_mod_poly_factor_t res, const
                                         fmpz_mod_poly_t poly, const fmpz_mod_ctx_t ctx)
```

Factorises a non-constant polynomial *poly* into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a baby step/giant step strategy for the distinct-degree factorization step. If *flint_get_num_threads()* is greater than one *fmpz_mod_poly_factor_distinct_deg_threaded()* is used.

```
void fmpz_mod_poly_factor_berlekamp(fmpz_mod_poly_factor_t factors, const fmpz_mod_poly_t
                                     f, const fmpz_mod_ctx_t ctx)
```

Factorises a non-constant polynomial *f* into monic irreducible factors using the Berlekamp algorithm.

```
void _fmpz_mod_poly_interval_poly_worker(void *arg_ptr)
```

Worker function to compute interval polynomials in distinct degree factorisation. Input/output is stored in *fmpz_mod_poly_interval_poly_arg_t*.

6.14.3 Root Finding

```
void fmpz_mod_poly_roots(fmpz_mod_poly_factor_t r, const fmpz_mod_poly_t f, int
                        with_multiplicity, const fmpz_mod_ctx_t ctx)
```

Fill *r* with factors of the form $x - r_i$ where the r_i are the distinct roots of a nonzero f in $\mathbb{Z}/p\mathbb{Z}$. It is expected and not checked that the modulus of *ctx* is prime. If *with_multiplicity* is zero, the exponent e_i of the factor $x - r_i$ is 1. Otherwise, it is the largest e_i such that $(x - r_i)_i^{e_i}$ divides f . This function throws if f is zero, but is otherwise always successful.

```
int fmpz_mod_poly_roots_factored(fmpz_mod_poly_factor_t r, const fmpz_mod_poly_t f, int
                                with_multiplicity, const fmpz_factor_t n, const
                                fmpz_mod_ctx_t ctx)
```

Fill *r* with factors of the form $x - r_i$ where the r_i are the distinct roots of a nonzero f in $\mathbb{Z}/n\mathbb{Z}$. It is expected and not checked that n is a prime factorization of the modulus of *ctx*. If *with_multiplicity* is zero, the exponent e_i of the factor $x - r_i$ is 1. Otherwise, it is the largest e_i such that $(x - r_i)_i^{e_i}$ divides f . The roots are first found modulo the primes in n , then lifted to the corresponding prime powers, then combined into roots of the original polynomial f . A return of 1 indicates the function was successful. A return of 0 indicates the function was not able to find the roots, possibly because there are too many of them. This function throws if f is zero.

6.15 fmpz_mod_mpoly.h – polynomials over the integers mod n

The exponents follow the *mpoly* interface. A coefficient may be referenced as a *fmpz **, but this may disappear in a future version.

6.15.1 Types, macros and constants

```
type fmpz_mod_mpoly_struct
```

A structure holding a multivariate polynomial over the integers mod n .

```
type fmpz_mod_mpoly_t
```

An array of length 1 of *fmpz_mod_mpoly_ctx_struct*.

```
type fmpz_mod_mpoly_ctx_struct
```

Context structure representing the parent ring of an *fmpz_mod_mpoly*.

```
type fmpz_mod_mpoly_ctx_t
```

An array of length 1 of *fmpz_mod_mpoly_struct*.

6.15.2 Context object

void **fmpz_mod_mpoly_ctx_init**(*fmpz_mod_mpoly_ctx_t* ctx, *slong* nvars, const *ordering_t* ord, const *fmpz_t* p)

Initialise a context object for a polynomial ring modulo n with $nvars$ variables and ordering ord . The possibilities for the ordering are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX.

slong **fmpz_mod_mpoly_ctx_nvars**(const *fmpz_mod_mpoly_ctx_t* ctx)

Return the number of variables used to initialize the context.

ordering_t **fmpz_mod_mpoly_ctx_ord**(const *fmpz_mod_mpoly_ctx_t* ctx)

Return the ordering used to initialize the context.

void **fmpz_mod_mpoly_ctx_get_modulus**(*fmpz_t* n, const *fmpz_mod_mpoly_ctx_t* ctx)

Set n to the modulus used to initialize the context.

void **fmpz_mod_mpoly_ctx_clear**(*fmpz_mod_mpoly_ctx_t* ctx)

Release up any space allocated by an *ctx*.

6.15.3 Memory management

void **fmpz_mod_mpoly_init**(*fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

Initialise A for use with the given an initialised context object. Its value is set to zero.

void **fmpz_mod_mpoly_init2**(*fmpz_mod_mpoly_t* A, *slong* alloc, const *fmpz_mod_mpoly_ctx_t* ctx)

Initialise A for use with the given an initialised context object. Its value is set to zero. It is allocated with space for $alloc$ terms and at least MPOLY_MIN_BITS bits for the exponents.

void **fmpz_mod_mpoly_init3**(*fmpz_mod_mpoly_t* A, *slong* alloc, *flint_bitcnt_t* bits, const *fmpz_mod_mpoly_ctx_t* ctx)

Initialise A for use with the given an initialised context object. Its value is set to zero. It is allocated with space for $alloc$ terms and $bits$ bits for the exponents.

void **fmpz_mod_mpoly_clear**(*fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

Release any space allocated for A .

6.15.4 Input/Output

The variable strings in x start with the variable of most significance at index 0. If x is NULL, the variables are named x_1 , x_2 , etc.

char ***fmpz_mod_mpoly_get_str_pretty**(const *fmpz_mod_mpoly_t* A, const char **x, const *fmpz_mod_mpoly_ctx_t* ctx)

Return a string, which the user is responsible for cleaning up, representing A , given an array of variable strings x .

int **fmpz_mod_mpoly_fprint_pretty**(FILE *file, const *fmpz_mod_mpoly_t* A, const char **x, const *fmpz_mod_mpoly_ctx_t* ctx)

Print a string representing A to *file*.

int **fmpz_mod_mpoly_print_pretty**(const *fmpz_mod_mpoly_t* A, const char **x, const *fmpz_mod_mpoly_ctx_t* ctx)

Print a string representing A to *stdout*.

int **fmpz_mod_mpoly_set_str_pretty**(*fmpz_mod_mpoly_t* A, const char *str, const char **x, const *fmpz_mod_mpoly_ctx_t* ctx)

Set A to the polynomial in the null-terminates string str given an array x of variable strings. If parsing str fails, A is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in x . The character $^$ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

6.15.5 Basic manipulation

void **fmpz_mod_mpoly_gen**(fmpz_mod_mpoly_t A , *long* var , const fmpz_mod_mpoly_ctx_t ctx)

Set A to the variable of index var , where $var = 0$ corresponds to the variable with the most significance with respect to the ordering.

int **fmpz_mod_mpoly_is_gen**(const fmpz_mod_mpoly_t A , *long* var , const fmpz_mod_mpoly_ctx_t ctx)

If $var \geq 0$, return 1 if A is equal to the var -th generator, otherwise return 0 . If $var < 0$, return 1 if the polynomial is equal to any generator, otherwise return 0 .

void **fmpz_mod_mpoly_set**(fmpz_mod_mpoly_t A , const fmpz_mod_mpoly_t B , const fmpz_mod_mpoly_ctx_t ctx)

Set A to B .

int **fmpz_mod_mpoly_equal**(const fmpz_mod_mpoly_t A , const fmpz_mod_mpoly_t B , const fmpz_mod_mpoly_ctx_t ctx)

Return 1 if A is equal to B , else return 0 .

void **fmpz_mod_mpoly_swap**(fmpz_mod_mpoly_t $poly1$, fmpz_mod_mpoly_t $poly2$, const fmpz_mod_mpoly_ctx_t ctx)

Efficiently swap A and B .

6.15.6 Constants

int **fmpz_mod_mpoly_is_fmpz**(const fmpz_mod_mpoly_t A , const fmpz_mod_mpoly_ctx_t ctx)

Return 1 if A is a constant, else return 0 .

void **fmpz_mod_mpoly_get_fmpz**(fmpz_t c , const fmpz_mod_mpoly_t A , const fmpz_mod_mpoly_ctx_t ctx)

Assuming that A is a constant, set c to this constant. This function throws if A is not a constant.

void **fmpz_mod_mpoly_set_fmpz**(fmpz_mod_mpoly_t A , const fmpz_t c , const fmpz_mod_mpoly_ctx_t ctx)

void **fmpz_mod_mpoly_set_ui**(fmpz_mod_mpoly_t A , *ulong* c , const fmpz_mod_mpoly_ctx_t ctx)

void **fmpz_mod_mpoly_set_si**(fmpz_mod_mpoly_t A , *long* c , const fmpz_mod_mpoly_ctx_t ctx)

Set A to the constant c .

void **fmpz_mod_mpoly_zero**(fmpz_mod_mpoly_t A , const fmpz_mod_mpoly_ctx_t ctx)

Set A to the constant 0 .

void **fmpz_mod_mpoly_one**(fmpz_mod_mpoly_t A , const fmpz_mod_mpoly_ctx_t ctx)

Set A to the constant 1 .

int **fmpz_mod_mpoly_equal_fmpz**(const fmpz_mod_mpoly_t A , const fmpz_t c , const fmpz_mod_mpoly_ctx_t ctx)

int **fmpz_mod_mpoly_equal_ui**(const fmpz_mod_mpoly_t A , *ulong* c , const fmpz_mod_mpoly_ctx_t ctx)

```
int fmpz_mod_mpoly_equal_si(const fmpz_mod_mpoly_t A, slong c, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if A is equal to the constant c , else return 0.

```
int fmpz_mod_mpoly_is_zero(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if A is the constant 0, else return 0.

```
int fmpz_mod_mpoly_is_one(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if A is the constant 1, else return 0.

6.15.7 Degrees

```
int fmpz_mod_mpoly_degrees_fit_si(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if the degrees of A with respect to each variable fit into an `slong`, otherwise return 0.

```
void fmpz_mod_mpoly_degrees_fmpz(fmpz **degs, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_degrees_si(slong *degs, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Set $degs$ to the degrees of A with respect to each variable. If A is zero, all degrees are set to -1 .

```
void fmpz_mod_mpoly_degree_fmpz(fmpz_t deg, const fmpz_mod_mpoly_t A, slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

```
slong fmpz_mod_mpoly_degree_si(const fmpz_mod_mpoly_t A, slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

Either return or set deg to the degree of A with respect to the variable of index var . If A is zero, the degree is defined to be -1 .

```
int fmpz_mod_mpoly_total_degree_fits_si(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if the total degree of A fits into an `slong`, otherwise return 0.

```
void fmpz_mod_mpoly_total_degree_fmpz(fmpz_t tdeg, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

```
slong fmpz_mod_mpoly_total_degree_si(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Either return or set $tdeg$ to the total degree of A . If A is zero, the total degree is defined to be -1 .

```
void fmpz_mod_mpoly_used_vars(int *used, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

For each variable index i , set $used[i]$ to nonzero if the variable of index i appears in A and to zero otherwise.

6.15.8 Coefficients

```
void fmpz_mod_mpoly_get_coeff_fmpz_monomial(fmpz_t c, const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t M, const fmpz_mod_mpoly_ctx_t ctx)
```

Assuming that M is a monomial, set c to the coefficient of the corresponding monomial in A . This function throws if M is not a monomial.

```
void fmpz_mod_mpoly_set_coeff_fmpz_monomial(fmpz_mod_mpoly_t A, const fmpz_t c, const fmpz_mod_mpoly_t M, const fmpz_mod_mpoly_ctx_t ctx)
```

Assuming that M is a monomial, set the coefficient of the corresponding monomial in A to c . This function throws if M is not a monomial.

```
void fmpz_mod_mpoly_get_coeff_fmpz_fmpz(fmpz_t c, const fmpz_mod_mpoly_t A, fmpz *const
                                         *exp, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_get_coeff_fmpz_ui(fmpz_t c, const fmpz_mod_mpoly_t A, const ulong *exp,
                                       const fmpz_mod_mpoly_ctx_t ctx)
```

Set c to the coefficient of the monomial with exponent vector exp .

```
void fmpz_mod_mpoly_set_coeff_fmpz_fmpz(fmpz_mod_mpoly_t A, const fmpz_t c, fmpz *const
                                         *exp, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_set_coeff_ui_fmpz(fmpz_mod_mpoly_t A, ulong c, fmpz *const *exp, const
                                       fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_set_coeff_si_fmpz(fmpz_mod_mpoly_t A, slong c, fmpz *const *exp, const
                                       fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_set_coeff_fmpz_ui(fmpz_mod_mpoly_t A, const fmpz_t c, const ulong *exp,
                                       const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_set_coeff_ui_ui(fmpz_mod_mpoly_t A, ulong c, const ulong *exp, const
                                       fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_set_coeff_si_ui(fmpz_mod_mpoly_t A, slong c, const ulong *exp, const
                                       fmpz_mod_mpoly_ctx_t ctx)
```

Set the coefficient of the monomial with exponent vector exp to c .

```
void fmpz_mod_mpoly_get_coeff_vars_ui(fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_t A,
                                       const slong *vars, const ulong *exps, slong length, const
                                       fmpz_mod_mpoly_ctx_t ctx)
```

Set C to the coefficient of A with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \leq nvars(A)$ and that the variables in $vars$ are distinct.

6.15.9 Comparison

```
int fmpz_mod_mpoly_cmp(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                      fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 (resp. -1 , or 0) if A is after (resp. before, same as) B in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when A and B are both monomials.

6.15.10 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

```
int fmpz_mod_mpoly_is_canonical(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if A is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficient, and the terms must be sorted from greatest to least.

```
slong fmpz_mod_mpoly_length(const fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Return the number of terms in A . If the polynomial is in canonical form, this will be the number of nonzero coefficients.

```
void fmpz_mod_mpoly_resize(fmpz_mod_mpoly_t A, slong new_length, const
                          fmpz_mod_mpoly_ctx_t ctx)
```

Set the length of A to new_length . Terms are either deleted from the end, or new zero terms are appended.

```
void fmpz_mod_mpoly_get_term_coeff_fmpz(fmpz_t c, const fmpz_mod_mpoly_t A, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Set c to the coefficient of the term of index i .

```
void fmpz_mod_mpoly_set_term_coeff_fmpz(fmpz_mod_mpoly_t A, slong i, const fmpz_t c, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_set_term_coeff_ui(fmpz_mod_mpoly_t A, slong i, ulong c, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_set_term_coeff_si(fmpz_mod_mpoly_t A, slong i, slong c, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Set the coefficient of the term of index i to c .

```
int fmpz_mod_mpoly_term_exp_fits_si(const fmpz_mod_mpoly_t poly, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
int fmpz_mod_mpoly_term_exp_fits_ui(const fmpz_mod_mpoly_t poly, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if all entries of the exponent vector of the term of index i fit into an `slong` (resp. a `ulong`). Otherwise, return 0.

```
void fmpz_mod_mpoly_get_term_exp_fmpz(fmpz **exp, const fmpz_mod_mpoly_t A, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_get_term_exp_ui(ulong *exp, const fmpz_mod_mpoly_t A, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_get_term_exp_si(slong *exp, const fmpz_mod_mpoly_t A, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Set exp to the exponent vector of the term of index i . The `_ui` (resp. `_si`) version throws if any entry does not fit into a `ulong` (resp. `slong`).

```
ulong fmpz_mod_mpoly_get_term_var_exp_ui(const fmpz_mod_mpoly_t A, slong i, slong var, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
slong fmpz_mod_mpoly_get_term_var_exp_si(const fmpz_mod_mpoly_t A, slong i, slong var, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Return the exponent of the variable var of the term of index i . This function throws if the exponent does not fit into a `ulong` (resp. `slong`).

```
void fmpz_mod_mpoly_set_term_exp_fmpz(fmpz_mod_mpoly_t A, slong i, fmpz *const *exp, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_set_term_exp_ui(fmpz_mod_mpoly_t A, slong i, const ulong *exp, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Set the exponent vector of the term of index i to exp .

```
void fmpz_mod_mpoly_get_term(fmpz_mod_mpoly_t M, const fmpz_mod_mpoly_t A, slong i, const
    fmpz_mod_mpoly_ctx_t ctx)
```

Set M to the term of index i in A .

```
void fmpz_mod_mpoly_get_term_monomial(fmpz_mod_mpoly_t M, const fmpz_mod_mpoly_t A,
    slong i, const fmpz_mod_mpoly_ctx_t ctx)
```

Set M to the monomial of the term of index i in A . The coefficient of M will be one.

```
void fmpz_mod_mpoly_push_term_fmpz_fmpz(fmpz_mod_mpoly_t A, const fmpz_t c, fmpz *const
    *exp, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_push_term_fmpz_ffmpz(fmpz_mod_mpoly_t A, const fmpz_t c, const fmpz
    *exp, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_push_term_ui_fmpz(fmpz_mod_mpoly_t A, ulong c, fmpz *const *exp, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_push_term_ui_ffmpz(fmpz_mod_mpoly_t A, ulong c, const fmpz *exp, const
    fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_push_term_si_fmpz(fmpz_mod_mpoly_t A, slong c, fmpz *const *exp, const
                                     fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_si_ffmpz(fmpz_mod_mpoly_t A, slong c, const fmpz *exp, const
                                       fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_fmpz_ui(fmpz_mod_mpoly_t A, const fmpz_t c, const ulong *exp,
                                      const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_ui_ui(fmpz_mod_mpoly_t A, ulong c, const ulong *exp, const
                                    fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_push_term_si_ui(fmpz_mod_mpoly_t A, slong c, const ulong *exp, const
                                    fmpz_mod_mpoly_ctx_t ctx)
```

Append a term to A with coefficient c and exponent vector exp . This function runs in constant average time.

```
void fmpz_mod_mpoly_sort_terms(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t ctx)
```

Sort the terms of A into the canonical ordering dictated by the ordering in ctx . This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the size of A .

```
void fmpz_mod_mpoly_combine_like_terms(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_ctx_t
                                       ctx)
```

Combine adjacent like terms in A and delete terms with coefficient zero. If the terms of A were sorted to begin with, the result will be in canonical form. This function runs in linear time in the size of A .

```
void fmpz_mod_mpoly_reverse(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

Set A to the reversal of B .

6.15.11 Random generation

```
void fmpz_mod_mpoly_randtest_bound(fmpz_mod_mpoly_t A, flint_rand_t state, slong length,
                                   ulong exp_bound, const fmpz_mod_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents in the range $[0, exp\_bound - 1]$. The exponents of each variable are generated by calls to `n_randint(state, exp_bound)`.

```
void fmpz_mod_mpoly_randtest_bounds(fmpz_mod_mpoly_t A, flint_rand_t state, slong length,
                                    ulong *exp_bounds, const fmpz_mod_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents in the range $[0, exp\_bounds[i] - 1]$. The exponents of the variable of index i are generated by calls to `n_randint(state, exp_bounds[i])`.

```
void fmpz_mod_mpoly_randtest_bits(fmpz_mod_mpoly_t A, flint_rand_t state, slong length, ulong
                                  exp_bits, const fmpz_mod_mpoly_ctx_t ctx)
```

Generate a random polynomial with length up to $length$ and exponents whose packed form does not exceed the given bit count.

6.15.12 Addition/Subtraction

```
void fmpz_mod_mpoly_add_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_t
                             c, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_add_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong c, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_add_si(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong c, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B + c$.

```
void fmpz_mod_mpoly_sub_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_t
                             c, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_sub_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong c, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_sub_si(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong c, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B - c$.

```
void fmpz_mod_mpoly_add(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                        fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B + C$.

```
void fmpz_mod_mpoly_sub(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                        fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B - C$.

6.15.13 Scalar operations

```
void fmpz_mod_mpoly_neg(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                        fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $-B$.

```
void fmpz_mod_mpoly_scalar_mul_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                                    fmpz_t c, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_scalar_mul_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong c,
                                  const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_scalar_mul_si(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong c,
                                  const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B \times c$.

```
void fmpz_mod_mpoly_scalar_addmul_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B,
                                       const fmpz_mod_mpoly_t C, const fmpz_t d, const
                                       fmpz_mod_mpoly_ctx_t ctx)
```

Sets A to $B + C \times d$.

```
void fmpz_mod_mpoly_make_monic(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                               fmpz_mod_mpoly_ctx_t ctx)
```

Set A to B divided by the leading coefficient of B . This throws if B is zero or the leading coefficient is not invertible.

6.15.14 Differentiation

```
void fmpz_mod_mpoly_derivative(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, slong var,
                             const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to the derivative of B with respect to the variable of index var .

6.15.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```
void fmpz_mod_mpoly_evaluate_all_fmpz(fmpz_t eval, const fmpz_mod_mpoly_t A, fmpz *const
                                     *vals, const fmpz_mod_mpoly_ctx_t ctx)
```

Set ev to the evaluation of A where the variables are replaced by the corresponding elements of the array $vals$.

```
void fmpz_mod_mpoly_evaluate_one_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B,
                                     slong var, const fmpz_t val, const
                                     fmpz_mod_mpoly_ctx_t ctx)
```

Set A to the evaluation of B where the variable of index var is replaced by val . Return 1 for success and 0 for failure.

```
int fmpz_mod_mpoly_compose_fmpz_poly(fmpz_poly_t A, const fmpz_mod_mpoly_t B,
                                     fmpz_poly_struct *const *C, const
                                     fmpz_mod_mpoly_ctx_t ctxB)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . The context object of B is $ctxB$. Return 1 for success and 0 for failure.

```
int fmpz_mod_mpoly_compose_fmpz_mod_mpoly_geobucket(fmpz_mod_mpoly_t A, const
                                                    fmpz_mod_mpoly_t B,
                                                    fmpz_mod_mpoly_struct *const *C,
                                                    const fmpz_mod_mpoly_ctx_t ctxB,
                                                    const fmpz_mod_mpoly_ctx_t ctxAC)
```

```
int fmpz_mod_mpoly_compose_fmpz_mod_mpoly_horner(fmpz_mod_mpoly_t A, const
                                                  fmpz_mod_mpoly_t B,
                                                  fmpz_mod_mpoly_struct *const *C, const
                                                  fmpz_mod_mpoly_ctx_t ctxB, const
                                                  fmpz_mod_mpoly_ctx_t ctxAC)
```

```
int fmpz_mod_mpoly_compose_fmpz_mod_mpoly(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B,
                                           fmpz_mod_mpoly_struct *const *C, const
                                           fmpz_mod_mpoly_ctx_t ctxB, const
                                           fmpz_mod_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . Both A and the elements of C have context object $ctxAC$, while B has context object $ctxB$. The length of the array C is the number of variables in $ctxB$. Neither A nor B is allowed to alias any other polynomial. Return 1 for success and 0 for failure. The main method attempts to perform the calculation using matrices and chooses heuristically between the `geobucket` and `horner` methods if needed.

```
void fmpz_mod_mpoly_compose_fmpz_mod_mpoly_gen(fmpz_mod_mpoly_t A, const
                                              fmpz_mod_mpoly_t B, const slong *c, const
                                              fmpz_mod_mpoly_ctx_t ctxB, const
                                              fmpz_mod_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variable of index i in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array C is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.

6.15.16 Multiplication

```
void fmpz_mod_mpoly_mul(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                        fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B \times C$.

```
void fmpz_mod_mpoly_mul_johnson(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                                fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to $B \times C$ using Johnson's heap-based method.

```
int fmpz_mod_mpoly_mul_dense(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                             fmpz_mod_mpoly_t C, const fmpz_mod_mpoly_ctx_t ctx)
```

Try to set A to $B \times C$ using dense arithmetic. If the return is 0, the operation was unsuccessful. Otherwise, it was successful and the return is 1.

6.15.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int fmpz_mod_mpoly_pow_fmpz(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz_t k,
                            const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

```
int fmpz_mod_mpoly_pow_ui(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, ulong k, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

6.15.18 Division

The division functions assume that the modulus is prime.

```
int fmpz_mod_mpoly_divides(fmpz_mod_mpoly_t Q, const fmpz_mod_mpoly_t A, const
                           fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)
```

If A is divisible by B , set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0.

```
void fmpz_mod_mpoly_div(fmpz_mod_mpoly_t Q, const fmpz_mod_mpoly_t A, const
                        fmpz_mod_mpoly_t B, const fmpz_mod_mpoly_ctx_t ctx)
```

Set Q to the quotient of A by B , discarding the remainder.

```
void fmpz_mod_mpoly_divrem(fmpz_mod_mpoly_t Q, fmpz_mod_mpoly_t R, const
                           fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

Set Q and R to the quotient and remainder of A divided by B .

```
void fmpz_mod_mpoly_divrem_ideal(fmpz_mod_mpoly_struct **Q, fmpz_mod_mpoly_t R, const
                                 fmpz_mod_mpoly_t A, fmpz_mod_mpoly_struct *const *B,
                                 slong len, const fmpz_mod_mpoly_ctx_t ctx)
```

This function is as per `fmpz_mod_mpoly_divrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q . The number of divisor (and hence quotient) polynomials, is given by len .

6.15.19 Greatest Common Divisor

void `fmpz_mod_mpoly_term_content`(*fmpz_mod_mpoly_t* M, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

Set M to the GCD of the terms of A . If A is zero, M will be zero. Otherwise, M will be a monomial with coefficient one.

int `fmpz_mod_mpoly_content_vars`(*fmpz_mod_mpoly_t* g, const *fmpz_mod_mpoly_t* A, *slong* *vars, *slong* vars_length, const *fmpz_mod_mpoly_ctx_t* ctx)

Set g to the GCD of the coefficients of A when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, g will be independent of the variables $vars$.

int `fmpz_mod_mpoly_gcd`(*fmpz_mod_mpoly_t* G, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

Try to set G to the monic GCD of A and B . The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and G is left untouched.

int `fmpz_mod_mpoly_gcd_cofactors`(*fmpz_mod_mpoly_t* G, *fmpz_mod_mpoly_t* Abar, *fmpz_mod_mpoly_t* Bbar, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

Do the operation of `fmpz_mod_mpoly_gcd()` and also compute $Abar = A/G$ and $Bbar = B/G$ if successful.

int `fmpz_mod_mpoly_gcd_brown`(*fmpz_mod_mpoly_t* G, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

int `fmpz_mod_mpoly_gcd_hensel`(*fmpz_mod_mpoly_t* G, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

int `fmpz_mod_mpoly_gcd_subresultant`(*fmpz_mod_mpoly_t* G, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

int `fmpz_mod_mpoly_gcd_zippel`(*fmpz_mod_mpoly_t* G, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

int `fmpz_mod_mpoly_gcd_zippel2`(*fmpz_mod_mpoly_t* G, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

Try to set G to the GCD of A and B using various algorithms.

int `fmpz_mod_mpoly_resultant`(*fmpz_mod_mpoly_t* R, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, *slong* var, const *fmpz_mod_mpoly_ctx_t* ctx)

Try to set R to the resultant of A and B with respect to the variable of index var .

int `fmpz_mod_mpoly_discriminant`(*fmpz_mod_mpoly_t* D, const *fmpz_mod_mpoly_t* A, *slong* var, const *fmpz_mod_mpoly_ctx_t* ctx)

Try to set D to the discriminant of A with respect to the variable of index var .

6.15.20 Square Root

The square root functions assume that the modulus is prime for correct operation.

int `fmpz_mod_mpoly_sqrt`(*fmpz_mod_mpoly_t* Q, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

If $Q^2 = A$ has a solution, set Q to a solution and return 1, otherwise return 0 and set Q to zero.

int `fmpz_mod_mpoly_is_square`(const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

Return 1 if A is a perfect square, otherwise return 0.

int `fmpz_mod_mpoly_quadratic_root`(*fmpz_mod_mpoly_t* Q, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_ctx_t* ctx)

If $Q^2 + AQ = B$ has a solution, set Q to a solution and return 1, otherwise return 0.

6.15.21 Univariate Functions

An `fmpz_mod_mpoly_univar_t` holds a univariate polynomial in some main variable with `fmpz_mod_mpoly_t` coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{Z}/n\mathbb{Z}[x_1, \dots, x_m]$ as an element of $(\mathbb{Z}/n\mathbb{Z}[x_1, \dots, x_{v-1}, x_{v+1}, \dots, x_m])[x_v]$ and vice versa.

```
void fmpz_mod_mpoly_univar_init(fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_ctx_t
                               ctx)
```

Initialize A .

```
void fmpz_mod_mpoly_univar_clear(fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_ctx_t
                                 ctx)
```

Clear A .

```
void fmpz_mod_mpoly_univar_swap(fmpz_mod_mpoly_univar_t A, fmpz_mod_mpoly_univar_t B,
                                const fmpz_mod_mpoly_ctx_t ctx)
```

Swap A and B .

```
void fmpz_mod_mpoly_to_univar(fmpz_mod_mpoly_univar_t A, const fmpz_mod_mpoly_t B,
                              slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to a univariate form of B by pulling out the variable of index var . The coefficients of A will still belong to the content ctx but will not depend on the variable of index var .

```
void fmpz_mod_mpoly_from_univar(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_univar_t B,
                                slong var, const fmpz_mod_mpoly_ctx_t ctx)
```

Set A to the normal form of B by putting in the variable of index var . This function is undefined if the coefficients of B depend on the variable of index var .

```
int fmpz_mod_mpoly_univar_degree_fits_si(const fmpz_mod_mpoly_univar_t A, const
                                          fmpz_mod_mpoly_ctx_t ctx)
```

Return 1 if the degree of A with respect to the main variable fits an `slong`. Otherwise, return 0.

```
slong fmpz_mod_mpoly_univar_length(const fmpz_mod_mpoly_univar_t A, const
                                   fmpz_mod_mpoly_ctx_t ctx)
```

Return the number of terms in A with respect to the main variable.

```
slong fmpz_mod_mpoly_univar_get_term_exp_si(fmpz_mod_mpoly_univar_t A, slong i, const
                                             fmpz_mod_mpoly_ctx_t ctx)
```

Return the exponent of the term of index i of A .

```
void fmpz_mod_mpoly_univar_get_term_coeff(fmpz_mod_mpoly_t c, const
                                           fmpz_mod_mpoly_univar_t A, slong i, const
                                           fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_univar_swap_term_coeff(fmpz_mod_mpoly_t c,
                                             fmpz_mod_mpoly_univar_t A, slong i, const
                                             fmpz_mod_mpoly_ctx_t ctx)
```

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A .

```
void fmpz_mod_mpoly_univar_set_coeff_ui(fmpz_mod_mpoly_univar_t Ax, ulong e, const
                                         fmpz_mod_mpoly_t c, const fmpz_mod_mpoly_ctx_t
                                         ctx)
```

Set the coefficient of X^e in Ax to c .

```
int fmpz_mod_mpoly_univar_resultant(fmpz_mod_mpoly_t R, const fmpz_mod_mpoly_univar_t
                                    Ax, const fmpz_mod_mpoly_univar_t Bx, const
                                    fmpz_mod_mpoly_ctx_t ctx)
```

Try to set R to the resultant of Ax and Bx .

```
int fmpz_mod_mpoly_univar_discriminant(fmpz_mod_mpoly_t D, const
                                       fmpz_mod_mpoly_univar_t Ax, const
                                       fmpz_mod_mpoly_ctx_t ctx)
```

Try to set D to the discriminant of Ax .

6.15.22 Vectors

```
type fmpz_mod_mpoly_vec_struct
```

```
type fmpz_mod_mpoly_vec_t
```

A type holding a vector of *fmpz_mod_mpoly_t*.

```
fmpz_mod_mpoly_vec_entry(vec, i)
```

Macro for accessing the entry at position i in *vec*.

```
void fmpz_mod_mpoly_vec_init(fmpz_mod_mpoly_vec_t vec, slong len, const
                             fmpz_mod_mpoly_ctx_t ctx)
```

Initializes *vec* to a vector of length *len*, setting all entries to the zero polynomial.

```
void fmpz_mod_mpoly_vec_clear(fmpz_mod_mpoly_vec_t vec, const fmpz_mod_mpoly_ctx_t ctx)
```

Clears *vec*, freeing its allocated memory.

```
void fmpz_mod_mpoly_vec_print(const fmpz_mod_mpoly_vec_t vec, const fmpz_mod_mpoly_ctx_t
                              ctx)
```

Prints *vec* to standard output.

```
void fmpz_mod_mpoly_vec_swap(fmpz_mod_mpoly_vec_t x, fmpz_mod_mpoly_vec_t y, const
                             fmpz_mod_mpoly_ctx_t ctx)
```

Swaps x and y efficiently.

```
void fmpz_mod_mpoly_vec_fit_length(fmpz_mod_mpoly_vec_t vec, slong len, const
                                   fmpz_mod_mpoly_ctx_t ctx)
```

Allocates room for *len* entries in *vec*.

```
void fmpz_mod_mpoly_vec_set(fmpz_mod_mpoly_vec_t dest, const fmpz_mod_mpoly_vec_t src,
                           const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *dest* to a copy of *src*.

```
void fmpz_mod_mpoly_vec_append(fmpz_mod_mpoly_vec_t vec, const fmpz_mod_mpoly_t f, const
                               fmpz_mod_mpoly_ctx_t ctx)
```

Appends f to the end of *vec*.

```
slong fmpz_mod_mpoly_vec_insert_unique(fmpz_mod_mpoly_vec_t vec, const fmpz_mod_mpoly_t
                                       f, const fmpz_mod_mpoly_ctx_t ctx)
```

Inserts f without duplication into *vec* and returns its index. If this polynomial already exists, *vec* is unchanged. If this polynomial does not exist in *vec*, it is appended.

```
void fmpz_mod_mpoly_vec_set_length(fmpz_mod_mpoly_vec_t vec, slong len, const
                                   fmpz_mod_mpoly_ctx_t ctx)
```

Sets the length of *vec* to *len*, truncating or zero-extending as needed.

```
void fmpz_mod_mpoly_vec_randtest_not_zero(fmpz_mod_mpoly_vec_t vec, flint_rand_t state,
                                           slong len, slong poly_len, ulong exp_bound,
                                           fmpz_mod_mpoly_ctx_t ctx)
```

Sets *vec* to a random vector with exactly *len* entries, all nonzero, with random parameters defined by *poly_len* and *exp_bound*.

6.15.23 Ideals and Gröbner bases

The following methods deal with ideals in $\mathbb{Z}/n\mathbb{Z}[x_1, \dots, x_m]$. We use monic polynomials as normalised generators.

```
void fmpz_mod_mpoly_spoly(fmpz_mod_mpoly_t res, const fmpz_mod_mpoly_t f, const
                        fmpz_mod_mpoly_t g, const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the *S*-polynomial of *f* and *g*, scaled by making *f* and *g* monic first.

```
void fmpz_mod_mpoly_reduction_monics_part(fmpz_mod_mpoly_t res, const fmpz_mod_mpoly_t f,
                                          const fmpz_mod_mpoly_vec_t vec, const
                                          fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the monic remainder of multivariate division with remainder with respect to the polynomials *vec*.

```
int fmpz_mod_mpoly_vec_is_groebner(const fmpz_mod_mpoly_vec_t G, const
                                   fmpz_mod_mpoly_vec_t F, const fmpz_mod_mpoly_ctx_t
                                   ctx)
```

If *F* is *NULL*, checks if *G* is a Gröbner basis. If *F* is not *NULL*, checks if *G* is a Gröbner basis for *F*.

```
int fmpz_mod_mpoly_vec_is_autoreduced(const fmpz_mod_mpoly_vec_t F, const
                                       fmpz_mod_mpoly_ctx_t ctx)
```

Checks whether the vector *F* is autoreduced (or inter-reduced).

```
void fmpz_mod_mpoly_vec_autoreduction(fmpz_mod_mpoly_vec_t H, const
                                       fmpz_mod_mpoly_vec_t F, const
                                       fmpz_mod_mpoly_ctx_t ctx)
```

Sets *H* to the autoreduction (inter-reduction) of *F*.

```
void fmpz_mod_mpoly_vec_autoreduction_groebner(fmpz_mod_mpoly_vec_t H, const
                                                fmpz_mod_mpoly_vec_t G, const
                                                fmpz_mod_mpoly_ctx_t ctx)
```

Sets *H* to the autoreduction (inter-reduction) of *G*. Assumes that *G* is a Gröbner basis. This produces a reduced Gröbner basis, which is unique (up to the sort order of the entries in the vector).

```
void fmpz_mod_mpoly_buchberger_naive(fmpz_mod_mpoly_vec_t G, const
                                      fmpz_mod_mpoly_vec_t F, const fmpz_mod_mpoly_ctx_t
                                      ctx)
```

Sets *G* to a Gröbner basis for *F*, computed using a naive implementation of Buchberger's algorithm.

```
int fmpz_mod_mpoly_buchberger_naive_with_limits(fmpz_mod_mpoly_vec_t G, const
                                                fmpz_mod_mpoly_vec_t F, slong
                                                ideal_len_limit, slong poly_len_limit, const
                                                fmpz_mod_mpoly_ctx_t ctx)
```

As *fmpz_mod_mpoly_buchberger_naive()*, but halts if during the execution of Buchberger's algorithm the length of the ideal basis set exceeds *ideal_len_limit*, or the length of any polynomial exceeds *poly_len_limit*. Returns 1 for success and 0 for failure. On failure, *G* is a valid basis for *F* but it might not be a Gröbner basis.

6.15.24 Converting to/from other polynomial types

```
void fmpz_mod_mpoly_set_fmpz_mpoly(fmpz_mod_mpoly_t A, const fmpz_mpoly_t B, const
                                   fmpz_mod_mpoly_ctx_t ctxm, const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *fmpz_mod_mpoly_t A* to *fmpz_mpoly_t B* with coefficients modulo the modulus in *ctxm*.

```
void fmpz_mod_mpoly_get_fmpz_mpoly(fmpz_mpoly_t A, const fmpz_mod_mpoly_t B, const
                                   fmpz_mod_mpoly_ctx_t ctx)
```

Sets *fmpz_mpoly_t A* to the *fmpz_mod_mpoly_t B*.

6.15.25 Internal Functions

```
void fmpz_mod_mpoly_inflate(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz
                           *shift, const fmpz *stride, const fmpz_mod_mpoly_ctx_t ctx)
```

Apply the function $e \rightarrow \text{shift}[v] + \text{stride}[v] * e$ to each exponent e corresponding to the variable v . It is assumed that each shift and stride is not negative.

```
void fmpz_mod_mpoly_deflate(fmpz_mod_mpoly_t A, const fmpz_mod_mpoly_t B, const fmpz
                           *shift, const fmpz *stride, const fmpz_mod_mpoly_ctx_t ctx)
```

Apply the function $e \rightarrow (e - \text{shift}[v]) / \text{stride}[v]$ to each exponent e corresponding to the variable v . If any $\text{stride}[v]$ is zero, the corresponding numerator $e - \text{shift}[v]$ is assumed to be zero, and the quotient is defined as zero. This allows the function to undo the operation performed by *fmpz_mod_mpoly_inflate()* when possible.

```
void fmpz_mod_mpoly_deflation(fmpz *shift, fmpz *stride, const fmpz_mod_mpoly_t A, const
                             fmpz_mod_mpoly_ctx_t ctx)
```

For each variable v let S_v be the set of exponents appearing on v . Set $\text{shift}[v]$ to $\min(S_v)$ and set $\text{stride}[v]$ to $\gcd(S - \min(S_v))$. If A is zero, all shifts and strides are set to zero.

6.16 fmpz_mod_mpoly_factor.h – factorisation of multivariate polynomials over the integers mod n

6.16.1 Types, macros and constants

```
type fmpz_mod_mpoly_factor_struct
```

A struct for holding a factored polynomial over the integers mod n . There is a single constant and a product of bases to corresponding exponents.

```
type fmpz_mod_mpoly_factor_t
```

An array of length 1 of *fmpz_mod_mpoly_factor_struct*.

6.16.2 Memory management

```
void fmpz_mod_mpoly_factor_init(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t
                               ctx)
```

Initialise f .

```
void fmpz_mod_mpoly_factor_clear(fmpz_mod_mpoly_factor_t f, const fmpz_mod_mpoly_ctx_t
                                ctx)
```

Clear f .

6.16.3 Basic manipulation

void **fmpz_mod_mpoly_factor_swap**(*fmpz_mod_mpoly_factor_t* f, *fmpz_mod_mpoly_factor_t* g, const *fmpz_mod_mpoly_ctx_t* ctx)

Efficiently swap f and g .

slong **fmpz_mod_mpoly_factor_length**(const *fmpz_mod_mpoly_factor_t* f, const *fmpz_mod_mpoly_ctx_t* ctx)

Return the length of the product in f .

void **fmpz_mod_mpoly_factor_get_constant_fmpz**(*fmpz_t* c, const *fmpz_mod_mpoly_factor_t* f, const *fmpz_mod_mpoly_ctx_t* ctx)

Set c to the constant of f .

void **fmpz_mod_mpoly_factor_get_base**(*fmpz_mod_mpoly_t* B, const *fmpz_mod_mpoly_factor_t* f, *slong* i, const *fmpz_mod_mpoly_ctx_t* ctx)

void **fmpz_mod_mpoly_factor_swap_base**(*fmpz_mod_mpoly_t* B, *fmpz_mod_mpoly_factor_t* f, *slong* i, const *fmpz_mod_mpoly_ctx_t* ctx)

Set (resp. swap) B to (resp. with) the base of the term of index i in f .

slong **fmpz_mod_mpoly_factor_get_exp_si**(*fmpz_mod_mpoly_factor_t* f, *slong* i, const *fmpz_mod_mpoly_ctx_t* ctx)

Return the exponent of the term of index i in f . It is assumed to fit an *slong*.

void **fmpz_mod_mpoly_factor_sort**(*fmpz_mod_mpoly_factor_t* f, const *fmpz_mod_mpoly_ctx_t* ctx)

Sort the product of f first by exponent and then by base.

6.16.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A : f is simply set to a factorisation of A , and thus these functions should not depend on the initial value of the output f .

int **fmpz_mod_mpoly_factor_squarefree**(*fmpz_mod_mpoly_factor_t* f, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call **fmpz_mod_mpoly_factor_sort()** and then multiply the bases with the desired exponent.

int **fmpz_mod_mpoly_factor**(*fmpz_mod_mpoly_factor_t* f, const *fmpz_mod_mpoly_t* A, const *fmpz_mod_mpoly_ctx_t* ctx)

Set f to a factorization of A where the bases are irreducible.

6.17 fmpz_mod_mpoly_q.h – multivariate rational functions over $\mathbb{Z}/m\mathbb{Z}$

An *fmpz_mod_mpoly_q_t* represents an element of $\mathbb{F}_m(x_1, \dots, x_n)$ for some prime number m and fixed n as a pair of FLINT multivariate polynomials (*fmpz_mod_mpoly_t*). Instances are always kept in canonical form by ensuring that the GCD of numerator and denominator is 1 and then normalizing denominator to a monic polynomial.

The user must create a multivariate polynomial context (*fmpz_mod_mpoly_ctx_t*) specifying the prime number m for the field $\mathbb{F}_m$ the number of variables n and the monomial ordering. The user is responsible for verifying that m is a prime number; if m is composite, undefined behaviour may occur.

6.17.1 Types and macros

type `fmpz_mod_mpoly_q_struct`

type `fmpz_mod_mpoly_q_t`

An `fmpz_mod_mpoly_q_struct` consists of a pair of `fmpz_mod_mpoly_struct`s. An `fmpz_mod_mpoly_q_t` is defined as an array of length one of type `fmpz_mod_mpoly_q_struct`, permitting an `fmpz_mod_mpoly_q_t` to be passed by reference.

`fmpz_mod_mpoly_q_numref(x)`

Macro returning a pointer to the numerator of x which can be used as an `fmpz_mod_mpoly_t`.

`fmpz_mod_mpoly_q_denref(x)`

Macro returning a pointer to the denominator of x which can be used as an `fmpz_mod_mpoly_t`.

6.17.2 Memory management

void `fmpz_mod_mpoly_q_init(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_ctx_t ctx)`

Initializes res for use, and sets its value to zero.

void `fmpz_mod_mpoly_q_clear(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_ctx_t ctx)`

Clears res , freeing or recycling its allocated memory.

6.17.3 Assignment

void `fmpz_mod_mpoly_q_swap(fmpz_mod_mpoly_q_t x, fmpz_mod_mpoly_q_t y, const fmpz_mod_mpoly_ctx_t ctx)`

Swaps the values of x and y efficiently.

void `fmpz_mod_mpoly_q_set(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const fmpz_mod_mpoly_ctx_t ctx)`

int `fmpz_mod_mpoly_q_set_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_t x, const fmpz_mod_mpoly_ctx_t ctx)`

void `fmpz_mod_mpoly_q_set_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_t x, const fmpz_mod_mpoly_ctx_t ctx)`

void `fmpz_mod_mpoly_q_set_si(fmpz_mod_mpoly_q_t res, slong x, const fmpz_mod_mpoly_ctx_t ctx)`

Sets res to the value x . The `fmpz` version returns 1 if the denominator of x is invertible, otherwise returns 0.

6.17.4 Canonicalisation

void `fmpz_mod_mpoly_q_canonicalise(fmpz_mod_mpoly_q_t x, const fmpz_mod_mpoly_ctx_t ctx)`

Puts the numerator and denominator of x in canonical form by removing common content and making the denominator monic.

int `fmpz_mod_mpoly_q_is_canonical(const fmpz_mod_mpoly_q_t x, const fmpz_mod_mpoly_ctx_t ctx)`

Returns whether x is in canonical form.

In addition to verifying that the numerator and denominator have no common content and that the denominator is monic, this function checks that the denominator is nonzero and that the numerator and denominator have correctly sorted terms (these properties should normally hold; verifying them provides an extra consistency check for test code).

6.17.5 Properties

int **fmpz_mod_mpoly_q_is_zero**(const *fmpz_mod_mpoly_q_t* x, const *fmpz_mod_mpoly_ctx_t* ctx)

Returns whether x is the constant 0.

int **fmpz_mod_mpoly_q_is_one**(const *fmpz_mod_mpoly_q_t* x, const *fmpz_mod_mpoly_ctx_t* ctx)

Returns whether x is the constant 1.

void **fmpz_mod_mpoly_q_used_vars**(int *used, const *fmpz_mod_mpoly_q_t* f, const *fmpz_mod_mpoly_ctx_t* ctx)

void **fmpz_mod_mpoly_q_used_vars_num**(int *used, const *fmpz_mod_mpoly_q_t* f, const *fmpz_mod_mpoly_ctx_t* ctx)

void **fmpz_mod_mpoly_q_used_vars_den**(int *used, const *fmpz_mod_mpoly_q_t* f, const *fmpz_mod_mpoly_ctx_t* ctx)

For each variable, sets the corresponding entry in *used* to the boolean flag indicating whether that variable appears in the rational function (respectively its numerator or denominator).

6.17.6 Special values

void **fmpz_mod_mpoly_q_zero**(*fmpz_mod_mpoly_q_t* res, const *fmpz_mod_mpoly_ctx_t* ctx)

Sets *res* to the constant 0.

void **fmpz_mod_mpoly_q_one**(*fmpz_mod_mpoly_q_t* res, const *fmpz_mod_mpoly_ctx_t* ctx)

Sets *res* to the constant 1.

void **fmpz_mod_mpoly_q_gen**(*fmpz_mod_mpoly_q_t* res, *long* i, const *fmpz_mod_mpoly_ctx_t* ctx)

Sets *res* to the generator x_{i+1} . Requires $0 \leq i < n$ where n is the number of variables of *ctx*.

6.17.7 Input and output

The variable strings in x start with the variable of most significance at index 0. If x is *NULL*, the variables are named x_1 , x_2 , etc.

void **fmpz_mod_mpoly_q_print_pretty**(const *fmpz_mod_mpoly_q_t* f, const char **x, const *fmpz_mod_mpoly_ctx_t* ctx)

Prints *res* to standard output. If x is not *NULL*, the strings in x are used as the symbols for the variables.

char ***fmpz_mod_mpoly_q_get_str_pretty**(const *fmpz_mod_mpoly_q_t* f, const char **x, const *fmpz_mod_mpoly_ctx_t* ctx)

Return a string, which the user is responsible for cleaning up, representing f , given an array of variable strings x .

int **fmpz_mod_mpoly_q_set_str_pretty**(*fmpz_mod_mpoly_q_t* res, const char *s, const char **x, *fmpz_mod_mpoly_ctx_t* ctx)

Set *res* to the fraction in the null-terminated string *str* given an array x of variable strings. If parsing *str* fails, *res* is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in x . The character $\wedge$ must be immediately followed by the (integer) exponent. If division by zero occurs, parsing fails.

6.17.8 Random generation

```
void fmpz_mod_mpoly_q_randtest(fmpz_mod_mpoly_q_t res, flint_rand_t state, slong length, slong
                               exp_bound, const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to a random rational function where both numerator and denominator have up to *length* terms and exponents strictly smaller than *exp_bound*.

6.17.9 Comparisons

```
int fmpz_mod_mpoly_q_equal(const fmpz_mod_mpoly_q_t x, const fmpz_mod_mpoly_q_t y, const
                           fmpz_mod_mpoly_ctx_t ctx)
```

Returns whether *x* and *y* are equal.

6.17.10 Arithmetic

The functions below which coerce from an *fmpq* or divide by an integer type perform error handling by returning 1 if the denominator is invertible and 0 otherwise.

```
void fmpz_mod_mpoly_q_neg(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                          fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the negation of *x*.

```
void fmpz_mod_mpoly_q_add(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                          fmpz_mod_mpoly_q_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
int fmpz_mod_mpoly_q_add_fmpq(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpq_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_add_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_add_fmpz_mod(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x,
                                    const fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_add_si(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, slong y,
                             const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the sum of *x* and *y*.

```
void fmpz_mod_mpoly_q_sub(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                          fmpz_mod_mpoly_q_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
int fmpz_mod_mpoly_q_sub_fmpq(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpq_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_sub_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_sub_fmpz_mod(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x,
                                    const fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_sub_si(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, slong y,
                             const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the difference of *x* and *y*.

```
void fmpz_mod_mpoly_q_mul(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                          fmpz_mod_mpoly_q_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
int fmpz_mod_mpoly_q_mul_fmpq(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpq_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_mul_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
```

```
void fmpz_mod_mpoly_q_mul_fmpz_mod(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x,
                                   const fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
void fmpz_mod_mpoly_q_mul_si(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, slong y,
                             const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the product of *x* and *y*.

```
void fmpz_mod_mpoly_q_div(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                          fmpz_mod_mpoly_q_t y, const fmpz_mod_mpoly_ctx_t ctx)
int fmpz_mod_mpoly_q_div_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
int fmpz_mod_mpoly_q_div_fmpz(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                              fmpz_t y, const fmpz_mod_mpoly_ctx_t ctx)
int fmpz_mod_mpoly_q_div_si(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, slong y,
                            const fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the quotient of *x* and *y*.

```
void fmpz_mod_mpoly_q_inv(fmpz_mod_mpoly_q_t res, const fmpz_mod_mpoly_q_t x, const
                          fmpz_mod_mpoly_ctx_t ctx)
```

Sets *res* to the inverse of *x*. Division by zero calls *flint_abort*.

GROUPS AND OTHER STRUCTURES

7.1 perm.h – permutations

7.1.1 Memory management

slong *_perm_init(*slong* n)
Initialises the permutation for use.

void _perm_clear(*slong* *vec)
Clears the permutation.

7.1.2 Assignment

void _perm_set(*slong* *res, const *slong* *vec, *slong* n)
Sets the permutation **res** to the same as the permutation **vec**.

void _perm_one(*slong* *vec, *slong* n)
Sets the permutation to the identity permutation.

void _perm_inv(*slong* *res, const *slong* *vec, *slong* n)
Sets **res** to the inverse permutation of **vec**. Allows aliasing of **res** and **vec**.

7.1.3 Composition

void _perm_compose(*slong* *res, const *slong* *vec1, const *slong* *vec2, *slong* n)
Forms the composition $\pi_1 \circ \pi_2$ of two permutations π_1 and π_2 . Here, π_2 is applied first, that is, $(\pi_1 \circ \pi_2)(i) = \pi_1(\pi_2(i))$.
Allows aliasing of **res**, **vec1** and **vec2**.

7.1.4 Parity

int _perm_parity(const *slong* *vec, *slong* n)
Returns the parity of **vec**, 0 if the permutation is even and 1 if the permutation is odd.

7.1.5 Randomisation

int **_perm_randtest**(*slong* *vec, *slong* n, *flint_rand_t* state)

Generates a random permutation vector of length n and returns its parity, 0 or 1.

This function uses the Knuth shuffle algorithm to generate a uniformly random permutation without retries.

7.2 qfb.h – binary quadratic forms

Authors:

- William Hart
- Håvard Damm-Johnsen (updated documentation)

7.2.1 Introduction

This module contains functionality for creating, listing and reducing binary quadratic forms. A **qfb** struct consists of three **fmpz_t** s , a , b and c , and basic algorithms for operations such as reduction, composition and enumerating are implemented and described below.

Currently the code only works for definite binary quadratic forms.

7.2.2 Memory management

void **qfb_init**(qfb_t q)

Initialise a **qfb_t** q for use.

void **qfb_clear**(qfb_t q)

Clear a **qfb_t** after use. This releases any memory allocated for q back to flint.

void **qfb_array_clear**(qfb **forms, *slong* num)

Clean up an array of **qfb** structs allocated by a **qfb** function. The parameter **num** must be set to the length of the array.

7.2.3 Hash table

qfb_hash_t ***qfb_hash_init**(*slong* depth)

Initialises a hash table of size 2^{depth} .

void **qfb_hash_clear**(qfb_hash_t *qhash, *slong* depth)

Frees all memory used by a hash table of size 2^{depth} .

void **qfb_hash_insert**(qfb_hash_t *qhash, qfb_t q, qfb_t q2, *slong* iter, *slong* depth)

Insert the binary quadratic form q into the given hash table of size 2^{depth} in the field q of the hash structure. Also store the second binary quadratic form q_2 (if not NULL) in the similarly named field and **iter** in the similarly named field of the hash structure.

slong **qfb_hash_find**(qfb_hash_t *qhash, qfb_t q, *slong* depth)

Search for the given binary quadratic form or its inverse in the given hash table of size 2^{depth} . If it is found, return the index in the table (which is an array of **qfb_hash_t** structs), otherwise return -1.

7.2.4 Basic manipulation

void **qfb_set**(qfb_t f, qfb_t g)

Set the binary quadratic form f to be equal to g .

7.2.5 Comparison

int **qfb_equal**(qfb_t f, qfb_t g)

Returns 1 if f and g are identical binary quadratic forms, otherwise returns 0.

7.2.6 Input/output

void **qfb_print**(qfb_t q)

Print a binary quadratic form q in the format (a, b, c) where a, b, c are the entries of q .

7.2.7 Computing with forms

void **qfb_discriminant**(fmpz_t D, qfb_t f)

Set D to the discriminant of the binary quadratic form f , i.e. to $b^2 - 4ac$, where $f = (a, b, c)$.

void **qfb_reduce**(qfb_t r, qfb_t f, fmpz_t D)

Set r to a reduced form equivalent to the binary quadratic form f of discriminant D .

int **qfb_is_reduced**(qfb_t r)

Returns 1 if q is a reduced binary quadratic form, otherwise returns 0. Note that this only tests for definite quadratic forms, so a form $r = (a, b, c)$ is reduced if and only if $|b| \leq a \leq c$ and if either inequality is an equality, then $b \geq 0$.

slong **qfb_reduced_forms**(qfb **forms, *slong* d)

Given a discriminant d (negative for negative definite forms), compute all the reduced binary quadratic forms of that discriminant. The function allocates space for these and returns it in the variable **forms** (the user is responsible for cleaning this up by a single call to **qfb_array_clear** on **forms**, after use.) The function returns the number of forms generated (the form class number). The forms are stored in an array of **qfb** structs, which contain fields **a**, **b**, **c** corresponding to forms (a, b, c) .

slong **qfb_reduced_forms_large**(qfb **forms, *slong* d)

As for **qfb_reduced_forms**. However, for small $|d|$ it requires fewer primes to be computed at a small cost in speed. It is called automatically by **qfb_reduced_forms** for large $|d|$ so that **flint_primes** is not exhausted.

void **qfb_nucomp**(qfb_t r, const qfb_t f, const qfb_t g, fmpz_t D, fmpz_t L)

Shanks' NUCOMP as described in [JvdP2002].

Computes the near reduced composition of forms f and g given $L = \lfloor |D|^{1/4} \rfloor$ where D is the common discriminant of f and g . The result is returned in r .

We require that f is a primitive form.

void **qfb_nudup1**(qfb_t r, const qfb_t f, fmpz_t D, fmpz_t L)

As for **nucomp** except that the form f is composed with itself. We require that f is a primitive form.

void **qfb_pow_ui**(qfb_t r, qfb_t f, fmpz_t D, *ulong* exp)

Compute the near reduced form r which is the result of composing the principal form (identity) with f exp times.

We require D to be set to the discriminant of f and that f is a primitive form.

void **qfb_pow**(qfb_t r, qfb_t f, *fmpz_t* D, *fmpz_t* exp)

As per **qfb_pow_ui**.

void **qfb_inverse**(qfb_t r, qfb_t f)

Set r to the inverse of the binary quadratic form f .

int **qfb_is_principal_form**(qfb_t f, *fmpz_t* D)

Return 1 if f is the reduced principal form of discriminant D , i.e. the identity in the form class group, else 0.

void **qfb_principal_form**(qfb_t f, *fmpz_t* D)

Set f to the principal form of discriminant D , i.e. the identity in the form class group.

int **qfb_is_primitive**(qfb_t f)

Return 1 if f is primitive, i.e. the greatest common divisor of its three coefficients is 1. Otherwise the function returns 0.

void **qfb_prime_form**(qfb_t r, *fmpz_t* D, *fmpz_t* p)

Sets r to the unique prime (p, b, c) of discriminant D , i.e. with $0 < b \leq p$. We require that p is a prime.

int **qfb_exponent_element**(*fmpz_t* exponent, qfb_t f, *fmpz_t* n, *ulong* B1, *ulong* B2_sqrt)

Find the exponent of the element f in the form class group of forms of discriminant n , doing a stage 1 with primes up to at least B1 and a stage 2 for a single large prime up to at least the square of B2_sqrt. If the function fails to find the exponent it returns 0, otherwise the function returns 1 and **exponent** is set to the exponent of f , i.e. the minimum power of f which gives the identity.

It is assumed that the form f is reduced. We require that **iters** is a power of 2 and that **iters** ≥ 1024 .

The function performs a stage 2 which stores up to $4 \times \text{iters}$ binary quadratic forms, and $12 \times \text{iters}$ additional limbs of data in a hash table, where **iters** is the square root of B2.

int **qfb_exponent**(*fmpz_t* exponent, *fmpz_t* n, *ulong* B1, *ulong* B2_sqrt, *slong* c)

Compute the exponent of the class group of discriminant n , doing a stage 1 with primes up to at least B1 and a stage 2 for a single large prime up to at least the square of B2_sqrt, and with probability at least $1 - 2^{-c}$. If the prime limits are exhausted without finding the exponent, the function returns 0, otherwise it returns 1 and **exponent** is set to the computed exponent, i.e. the minimum power to which every element of the class group has to be raised in order to get the identity.

The function performs a stage 2 which stores up to $4 \times \text{iters}$ binary quadratic forms, and $12 \times \text{iters}$ additional limbs of data in a hash table, where **iters** is the square root of B2.

We use algorithm 8.1 of [Sut2007].

int **qfb_exponent_grh**(*fmpz_t* exponent, *fmpz_t* n, *ulong* B1, *ulong* B2_sqrt)

Similar to **qfb_exponent** except that the bound c is automatically generated such that the exponent is guaranteed to be correct, if found, assuming the GRH, namely that the class group is generated by primes less than $6 \log^2(|n|)$ as described in [BD1992].

7.3 dirichlet.h – Dirichlet characters

Warning: the interfaces in this module are experimental and may change without notice.

This module allows working with Dirichlet characters algebraically. For evaluations of characters as complex numbers, see [acb_dirichlet.h – Dirichlet L-functions, Riemann zeta and related functions](#).

7.3.1 Dirichlet characters

Working with Dirichlet characters mod q consists mainly in going from residue classes mod q to exponents on a set of generators of the group.

This implementation relies on the Conrey numbering scheme introduced in the [L-functions and Modular Forms DataBase](#), which is an explicit choice of generators allowing to represent Dirichlet characters via the pairing

$$\begin{array}{ccc} (\mathbb{Z}/q\mathbb{Z})^\times \times (\mathbb{Z}/q\mathbb{Z})^\times & \rightarrow & \bigoplus_i \mathbb{Z}/\phi_i\mathbb{Z} \times \mathbb{Z}/\phi_i\mathbb{Z} \rightarrow \mathbb{C} \\ (m, n) & \mapsto & (a_i, b_i) \mapsto \chi_q(m, n) = \exp(2i\pi \sum \frac{a_i b_i}{\phi_i}) \end{array}$$

We call *number* a residue class m modulo q , and *log* the corresponding vector (a_i) of exponents of Conrey generators.

Going from a *log* to the corresponding *number* is a cheap operation we call *exponential*, while the converse requires computing discrete logarithms.

7.3.2 Multiplicative group modulo q

type `dirichlet_group_struct`

type `dirichlet_group_t`

Represents the group of Dirichlet characters mod q .

An `dirichlet_group_t` is defined as an array of `dirichlet_group_struct` of length 1, permitting it to be passed by reference.

int `dirichlet_group_init`(`dirichlet_group_t` G , `ulong` q)

Initializes G to the group of Dirichlet characters mod q .

This method computes a canonical decomposition of G in terms of cyclic groups, which are the mod p^e subgroups for $p^e \parallel q$, plus the specific generator described by Conrey for each subgroup.

In particular G contains:

- the number *num* of components
- the generators
- the exponent *expo* of the group

It does *not* automatically precompute lookup tables of discrete logarithms or numerical roots of unity, and can therefore safely be called even with large q .

For implementation reasons, the largest prime factor of q must not exceed 10^{16} . This restriction could be removed in the future. The function returns 1 on success and 0 if a factor is too large.

void `dirichlet_subgroup_init`(`dirichlet_group_t` H , const `dirichlet_group_t` G , `ulong` h)

Given an already computed group G mod q , initialize its subgroup H defined mod $h \mid q$. Precomputed discrete log tables are inherited.

void `dirichlet_group_clear`(`dirichlet_group_t` G)

Clears G . Remark this function does *not* clear the discrete logarithm tables stored in G (which may be shared with another group).

`ulong dirichlet_group_size(const dirichlet_group_t G)`

Returns the number of elements in G , i.e. $\varphi(q)$.

`ulong dirichlet_group_num_primitive(const dirichlet_group_t G)`

Returns the number of primitive elements in G .

`void dirichlet_group_dlog_precompute(dirichlet_group_t G, ulong num)`

Precompute decomposition and tables for discrete log computations in G , so as to minimize the complexity of num calls to discrete logarithms.

If num gets very large, the entire group may be indexed.

`void dirichlet_group_dlog_clear(dirichlet_group_t G)`

Clear discrete logarithm tables in G . When discrete logarithm tables are shared with subgroups, those subgroups must be cleared before clearing the tables.

7.3.3 Character type

type `dirichlet_char_struct`

type `dirichlet_char_t`

Represents a Dirichlet character. This structure contains both a *number* (residue class) and the corresponding *log* (exponents on the group generators).

An `dirichlet_char_t` is defined as an array of `dirichlet_char_struct` of length 1, permitting it to be passed by reference.

`void dirichlet_char_init(dirichlet_char_t chi, const dirichlet_group_t G)`

Initializes *chi* to an element of the group G and sets its value to the principal character.

`void dirichlet_char_clear(dirichlet_char_t chi)`

Clears *chi*.

`void dirichlet_char_print(const dirichlet_group_t G, const dirichlet_char_t chi)`

Prints the array of exponents representing this character.

`void dirichlet_char_log(dirichlet_char_t x, const dirichlet_group_t G, ulong m)`

Sets x to the character of number m , computing its log using discrete logarithm in G .

`ulong dirichlet_char_exp(const dirichlet_group_t G, const dirichlet_char_t x)`

Returns the number m corresponding to exponents in x .

`ulong _dirichlet_char_exp(dirichlet_char_t x, const dirichlet_group_t G)`

Computes and returns the number m corresponding to exponents in x . This function is for internal use.

`void dirichlet_char_one(dirichlet_char_t x, const dirichlet_group_t G)`

Sets x to the principal character in G , having $\log [0, \dots, 0]$.

`void dirichlet_char_first_primitive(dirichlet_char_t x, const dirichlet_group_t G)`

Sets x to the first primitive character of G , having $\log [1, \dots, 1]$, or $[0, 1, \dots, 1]$ if $8 \mid q$.

`void dirichlet_char_set(dirichlet_char_t x, const dirichlet_group_t G, const dirichlet_char_t y)`

Sets x to the element y .

`int dirichlet_char_next(dirichlet_char_t x, const dirichlet_group_t G)`

Sets x to the next character in G according to lexicographic ordering of $\log$.

The return value is the index of the last updated exponent of x , or -1 if the last element has been reached.

This function allows to iterate on all elements of G looping on their $\log$. Note that it produces elements in seemingly random *number* order.

The following template can be used for such a loop:

```
dirichlet_char_one(chi, G);
do {
    /* use character chi */
} while (dirichlet_char_next(chi, G) >= 0);
```

int **dirichlet_char_next_primitive**(*dirichlet_char_t* x, const *dirichlet_group_t* G)

Same as *dirichlet_char_next()*, but jumps to the next primitive character of G .

ulong **dirichlet_index_char**(const *dirichlet_group_t* G, const *dirichlet_char_t* x)

Returns the lexicographic index of the $\log$ of x as an integer in $0 \dots \varphi(q)$.

void **dirichlet_char_index**(*dirichlet_char_t* x, const *dirichlet_group_t* G, ulong j)

Sets x to the character whose $\log$ has lexicographic index j .

int **dirichlet_char_eq**(const *dirichlet_char_t* x, const *dirichlet_char_t* y)

int **dirichlet_char_eq_deep**(const *dirichlet_group_t* G, const *dirichlet_char_t* x, const *dirichlet_char_t* y)

Return 1 if x equals y .

The second version checks every byte of the representation and is intended for testing only.

7.3.4 Character properties

As a consequence of the Conrey numbering, all these numbers are available at the level of *number* and *char* object. Both case require no discrete log computation.

int **dirichlet_char_is_principal**(const *dirichlet_group_t* G, const *dirichlet_char_t* chi)

Returns 1 if *chi* is the principal character mod q .

ulong **dirichlet_conductor_ui**(const *dirichlet_group_t* G, ulong a)

ulong **dirichlet_conductor_char**(const *dirichlet_group_t* G, const *dirichlet_char_t* x)

Returns the *conductor* of $\chi_q(a, \cdot)$, that is the smallest r dividing q such $\chi_q(a, \cdot)$ can be obtained as a character mod r .

int **dirichlet_parity_ui**(const *dirichlet_group_t* G, ulong a)

int **dirichlet_parity_char**(const *dirichlet_group_t* G, const *dirichlet_char_t* x)

Returns the *parity* λ in $\{0, 1\}$ of $\chi_q(a, \cdot)$, such that $\chi_q(a, -1) = (-1)^\lambda$.

ulong **dirichlet_order_ui**(const *dirichlet_group_t* G, ulong a)

ulong **dirichlet_order_char**(const *dirichlet_group_t* G, const *dirichlet_char_t* x)

Returns the order of $\chi_q(a, \cdot)$ which is the order of a mod q .

int **dirichlet_char_is_real**(const *dirichlet_group_t* G, const *dirichlet_char_t* chi)

Returns 1 if *chi* is a real character (iff it has order ≤ 2).

int **dirichlet_char_is_primitive**(const *dirichlet_group_t* G, const *dirichlet_char_t* chi)

Returns 1 if *chi* is primitive (iff its conductor is exactly q).

7.3.5 Character evaluation

Dirichlet characters take value in a finite cyclic group of roots of unity plus zero.

Evaluation functions return a *ulong*, this number corresponds to the power of a primitive root of unity, the special value `DIRICHLET_CHI_NULL` encoding the zero value.

ulong `dirichlet_pairing`(const *dirichlet_group_t* G, *ulong* m, *ulong* n)

ulong `dirichlet_pairing_char`(const *dirichlet_group_t* G, const *dirichlet_char_t* chi, const *dirichlet_char_t* psi)

Compute the value of the Dirichlet pairing on numbers m and n , as exponent modulo $G \rightarrow expo$.

The *char* variant takes as input two characters, so that no discrete logarithm is computed.

The returned value is the numerator of the actual value exponent mod the group exponent $G \rightarrow expo$.

ulong `dirichlet_chi`(const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *ulong* n)

Compute the value $\chi(n)$ as the exponent modulo $G \rightarrow expo$.

void `dirichlet_chi_vec`(*ulong* *v, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *ulong* nv)

Compute the list of exponent values $v[k]$ for $0 \leq k < nv$, as exponents modulo $G \rightarrow expo$.

void `dirichlet_chi_vec_order`(*ulong* *v, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *ulong* order, *ulong* nv)

Compute the list of exponent values $v[k]$ for $0 \leq k < nv$, as exponents modulo *order*, which is assumed to be a multiple of the order of *chi*.

7.3.6 Character operations

void `dirichlet_char_mul`(*dirichlet_char_t* chi2, const *dirichlet_group_t* G, const *dirichlet_char_t* chi1, const *dirichlet_char_t* chi2)

Multiply two characters of the same group G .

void `dirichlet_char_pow`(*dirichlet_char_t* c, const *dirichlet_group_t* G, const *dirichlet_char_t* a, *ulong* n)

Take the power of a character.

void `dirichlet_char_lift`(*dirichlet_char_t* chi_G, const *dirichlet_group_t* G, const *dirichlet_char_t* chi_H, const *dirichlet_group_t* H)

If H is a subgroup of G , computes the character in G corresponding to chi_H in H .

void `dirichlet_char_lower`(*dirichlet_char_t* chi_H, const *dirichlet_group_t* H, const *dirichlet_char_t* chi_G, const *dirichlet_group_t* G)

If chi_G is a character of G which factors through H , sets chi_H to the corresponding restriction in H .

This requires $c(\chi_G) \mid q_H \mid q_G$, where $c(\chi_G)$ is the conductor of χ_G and q_G, q_H are the moduli of G and H .

7.4 dlog.h – discrete logarithms mod ulong primes

This module implements discrete logarithms, with the application to Dirichlet characters in mind.

In particular, this module defines a `dlog_precomp_t` structure permitting to describe a discrete log problem in some subgroup of $(\mathbb{Z}/p^e\mathbb{Z})^\times$ for primepower moduli p^e , and store precomputed data for faster computation of several such discrete logarithms.

When initializing this data, the user provides both a group description and the expected number of subsequent discrete logarithms calls. The choice of algorithm and the amount of stored data depend both on the structure of the group and this number.

No particular effort has been made towards single discrete logarithm computation. Currently only machine size primepower moduli are supported.

7.4.1 Types, macros and constants

`DLOG_NONE`

Return value when the discrete logarithm does not exist

type `dlog_precomp_struct`

type `dlog_precomp_t`

Structure for discrete logarithm precomputed data.

A `dlog_precomp_t` is defined as an array of length one of type `dlog_precomp_struct`, permitting a `dlog_precomp_t` to be passed by reference.

7.4.2 Single evaluation

`ulong dlog_once(ulong b, ulong a, const nmod_t mod, ulong n)`

Return x such that $b = a^x$ in $(\mathbb{Z}/\text{mod}\mathbb{Z})^\times$, where a is known to have order n .

7.4.3 Precomputations

`void dlog_precomp_n_init(dlog_precomp_t pre, ulong a, ulong mod, ulong n, ulong num)`

Precompute data for num discrete logarithms evaluations in the subgroup generated by a modulo mod , where a is known to have order n .

`ulong dlog_precomp(const dlog_precomp_t pre, ulong b)`

Return $\log(b)$ for the group described in pre .

`void dlog_precomp_clear(dlog_precomp_t pre)`

Clears t .

Specialized versions of `dlog_precomp_n_init()` are available when specific information is known about the group:

`void dlog_precomp_modpe_init(dlog_precomp_t pre, ulong a, ulong p, ulong e, ulong pe, ulong num)`

Assume that a generates the group of residues modulo pe equal p^e for prime p .

`void dlog_precomp_p_init(dlog_precomp_t pre, ulong a, ulong mod, ulong p, ulong num)`

Assume that a has prime order p .

`void dlog_precomp_pe_init(dlog_precomp_t pre, ulong a, ulong mod, ulong p, ulong e, ulong pe, ulong num)`

Assume that a has primepower order pe .

void **dlog_precomp_small_init**(*dlog_precomp_t* pre, *ulong* a, *ulong* mod, *ulong* n, *ulong* num)

Make a complete lookup table of size n . If mod is small, this is done using an element-indexed array (see *dlog_table_t*), otherwise with a sorted array allowing binary search.

7.4.4 Vector evaluations

These functions compute all logarithms of successive integers $1 \dots n$.

void **dlog_vec_fill**(*ulong* *v, *ulong* nv, *ulong* x)

Sets values $v[k]$ to x for all k less than nv .

void **dlog_vec_set_not_found**(*ulong* *v, *ulong* nv, *nmod_t* mod)

Sets values $v[k]$ to *DLOG_NONE* for all k not coprime to mod .

void **dlog_vec**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

Sets $v[k]$ to $\log(k, a)$ times value va for $0 \leq k < nv$, where a has order na . va should be 1 for usual log computation.

void **dlog_vec_add**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

Same parameters as before, but adds $\log(k, a) \times v_a$ to $v[k]$ and reduce modulo $order$ instead of replacing the value. Indices k such that $v[k]$ equals *DLOG_NONE* are ignored.

Depending on the relative size of nv and na , these two *dlog_vec* functions call one of the following functions.

void **dlog_vec_loop**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

void **dlog_vec_loop_add**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

Perform a complete loop of size na on powers of a to fill the logarithm values, discarding powers outside the bounds of v . This requires no discrete logarithm computation.

void **dlog_vec_eratos**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

void **dlog_vec_eratos_add**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

Compute discrete logarithms of prime numbers less than nv and propagate to composite numbers.

void **dlog_vec_sieve_add**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

void **dlog_vec_sieve**(*ulong* *v, *ulong* nv, *ulong* a, *ulong* va, *nmod_t* mod, *ulong* na, *nmod_t* order)

Compute the discrete logarithms of the first few prime numbers, then use them as a factor base to obtain the logarithms of larger primes by sieving techniques.

In the the present implementation, the full index-calculus method is not implemented.

7.4.5 Internal discrete logarithm strategies

Several discrete logarithms strategies are implemented:

- Complete lookup table for small groups.
- Baby-step giant-step table.

combined with mathematical reductions:

- Pohlig-Hellman decomposition (Chinese remainder decomposition on the order of the group and base p decomposition for primepower order).
- p -adic log for primepower modulus p^e .

The *dlog_precomp* structure makes recursive use of the following method-specific structures.

Complete table

type `dlog_table_struct`

type `dlog_table_t`

Structure for complete lookup table.

ulong `dlog_table_init(dlog_table_t t, ulong a, ulong mod)`

Initialize a table of powers of a modulo mod , storing all elements in an array of size mod .

void `dlog_table_clear(dlog_table_t t)`

Clears t .

ulong `dlog_table(const dlog_table_t t, ulong b)`

Return $\log(b, a)$ using the precomputed data t .

Baby-step giant-step table

type `dlog_bsgs_struct`

type `dlog_bsgs_t`

Structure for Baby-Step Giant-Step decomposition.

ulong `dlog_bsgs_init(dlog_bsgs_t t, ulong a, ulong mod, ulong n, ulong m)`

Initialize t and store the first m powers of a in a sorted array. The return value is a rough measure of the cost of each logarithm using this table. The user should take $m \approx \sqrt{kn}$ to compute k logarithms in a group of size n .

void `dlog_bsgs_clear(dlog_bsgs_t t)`

Clears t .

ulong `dlog_bsgs(const dlog_bsgs_t t, ulong b)`

Return $\log(b, a)$ using the precomputed data t .

Prime-power modulus decomposition

type `dlog_modpe_struct`

type `dlog_modpe_t`

Structure for discrete logarithm modulo primepower p^e .

A `dlog_modpe_t` is defined as an array of length one of type `dlog_modpe_struct`, permitting a `dlog_modpe_t` to be passed by reference.

ulong `dlog_modpe_init(dlog_modpe_t t, ulong a, ulong p, ulong e, ulong pe, ulong num)`

void `dlog_modpe_clear(dlog_modpe_t t)`

Clears t .

ulong `dlog_modpe(const dlog_modpe_t t, ulong b)`

Return $\log(b, a)$ using the precomputed data t .

CRT decomposition

type `dlog_crt_struct`

type `dlog_crt_t`

Structure for discrete logarithm for groups of composite order. A `dlog_crt_t` is defined as an array of length one of type `dlog_crt_struct`, permitting a `dlog_crt_t` to be passed by reference.

`ulong dlog_crt_init(dlog_crt_t t, ulong a, ulong mod, ulong n, ulong num)`

Precompute data for *num* evaluations of discrete logarithms in base *a* modulo *mod*, where *a* has composite order *n*, using chinese remainder decomposition.

`void dlog_crt_clear(dlog_crt_t t)`

Clears *t*.

`ulong dlog_crt(const dlog_crt_t t, ulong b)`

Return $\log(b, a)$ using the precomputed data *t*.

padic decomposition

type `dlog_power_struct`

type `dlog_power_t`

Structure for discrete logarithm for groups of primepower order. A `dlog_power_t` is defined as an array of length one of type `dlog_power_struct`, permitting a `dlog_power_t` to be passed by reference.

`ulong dlog_power_init(dlog_power_t t, ulong a, ulong mod, ulong p, ulong e, ulong num)`

Precompute data for *num* evaluations of discrete logarithms in base *a* modulo *mod*, where *a* has prime power order *pe* equals p^e , using decomposition in base *p*.

`void dlog_power_clear(dlog_power_t t)`

Clears *t*.

`ulong dlog_power(const dlog_power_t t, ulong b)`

Return $\log(b, a)$ using the precomputed data *t*.

Pollard rho method

type `dlog_rho_struct`

type `dlog_rho_t`

Structure for discrete logarithm using Pollard rho. A `dlog_rho_t` is defined as an array of length one of type `dlog_rho_struct`, permitting a `dlog_rho_t` to be passed by reference.

`void dlog_rho_init(dlog_rho_t t, ulong a, ulong mod, ulong n)`

Initialize random walks for evaluations of discrete logarithms in base *a* modulo *mod*, where *a* has order *n*.

`void dlog_rho_clear(dlog_rho_t t)`

Clears *t*.

`ulong dlog_rho(const dlog_rho_t t, ulong b)`

Return $\log(b, a)$ by the rho method in the group described by *t*.

7.5 `bool_mat.h` – matrices over booleans

A `bool_mat_t` represents a dense matrix over the boolean semiring $\langle \{0, 1\}, \vee, \wedge \rangle$, implemented as an array of entries of type `int`.

The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

7.5.1 Types, macros and constants

type `bool_mat_struct`

type `bool_mat_t`

Contains a pointer to a flat array of the entries (`entries`), an array of pointers to the start of each row (`rows`), and the number of rows (`r`) and columns (`c`).

An `bool_mat_t` is defined as an array of length one of type `bool_mat_struct`, permitting an `bool_mat_t` to be passed by reference.

int `bool_mat_get_entry`(const `bool_mat_t` *mat*, *slong* *i*, *slong* *j*)

Returns the entry of matrix *mat* at row *i* and column *j*.

void `bool_mat_set_entry`(`bool_mat_t` *mat*, *slong* *i*, *slong* *j*, int *x*)

Sets the entry of matrix *mat* at row *i* and column *j* to *x*.

`bool_mat_nrows`(*mat*)

Returns the number of rows of the matrix.

`bool_mat_ncols`(*mat*)

Returns the number of columns of the matrix.

7.5.2 Memory management

void `bool_mat_init`(`bool_mat_t` *mat*, *slong* *r*, *slong* *c*)

Initializes the matrix, setting it to the zero matrix with *r* rows and *c* columns.

void `bool_mat_clear`(`bool_mat_t` *mat*)

Clears the matrix, deallocating all entries.

int `bool_mat_is_empty`(const `bool_mat_t` *mat*)

Returns nonzero iff the number of rows or the number of columns in *mat* is zero. Note that this does not depend on the entry values of *mat*.

int `bool_mat_is_square`(const `bool_mat_t` *mat*)

Returns nonzero iff the number of rows is equal to the number of columns in *mat*.

7.5.3 Conversions

void `bool_mat_set`(`bool_mat_t` *dest*, const `bool_mat_t` *src*)

Sets *dest* to *src*. The operands must have identical dimensions.

7.5.4 Input and output

void **bool_mat_print**(const *bool_mat_t* mat)

Prints each entry in the matrix.

void **bool_mat_fprint**(FILE *file, const *bool_mat_t* mat)

Prints each entry in the matrix to the stream *file*.

7.5.5 Value comparisons

int **bool_mat_equal**(const *bool_mat_t* mat1, const *bool_mat_t* mat2)

Returns nonzero iff the matrices have the same dimensions and identical entries.

int **bool_mat_any**(const *bool_mat_t* mat)

Returns nonzero iff *mat* has a nonzero entry.

int **bool_mat_all**(const *bool_mat_t* mat)

Returns nonzero iff all entries of *mat* are nonzero.

int **bool_mat_is_diagonal**(const *bool_mat_t* A)

Returns nonzero iff $i \neq j \implies \bar{A}_{ij}$.

int **bool_mat_is_lower_triangular**(const *bool_mat_t* A)

Returns nonzero iff $i < j \implies \bar{A}_{ij}$.

int **bool_mat_is_transitive**(const *bool_mat_t* mat)

Returns nonzero iff $A_{ij} \wedge A_{jk} \implies A_{ik}$.

int **bool_mat_is_nilpotent**(const *bool_mat_t* A)

Returns nonzero iff some positive matrix power of *A* is zero.

7.5.6 Random generation

void **bool_mat_randtest**(*bool_mat_t* mat, *flint_rand_t* state)

Sets *mat* to a random matrix.

void **bool_mat_randtest_diagonal**(*bool_mat_t* mat, *flint_rand_t* state)

Sets *mat* to a random diagonal matrix.

void **bool_mat_randtest_nilpotent**(*bool_mat_t* mat, *flint_rand_t* state)

Sets *mat* to a random nilpotent matrix.

7.5.7 Special matrices

void **bool_mat_zero**(*bool_mat_t* mat)

Sets all entries in *mat* to zero.

void **bool_mat_one**(*bool_mat_t* mat)

Sets the entries on the main diagonal to ones, and all other entries to zero.

void **bool_mat_directed_path**(*bool_mat_t* A)

Sets A_{ij} to $j = i + 1$. Requires that *A* is a square matrix.

void **bool_mat_directed_cycle**(*bool_mat_t* A)

Sets A_{ij} to $j = (i + 1) \bmod n$ where *n* is the order of the square matrix *A*.

7.5.8 Transpose

void **bool_mat_transpose**(*bool_mat_t* dest, const *bool_mat_t* src)

Sets *dest* to the transpose of *src*. The operands must have compatible dimensions. Aliasing is allowed for square matrices.

7.5.9 Arithmetic

void **bool_mat_complement**(*bool_mat_t* B, const *bool_mat_t* A)

Sets *B* to the logical complement of *A*. That is B_{ij} is set to $\bar{A}_{ij}$. The operands must have the same dimensions.

void **bool_mat_add**(*bool_mat_t* res, const *bool_mat_t* mat1, const *bool_mat_t* mat2)

Sets *res* to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **bool_mat_mul**(*bool_mat_t* res, const *bool_mat_t* mat1, const *bool_mat_t* mat2)

Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

void **bool_mat_mul_entrywise**(*bool_mat_t* res, const *bool_mat_t* mat1, const *bool_mat_t* mat2)

Sets *res* to the entrywise product of *mat1* and *mat2*. The operands must have the same dimensions.

void **bool_mat_sqr**(*bool_mat_t* B, const *bool_mat_t* A)

Sets *B* to the matrix square of *A*. The operands must both be square with the same dimensions.

void **bool_mat_pow_ui**(*bool_mat_t* B, const *bool_mat_t* A, *ulong* exp)

Sets *B* to *A* raised to the power *exp*. Requires that *A* is a square matrix.

7.5.10 Special functions

int **bool_mat_trace**(const *bool_mat_t* mat)

Returns the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square. The sum is in the boolean semiring, so this function returns nonzero iff any entry on the diagonal of *mat* is nonzero.

slong **bool_mat_nilpotency_degree**(const *bool_mat_t* A)

Returns the nilpotency degree of the $n \times n$ matrix *A*. It returns the smallest positive *k* such that $A^k = 0$. If no such *k* exists then the function returns -1 if *n* is positive, and otherwise it returns 0.

void **bool_mat_transitive_closure**(*bool_mat_t* B, const *bool_mat_t* A)

Sets *B* to the transitive closure $\sum_{k=1}^{\infty} A^k$. The matrix *A* is required to be square.

slong **bool_mat_get_strongly_connected_components**(*slong* *p, const *bool_mat_t* A)

Partitions the *n* row and column indices of the $n \times n$ matrix *A* according to the strongly connected components (SCC) of the graph for which *A* is the adjacency matrix. If the graph has *k* SCCs then the function returns *k*, and for each vertex $i \in [0, n - 1]$, p_i is set to the index of the SCC to which the vertex belongs. The SCCs themselves can be considered as nodes in a directed acyclic graph (DAG), and the SCCs are indexed in postorder with respect to that DAG.

slong **bool_mat_all_pairs_longest_walk**(*fmpz_mat_t* B, const *bool_mat_t* A)

Sets B_{ij} to the length of the longest walk with endpoint vertices *i* and *j* in the graph whose adjacency matrix is *A*. The matrix *A* must be square. Empty walks with zero length which begin and end at the same vertex are allowed. If *j* is not reachable from *i* then no walk from *i* to *j* exists and B_{ij} is set to the special value -1 . If arbitrarily long walks from *i* to *j* exist then B_{ij} is set to the special value -2 .

The function returns -2 if any entry of B_{ij} is -2 , and otherwise it returns the maximum entry in B , except if A is empty in which case -1 is returned. Note that the returned value is one less than that of `nilpotency_degree()`.

This function can help quantify entrywise errors in a truncated evaluation of a matrix power series. If A is an indicator matrix with the same sparsity pattern as a matrix M over the real or complex numbers, and if B_{ij} does not take the special value -2 , then the tail $[\sum_{k=N}^{\infty} a_k M^k]_{ij}$ vanishes when $N > B_{ij}$.

NUMBER FIELDS AND ALGEBRAIC NUMBERS

8.1 `nf.h` – number fields

type `nf_struct`

type `nf_t`

Represents a number field.

void `nf_init`(*nf_t* nf, const *fmpz_poly_t* pol)

Perform basic initialisation of a number field (for element arithmetic) given a defining polynomial over $\mathbb{Q}$.

void `nf_clear`(*nf_t* nf)

Release resources used by a number field object. The object will need initialisation again before it can be used.

8.2 `nf_elem.h` – number field elements

Authors:

- William Hart

8.2.1 Initialisation

type `nf_elem_struct`

type `nf_elem_t`

Represents a number field element.

void `nf_elem_init`(*nf_elem_t* a, const *nf_t* nf)

Initialise a number field element to belong to the given number field `nf`. The element is set to zero.

void `nf_elem_clear`(*nf_elem_t* a, const *nf_t* nf)

Clear resources allocated by the given number field element in the given number field.

void `nf_elem_randtest`(*nf_elem_t* a, *flint_rand_t* state, *flint_bitcnt_t* bits, const *nf_t* nf)

Generate a random number field element a in the number field `nf` whose coefficients have up to the given number of bits.

void `nf_elem_canonicalise`(*nf_elem_t* a, const *nf_t* nf)

Canonicalise a number field element, i.e. reduce numerator and denominator to lowest terms. If the numerator is 0, set the denominator to 1.

void **`_nf_elem_reduce`**(*nf_elem_t* a, const *nf_t* nf)

Reduce a number field element modulo the defining polynomial. This is used with functions such as `nf_elem_mul_red` which allow reduction to be delayed. Does not canonicalise.

void **`nf_elem_reduce`**(*nf_elem_t* a, const *nf_t* nf)

Reduce a number field element modulo the defining polynomial. This is used with functions such as `nf_elem_mul_red` which allow reduction to be delayed.

int **`_nf_elem_invertible_check`**(*nf_elem_t* a, const *nf_t* nf)

Whilst the defining polynomial for a number field should by definition be irreducible, it is not enforced. Thus in test code, it is convenient to be able to check that a given number field element is invertible modulo the defining polynomial of the number field. This function does precisely this.

If *a* is invertible modulo the defining polynomial of *nf* the value 1 is returned, otherwise 0 is returned.

The function is only intended to be used in test code.

8.2.2 Conversion

void **`nf_elem_set_fmpz_mat_row`**(*nf_elem_t* b, const *fmpz_mat_t* M, const *slong* i, *fmpz_t* den, const *nf_t* nf)

Set *b* to the element specified by row *i* of the matrix *M* and with the given denominator *d*. Column 0 of the matrix corresponds to the constant coefficient of the number field element.

void **`nf_elem_get_fmpz_mat_row`**(*fmpz_mat_t* M, const *slong* i, *fmpz_t* den, const *nf_elem_t* b, const *nf_t* nf)

Set the row *i* of the matrix *M* to the coefficients of the numerator of the element *b* and *d* to the denominator of *b*. Column 0 of the matrix corresponds to the constant coefficient of the number field element.

void **`nf_elem_set_fmpz_poly`**(*nf_elem_t* a, const *fmpz_poly_t* pol, const *nf_t* nf)

Set *a* to the element corresponding to the polynomial *pol*.

void **`nf_elem_get_fmpz_poly`**(*fmpz_poly_t* pol, const *nf_elem_t* a, const *nf_t* nf)

Set *pol* to a polynomial corresponding to *a*, reduced modulo the defining polynomial of *nf*.

void **`nf_elem_get_nmod_poly_den`**(*nmod_poly_t* pol, const *nf_elem_t* a, const *nf_t* nf, int den)

Set *pol* to the reduction of the polynomial corresponding to the numerator of *a*. If *den* == 1, the result is multiplied by the inverse of the denominator of *a*. In this case it is assumed that the reduction of the denominator of *a* is invertible.

void **`nf_elem_get_nmod_poly`**(*nmod_poly_t* pol, const *nf_elem_t* a, const *nf_t* nf)

Set *pol* to the reduction of the polynomial corresponding to the numerator of *a*. The result is multiplied by the inverse of the denominator of *a*. It is assumed that the reduction of the denominator of *a* is invertible.

void **`nf_elem_get_fmpz_mod_poly_den`**(*fmpz_mod_poly_t* pol, const *nf_elem_t* a, const *nf_t* nf, int den, const *fmpz_mod_ctx_t* ctx)

Set *pol* to the reduction of the polynomial corresponding to the numerator of *a*. If *den* == 1, the result is multiplied by the inverse of the denominator of *a*. In this case it is assumed that the reduction of the denominator of *a* is invertible.

void **`nf_elem_get_fmpz_mod_poly`**(*fmpz_mod_poly_t* pol, const *nf_elem_t* a, const *nf_t* nf, const *fmpz_mod_ctx_t* ctx)

Set *pol* to the reduction of the polynomial corresponding to the numerator of *a*. The result is multiplied by the inverse of the denominator of *a*. It is assumed that the reduction of the denominator of *a* is invertible.

8.2.3 Basic manipulation

void **nf_elem_set_den**(*nf_elem_t* b, *fmpz_t* d, const *nf_t* nf)
 Set the denominator of the *nf_elem_t* b to the given integer *d*. Assumes $d > 0$.

void **nf_elem_get_den**(*fmpz_t* d, const *nf_elem_t* b, const *nf_t* nf)
 Set *d* to the denominator of the *nf_elem_t* b.

void **_nf_elem_set_coeff_num_fmpz**(*nf_elem_t* a, *slong* i, const *fmpz_t* d, const *nf_t* nf)
 Set the *i*-th coefficient of the denominator of *a* to the given integer *d*.

8.2.4 Comparison

int **_nf_elem_equal**(const *nf_elem_t* a, const *nf_elem_t* b, const *nf_t* nf)
 Return 1 if the given number field elements are equal in the given number field *nf*. This function does emph{not} assume *a* and *b* are canonicalised.

int **nf_elem_equal**(const *nf_elem_t* a, const *nf_elem_t* b, const *nf_t* nf)
 Return 1 if the given number field elements are equal in the given number field *nf*. This function assumes *a* and *b* emph{are} canonicalised.

int **nf_elem_is_zero**(const *nf_elem_t* a, const *nf_t* nf)
 Return 1 if the given number field element is equal to zero, otherwise return 0.

int **nf_elem_is_one**(const *nf_elem_t* a, const *nf_t* nf)
 Return 1 if the given number field element is equal to one, otherwise return 0.

8.2.5 I/O

void **nf_elem_print_pretty**(const *nf_elem_t* a, const *nf_t* nf, const char *var)
 Print the given number field element to `stdout` using the null-terminated string *var* not equal to `"\0"` as the name of the primitive element.

8.2.6 Arithmetic

void **nf_elem_zero**(*nf_elem_t* a, const *nf_t* nf)
 Set the given number field element to zero.

void **nf_elem_one**(*nf_elem_t* a, const *nf_t* nf)
 Set the given number field element to one.

void **nf_elem_set**(*nf_elem_t* a, const *nf_elem_t* b, const *nf_t* nf)
 Set the number field element *a* to equal the number field element *b*, i.e. set $a = b$.

void **nf_elem_neg**(*nf_elem_t* a, const *nf_elem_t* b, const *nf_t* nf)
 Set the number field element *a* to minus the number field element *b*, i.e. set $a = -b$.

void **nf_elem_swap**(*nf_elem_t* a, *nf_elem_t* b, const *nf_t* nf)
 Efficiently swap the two number field elements *a* and *b*.

void **nf_elem_mul_gen**(*nf_elem_t* a, const *nf_elem_t* b, const *nf_t* nf)
 Multiply the element *b* with the generator of the number field.

void **_nf_elem_add**(*nf_elem_t* r, const *nf_elem_t* a, const *nf_elem_t* b, const *nf_t* nf)
 Add two elements of a number field *nf*, i.e. set $r = a + b$. Canonicalisation is not performed.

`void nf_elem_add(nf_elem_t r, const nf_elem_t a, const nf_elem_t b, const nf_t nf)`
Add two elements of a number field `nf`, i.e. set $r = a + b$.

`void _nf_elem_sub(nf_elem_t r, const nf_elem_t a, const nf_elem_t b, const nf_t nf)`
Subtract two elements of a number field `nf`, i.e. set $r = a - b$. Canonicalisation is not performed.

`void nf_elem_sub(nf_elem_t r, const nf_elem_t a, const nf_elem_t b, const nf_t nf)`
Subtract two elements of a number field `nf`, i.e. set $r = a - b$.

`void _nf_elem_mul(nf_elem_t a, const nf_elem_t b, const nf_elem_t c, const nf_t nf)`
Multiply two elements of a number field `nf`, i.e. set $r = a * b$. Does not canonicalise. Aliasing of inputs with output is not supported.

`void _nf_elem_mul_red(nf_elem_t a, const nf_elem_t b, const nf_elem_t c, const nf_t nf, int red)`
As per `_nf_elem_mul`, but reduction modulo the defining polynomial of the number field is only carried out if `red == 1`. Assumes both inputs are reduced.

`void nf_elem_mul(nf_elem_t a, const nf_elem_t b, const nf_elem_t c, const nf_t nf)`
Multiply two elements of a number field `nf`, i.e. set $r = a * b$.

`void nf_elem_mul_red(nf_elem_t a, const nf_elem_t b, const nf_elem_t c, const nf_t nf, int red)`
As per `nf_elem_mul`, but reduction modulo the defining polynomial of the number field is only carried out if `red == 1`. Assumes both inputs are reduced.

`void _nf_elem_inv(nf_elem_t r, const nf_elem_t a, const nf_t nf)`
Invert an element of a number field `nf`, i.e. set $r = a^{-1}$. Aliasing of the input with the output is not supported.

`void nf_elem_inv(nf_elem_t r, const nf_elem_t a, const nf_t nf)`
Invert an element of a number field `nf`, i.e. set $r = a^{-1}$.

`void _nf_elem_div(nf_elem_t a, const nf_elem_t b, const nf_elem_t c, const nf_t nf)`
Set a to b/c in the given number field. Aliasing of a and b is not permitted.

`void nf_elem_div(nf_elem_t a, const nf_elem_t b, const nf_elem_t c, const nf_t nf)`
Set a to b/c in the given number field.

`void _nf_elem_pow(nf_elem_t res, const nf_elem_t a, ulong e, const nf_t nf)`
Set `res` to a^e using left-to-right binary exponentiation as described on p. 461 of [Knu1997].
Assumes that $a \neq 0$ and $e > 1$. Does not support aliasing.

`void nf_elem_pow(nf_elem_t res, const nf_elem_t a, ulong e, const nf_t nf)`
Set `res` to a^e using the binary exponentiation algorithm. If e is zero, returns one, so that in particular $0^0 = 1$.

`void _nf_elem_norm(fmpz_t rnum, fmpz_t rden, const nf_elem_t a, const nf_t nf)`
Set `rnum`, `rden` to the absolute norm of the given number field element a .

`void nf_elem_norm(fmpz_t res, const nf_elem_t a, const nf_t nf)`
Set `res` to the absolute norm of the given number field element a .

`void nf_elem_norm_div(fmpz_t res, const nf_elem_t a, const nf_t nf, const fmpz_t div, slong nbits)`
Set `res` to the absolute norm of the given number field element a , divided by `div`. Assumes the result to be an integer and having at most `nbits` bits.

`void _nf_elem_norm_div(fmpz_t rnum, fmpz_t rden, const nf_elem_t a, const nf_t nf, const fmpz_t divisor, slong nbits)`
Set `rnum`, `rden` to the absolute norm of the given number field element a , divided by `div`. Assumes the result to be an integer and having at most `nbits` bits.

`void _nf_elem_trace(fmpz_t rnum, fmpz_t rden, const nf_elem_t a, const nf_t nf)`
Set `rnum`, `rden` to the absolute trace of the given number field element a .

void **nf_elem_trace**(*fmpr_t* res, const *nf_elem_t* a, const *nf_t* nf)

Set *res* to the absolute trace of the given number field element *a*.

8.2.7 Representation matrix

void **nf_elem_rep_mat**(*fmpr_mat_t* res, const *nf_elem_t* a, const *nf_t* nf)

Set *res* to the matrix representing the multiplication with *a* with respect to the basis $1, a, \dots, a^{d-1}$, where *a* is the generator of the number field of *d* is its degree.

void **nf_elem_rep_mat_fmpz_mat_den**(*fmpz_mat_t* res, *fmpz_t* den, const *nf_elem_t* a, const *nf_t* nf)

Return a tuple *M, d* such that *M/d* is the matrix representing the multiplication with *a* with respect to the basis $1, a, \dots, a^{d-1}$, where *a* is the generator of the number field of *d* is its degree. The integral matrix *M* is primitive.

8.2.8 Modular reduction

void **nf_elem_mod_fmpz_den**(*nf_elem_t* z, const *nf_elem_t* a, const *fmpz_t* mod, const *nf_t* nf, int den)

If *den* == 0, return an element *z* with denominator 1, such that the coefficients of $z - da$ are divisible by *mod*, where *d* is the denominator of *a*. The coefficients of *z* are reduced modulo *mod*.

If *den* == 1, return an element *z*, such that $z - a$ has denominator 1 and the coefficients of $z - a$ are divisible by *mod*. The coefficients of *z* are reduced modulo $\text{mod} \cdot d$, where *d* is the denominator of *a*.

Reduction takes place with respect to the positive residue system.

void **nf_elem_smod_fmpz_den**(*nf_elem_t* z, const *nf_elem_t* a, const *fmpz_t* mod, const *nf_t* nf, int den)

If *den* == 0, return an element *z* with denominator 1, such that the coefficients of $z - da$ are divisible by *mod*, where *d* is the denominator of *a*. The coefficients of *z* are reduced modulo *mod*.

If *den* == 1, return an element *z*, such that $z - a$ has denominator 1 and the coefficients of $z - a$ are divisible by *mod*. The coefficients of *z* are reduced modulo $\text{mod} \cdot d$, where *d* is the denominator of *a*.

Reduction takes place with respect to the symmetric residue system.

void **nf_elem_mod_fmpz**(*nf_elem_t* res, const *nf_elem_t* a, const *fmpz_t* mod, const *nf_t* nf)

Return an element *z* such that $z - a$ has denominator 1 and the coefficients of $z - a$ are divisible by *mod*. The coefficients of *z* are reduced modulo $\text{mod} \cdot d$, where *d* is the denominator of *a*.

Reduction takes place with respect to the positive residue system.

void **nf_elem_smod_fmpz**(*nf_elem_t* res, const *nf_elem_t* a, const *fmpz_t* mod, const *nf_t* nf)

Return an element *z* such that $z - a$ has denominator 1 and the coefficients of $z - a$ are divisible by *mod*. The coefficients of *z* are reduced modulo $\text{mod} \cdot d$, where *d* is the denominator of *a*.

Reduction takes place with respect to the symmetric residue system.

void **nf_elem_coprime_den**(*nf_elem_t* res, const *nf_elem_t* a, const *fmpz_t* mod, const *nf_t* nf)

Return an element *z* such that the denominator of $z - a$ is coprime to *mod*.

Reduction takes place with respect to the positive residue system.

void **nf_elem_coprime_den_signed**(*nf_elem_t* res, const *nf_elem_t* a, const *fmpz_t* mod, const *nf_t* nf)

Return an element *z* such that the denominator of $z - a$ is coprime to *mod*.

Reduction takes place with respect to the symmetric residue system.

8.3 fmpz_i.h – Gaussian integers

This module allows working with elements of the ring $\mathbb{Z}[i]$. At present, only a minimal interface is provided.

8.3.1 Types, macros and constants

type `fmpz_i_struct`

type `fmpz_i_t`

Contains a pairs of integers representing the real and imaginary parts. An `fmpz_i_t` is defined as an array of length one of type `fmpz_i_struct`, permitting an `fmpz_i_t` to be passed by reference.

`fmpz_i_realref(x)`

Macro giving a pointer to the real part of x .

`fmpz_i_imagref(x)`

Macro giving a pointer to the imaginary part of x .

8.3.2 Basic manipulation

void `fmpz_i_init(fmpz_i_t x)`

void `fmpz_i_clear(fmpz_i_t x)`

void `fmpz_i_swap(fmpz_i_t x, fmpz_i_t y)`

void `fmpz_i_zero(fmpz_i_t x)`

void `fmpz_i_one(fmpz_i_t x)`

void `fmpz_i_set(fmpz_i_t res, const fmpz_i_t x)`

void `fmpz_i_set_si_si(fmpz_i_t res, slong a, slong b)`

8.3.3 Input and output

void `fmpz_i_print(const fmpz_i_t x)`

8.3.4 Random number generation

void `fmpz_i_randtest(fmpz_i_t res, flint_rand_t state, flint_bitcnt_t bits)`

8.3.5 Properties

int `fmpz_i_equal(const fmpz_i_t x, const fmpz_i_t y)`

int `fmpz_i_is_zero(const fmpz_i_t x)`

int `fmpz_i_is_one(const fmpz_i_t x)`

8.3.6 Units

```
int fmpz_i_is_unit(const fmpz_t x)

slong fmpz_i_canonical_unit_i_pow(const fmpz_t x)

void fmpz_i_canonicalise_unit(fmpz_t res, const fmpz_t x)
```

8.3.7 Norms

```
slong fmpz_bits(const fmpz_t x)

void fmpz_norm(fmpz_t res, const fmpz_t x)
```

8.3.8 Arithmetic

```
void fmpz_conj(fmpz_t res, const fmpz_t x)

void fmpz_neg(fmpz_t res, const fmpz_t x)

void fmpz_add(fmpz_t res, const fmpz_t x, const fmpz_t y)

void fmpz_sub(fmpz_t res, const fmpz_t x, const fmpz_t y)

void fmpz_sqr(fmpz_t res, const fmpz_t x)

void fmpz_mul(fmpz_t res, const fmpz_t x, const fmpz_t y)

void fmpz_pow_ui(fmpz_t res, const fmpz_t x, ulong exp)
```

8.3.9 Division

```
void fmpz_divexact(fmpz_t q, const fmpz_t x, const fmpz_t y)
    Sets  $q$  to the quotient of  $x$  and  $y$ , assuming that the division is exact.

void fmpz_divrem(fmpz_t q, fmpz_t r, const fmpz_t x, const fmpz_t y)
    Computes a quotient and remainder satisfying  $x = qy + r$  with  $N(r) \leq N(y)/2$ , with a canonical
    choice of remainder when breaking ties.

void fmpz_divrem_approx(fmpz_t q, fmpz_t r, const fmpz_t x, const fmpz_t y)
    Computes a quotient and remainder satisfying  $x = qy + r$  with  $N(r) < N(y)$ , with an
    implementation-defined, non-canonical choice of remainder.

slong fmpz_remove_one_plus_i(fmpz_t res, const fmpz_t x)
    Divide  $x$  exactly by the largest possible power  $(1 + i)^k$  and return the exponent  $k$ .
```

8.3.10 GCD

```
void fmpz_gcd_euclidean(fmpz_t res, const fmpz_t x, const fmpz_t y)

void fmpz_gcd_euclidean_improved(fmpz_t res, const fmpz_t x, const fmpz_t y)

void fmpz_gcd_binary(fmpz_t res, const fmpz_t x, const fmpz_t y)

void fmpz_gcd_shortest(fmpz_t res, const fmpz_t x, const fmpz_t y)
```


void **fmpz_gcd**(*fmpz_t* res, const *fmpz_t* x, const *fmpz_t* y)

Computes the GCD of x and y . The result is in canonical unit form.

The *euclidean* version is a straightforward implementation of Euclid's algorithm. The *euclidean_improved* version is optimized by performing approximate divisions. The *binary* version uses a $(1+i)$ -ary analog of the binary GCD algorithm for integers [Wei2000]. The *shortest* version finds the GCD as the shortest vector in a lattice. The default version chooses an algorithm automatically.

8.3.11 Primality testing

int **fmpz_is_prime**(const *fmpz_t* n)

Check whether n is a Gaussian prime.

int **fmpz_is_probabprime**(const *fmpz_t* n)

Check whether n is a probable Gaussian prime.

8.4 qqbar.h – algebraic numbers represented by minimal polynomials

A *qqbar_t* represents a real or complex algebraic number (an element of $\overline{\mathbb{Q}}$) by its unique reduced minimal polynomial in $\mathbb{Z}[x]$ and an isolating complex interval. The precision of isolating intervals is maintained automatically to ensure that all operations on *qqbar_t* instances are exact.

This representation is useful for working with individual algebraic numbers of moderate degree (up to 100, say). Arithmetic in this representation is expensive: an arithmetic operation on numbers of degrees m and n involves computing and then factoring an annihilating polynomial of degree mn and potentially also performing numerical root-finding. For doing repeated arithmetic, it is generally more efficient to work with the *ca_t* type in a fixed number field. The *qqbar_t* type is used internally by the *ca_t* type to represent the embedding of number fields in $\mathbb{R}$ or $\mathbb{C}$ and to decide predicates for algebraic numbers.

8.4.1 Types and macros

type **qqbar_struct**

type **qqbar_t**

A *qqbar_struct* consists of an *fmpz_poly_struct* and an *acb_struct*. A *qqbar_t* is defined as an array of length one of type *qqbar_struct*, permitting a *qqbar_t* to be passed by reference.

type **qqbar_ptr**

Alias for *qqbar_struct **, used for *qqbar* vectors.

type **qqbar_srcptr**

Alias for *const qqbar_struct **, used for *qqbar* vectors when passed as readonly input to functions.

QQBAR_POLY(x)

Macro returning a pointer to the minimal polynomial of x which can be used as an *fmpz_poly_t*.

QQBAR_COEFFS(x)

Macro returning a pointer to the array of *fmpz* coefficients of the minimal polynomial of x .

QQBAR_ENCLOSURE(x)

Macro returning a pointer to the enclosure of x which can be used as an *acb_t*.

8.4.2 Memory management

void `qqbar_init`(*qqbar_t* res)

Initializes the variable *res* for use, and sets its value to zero.

void `qqbar_clear`(*qqbar_t* res)

Clears the variable *res*, freeing or recycling its allocated memory.

qqbar_ptr `qqbar_vec_init`(*slong* len)

Returns a pointer to an array of *len* initialized *qqbar_struct*s.

void `_qqbar_vec_clear`(*qqbar_ptr* vec, *slong* len)

Clears all *len* entries in the vector *vec* and frees the vector itself.

8.4.3 Assignment

void `qqbar_swap`(*qqbar_t* x, *qqbar_t* y)

Swaps the values of *x* and *y* efficiently.

void `qqbar_set`(*qqbar_t* res, const *qqbar_t* x)

void `qqbar_set_si`(*qqbar_t* res, *slong* x)

void `qqbar_set_ui`(*qqbar_t* res, *ulong* x)

void `qqbar_set_fmpz`(*qqbar_t* res, const *fmpz_t* x)

void `qqbar_set_fmpq`(*qqbar_t* res, const *fmpq_t* x)

Sets *res* to the value *x*.

void `qqbar_set_re_im`(*qqbar_t* res, const *qqbar_t* x, const *qqbar_t* y)

Sets *res* to the value $x + yi$.

int `qqbar_set_d`(*qqbar_t* res, double x)

int `qqbar_set_re_im_d`(*qqbar_t* res, double x, double y)

Sets *res* to the value *x* or $x + yi$ respectively. These functions performs error handling: if *x* and *y* are finite, the conversion succeeds and the return flag is 1. If *x* or *y* is non-finite (infinity or NaN), the conversion fails and the return flag is 0.

8.4.4 Properties

slong `qqbar_degree`(const *qqbar_t* x)

Returns the degree of *x*, i.e. the degree of the minimal polynomial.

int `qqbar_is_rational`(const *qqbar_t* x)

Returns whether *x* is a rational number.

int `qqbar_is_integer`(const *qqbar_t* x)

Returns whether *x* is an integer (an element of $\mathbb{Z}$).

int `qqbar_is_algebraic_integer`(const *qqbar_t* x)

Returns whether *x* is an algebraic integer, i.e. whether its minimal polynomial has leading coefficient 1.

int `qqbar_is_zero`(const *qqbar_t* x)

int `qqbar_is_one`(const *qqbar_t* x)

int `qqbar_is_neg_one`(const *qqbar_t* x)

Returns whether *x* is the number 0, 1, -1 .

int `qqbar_is_i`(const *qqbar_t* x)

int **qqbar_is_neg_i**(const *qqbar_t* x)

Returns whether x is the imaginary unit i (respectively $-i$).

int **qqbar_is_real**(const *qqbar_t* x)

Returns whether x is a real number.

void **qqbar_height**(*fmpz_t* res, const *qqbar_t* x)

Sets res to the height of x (the largest absolute value of the coefficients of the minimal polynomial of x).

slong **qqbar_height_bits**(const *qqbar_t* x)

Returns the height of x (the largest absolute value of the coefficients of the minimal polynomial of x) measured in bits.

int **qqbar_within_limits**(const *qqbar_t* x, *slong* deg_limit, *slong* bits_limit)

Checks if x has degree bounded by deg_limit and height bounded by $bits\_limit$ bits, returning 0 (false) or 1 (true). If deg_limit is set to 0, the degree check is skipped, and similarly for $bits\_limit$.

int **qqbar_binop_within_limits**(const *qqbar_t* x, const *qqbar_t* y, *slong* deg_limit, *slong* bits_limit)

Checks if $x + y$, $x - y$, $x \cdot y$ and x/y certainly have degree bounded by deg_limit (by multiplying the degrees for x and y to obtain a trivial bound). For $bits\_limits$, the sum of the bit heights of x and y is checked against the bound (this is only a heuristic). If deg_limit is set to 0, the degree check is skipped, and similarly for $bits\_limit$.

8.4.5 Conversions

void **_qqbar_get_fmpq**(*fmpz_t* num, *fmpz_t* den, const *qqbar_t* x)

Sets num and den to the numerator and denominator of x . Aborts if x is not a rational number.

void **qqbar_get_fmpq**(*fmpq_t* res, const *qqbar_t* x)

Sets res to x . Aborts if x is not a rational number.

void **qqbar_get_fmpz**(*fmpz_t* res, const *qqbar_t* x)

Sets res to x . Aborts if x is not an integer.

8.4.6 Special values

void **qqbar_zero**(*qqbar_t* res)

Sets res to the number 0.

void **qqbar_one**(*qqbar_t* res)

Sets res to the number 1.

void **qqbar_i**(*qqbar_t* res)

Sets res to the imaginary unit i .

void **qqbar_phi**(*qqbar_t* res)

Sets res to the golden ratio $\varphi = \frac{1}{2}(\sqrt{5} + 1)$.

8.4.7 Input and output

void **qqbar_print**(const *qqbar_t* x)

Prints *res* to standard output. The output shows the degree and the list of coefficients of the minimal polynomial followed by a decimal representation of the enclosing interval. This function is mainly intended for debugging.

void **qqbar_printn**(const *qqbar_t* x, *slong* n)

Prints *res* to standard output. The output shows a decimal approximation to *n* digits.

void **qqbar_printnd**(const *qqbar_t* x, *slong* n)

Prints *res* to standard output. The output shows a decimal approximation to *n* digits, followed by the degree of the number.

For example, *print*, *printn* and *printnd* with *n* = 6 give the following output for the numbers 0, 1, *i*, φ , $\sqrt{2} - \sqrt{3}i$:

```
deg 1 [0, 1] 0
deg 1 [-1, 1] 1.00000
deg 2 [1, 0, 1] 1.00000*I
deg 2 [-1, -1, 1] [1.61803398874989484820458683436563811772 +/- 6.00e-39]
deg 4 [25, 0, 2, 0, 1] [1.4142135623730950488016887242096980786 +/- 8.67e-38] + [-1.
↪ 732050807568877293527446341505872367 +/- 1.10e-37]*I

0
1.00000
1.00000*I
1.61803
1.41421 - 1.73205*I

0 (deg 1)
1.00000 (deg 1)
1.00000*I (deg 2)
1.61803 (deg 2)
1.41421 - 1.73205*I (deg 4)
```

8.4.8 Random generation

void **qqbar_randtest**(*qqbar_t* res, *flint_rand_t* state, *slong* deg, *slong* bits)

Sets *res* to a random algebraic number with degree up to *deg* and with height (measured in bits) up to *bits*.

void **qqbar_randtest_real**(*qqbar_t* res, *flint_rand_t* state, *slong* deg, *slong* bits)

Sets *res* to a random real algebraic number with degree up to *deg* and with height (measured in bits) up to *bits*.

void **qqbar_randtest_nonreal**(*qqbar_t* res, *flint_rand_t* state, *slong* deg, *slong* bits)

Sets *res* to a random nonreal algebraic number with degree up to *deg* and with height (measured in bits) up to *bits*. Since all algebraic numbers of degree 1 are real, *deg* must be at least 2.

8.4.9 Comparisons

int `qqbar_equal`(const `qqbar_t` x, const `qqbar_t` y)

Returns whether x and y are equal.

int `qqbar_equal_fmpq_poly_val`(const `qqbar_t` x, const `fmpq_poly_t` f, const `qqbar_t` y)

Returns whether x is equal to $f(y)$. This function is more efficient than evaluating $f(y)$ and comparing the results.

int `qqbar_cmp_re`(const `qqbar_t` x, const `qqbar_t` y)

Compares the real parts of x and y , returning -1, 0 or +1.

int `qqbar_cmp_im`(const `qqbar_t` x, const `qqbar_t` y)

Compares the imaginary parts of x and y , returning -1, 0 or +1.

int `qqbar_cmpabs_re`(const `qqbar_t` x, const `qqbar_t` y)

Compares the absolute values of the real parts of x and y , returning -1, 0 or +1.

int `qqbar_cmpabs_im`(const `qqbar_t` x, const `qqbar_t` y)

Compares the absolute values of the imaginary parts of x and y , returning -1, 0 or +1.

int `qqbar_cmpabs`(const `qqbar_t` x, const `qqbar_t` y)

Compares the absolute values of x and y , returning -1, 0 or +1.

int `qqbar_cmp_root_order`(const `qqbar_t` x, const `qqbar_t` y)

Compares x and y using an arbitrary but convenient ordering defined on the complex numbers. This is useful for sorting the roots of a polynomial in a canonical order.

We define the root order as follows: real roots come first, in descending order. Nonreal roots are subsequently ordered first by real part in descending order, then in ascending order by the absolute value of the imaginary part, and then in descending order of the sign. This implies that complex conjugate roots are adjacent, with the root in the upper half plane first.

ulong `qqbar_hash`(const `qqbar_t` x)

Returns a hash of x . As currently implemented, this function only hashes the minimal polynomial of x . The user should mix in some bits based on the numerical value if it is critical to distinguish between conjugates of the same minimal polynomial. This function is also likely to produce serial runs of values for lexicographically close minimal polynomials. This is not necessarily a problem for use in hash tables, but if it is important that all bits in the output are random, the user should apply an integer hash function to the output.

8.4.10 Complex parts

void `qqbar_conj`(`qqbar_t` res, const `qqbar_t` x)

Sets res to the complex conjugate of x .

void `qqbar_re`(`qqbar_t` res, const `qqbar_t` x)

Sets res to the real part of x .

void `qqbar_im`(`qqbar_t` res, const `qqbar_t` x)

Sets res to the imaginary part of x .

void `qqbar_re_im`(`qqbar_t` res1, `qqbar_t` res2, const `qqbar_t` x)

Sets $res1$ to the real part of x and $res2$ to the imaginary part of x .

void `qqbar_abs`(`qqbar_t` res, const `qqbar_t` x)

Sets res to the absolute value of x :

void `qqbar_abs2`(`qqbar_t` res, const `qqbar_t` x)

Sets res to the square of the absolute value of x .

void **qqbar_sgn**(*qqbar_t* res, const *qqbar_t* x)

Sets *res* to the complex sign of *x*, defined as 0 if *x* is zero and as $x/|x|$ otherwise.

int **qqbar_sgn_re**(const *qqbar_t* x)

Returns the sign of the real part of *x* (-1, 0 or +1).

int **qqbar_sgn_im**(const *qqbar_t* x)

Returns the sign of the imaginary part of *x* (-1, 0 or +1).

int **qqbar_csgn**(const *qqbar_t* x)

Returns the extension of the real sign function taking the value 1 for *x* strictly in the right half plane, -1 for *x* strictly in the left half plane, and the sign of the imaginary part when *x* is on the imaginary axis. Equivalently, $\text{csgn}(x) = x/\sqrt{x^2}$ except that the value is 0 when *x* is zero.

8.4.11 Integer parts

void **qqbar_floor**(*fmpz_t* res, const *qqbar_t* x)

Sets *res* to the floor function of *x*. If *x* is not real, the value is defined as the floor function of the real part of *x*.

void **qqbar_ceil**(*fmpz_t* res, const *qqbar_t* x)

Sets *res* to the ceiling function of *x*. If *x* is not real, the value is defined as the ceiling function of the real part of *x*.

8.4.12 Arithmetic

void **qqbar_neg**(*qqbar_t* res, const *qqbar_t* x)

Sets *res* to the negation of *x*.

void **qqbar_add**(*qqbar_t* res, const *qqbar_t* x, const *qqbar_t* y)

void **qqbar_add_fmpq**(*qqbar_t* res, const *qqbar_t* x, const *fmpq_t* y)

void **qqbar_add_fmpz**(*qqbar_t* res, const *qqbar_t* x, const *fmpz_t* y)

void **qqbar_add_ui**(*qqbar_t* res, const *qqbar_t* x, *ulong* y)

void **qqbar_add_si**(*qqbar_t* res, const *qqbar_t* x, *slong* y)

Sets *res* to the sum of *x* and *y*.

void **qqbar_sub**(*qqbar_t* res, const *qqbar_t* x, const *qqbar_t* y)

void **qqbar_sub_fmpq**(*qqbar_t* res, const *qqbar_t* x, const *fmpq_t* y)

void **qqbar_sub_fmpz**(*qqbar_t* res, const *qqbar_t* x, const *fmpz_t* y)

void **qqbar_sub_ui**(*qqbar_t* res, const *qqbar_t* x, *ulong* y)

void **qqbar_sub_si**(*qqbar_t* res, const *qqbar_t* x, *slong* y)

void **qqbar_fmpq_sub**(*qqbar_t* res, const *fmpq_t* x, const *qqbar_t* y)

void **qqbar_fmpz_sub**(*qqbar_t* res, const *fmpz_t* x, const *qqbar_t* y)

void **qqbar_ui_sub**(*qqbar_t* res, *ulong* x, const *qqbar_t* y)

void **qqbar_si_sub**(*qqbar_t* res, *slong* x, const *qqbar_t* y)

Sets *res* to the difference of *x* and *y*.

void **qqbar_mul**(*qqbar_t* res, const *qqbar_t* x, const *qqbar_t* y)

void **qqbar_mul_fmpq**(*qqbar_t* res, const *qqbar_t* x, const *fmpq_t* y)

void **qqbar_mul_fmpz**(*qqbar_t* res, const *qqbar_t* x, const *fmpz_t* y)

void **qqbar_mul_ui**(*qqbar_t* res, const *qqbar_t* x, *ulong* y)

void **qqbar_mul_si**(*qqbar_t* res, const *qqbar_t* x, *slong* y)

Sets *res* to the product of *x* and *y*.

`void qqbar_mul_2exp_si(qqbar_t res, const qqbar_t x, slong e)`
Sets *res* to *x* multiplied by 2^e .

`void qqbar_sqr(qqbar_t res, const qqbar_t x)`
Sets *res* to the square of *x*.

`void qqbar_inv(qqbar_t res, const qqbar_t x)`
Sets *res* to the multiplicative inverse of *y*. Division by zero calls *flint_abort*.

`void qqbar_div(qqbar_t res, const qqbar_t x, const qqbar_t y)`
`void qqbar_div_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t y)`
`void qqbar_div_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t y)`
`void qqbar_div_ui(qqbar_t res, const qqbar_t x, ulong y)`
`void qqbar_div_si(qqbar_t res, const qqbar_t x, slong y)`
`void qqbar_fmpq_div(qqbar_t res, const fmpq_t x, const qqbar_t y)`
`void qqbar_fmpz_div(qqbar_t res, const fmpz_t x, const qqbar_t y)`
`void qqbar_ui_div(qqbar_t res, ulong x, const qqbar_t y)`
`void qqbar_si_div(qqbar_t res, slong x, const qqbar_t y)`
Sets *res* to the quotient of *x* and *y*. Division by zero calls *flint_abort*.

`void qqbar_scalar_op(qqbar_t res, const qqbar_t x, const fmpz_t a, const fmpz_t b, const fmpz_t c)`
Sets *res* to the rational affine transformation $(ax + b)/c$, performed as a single operation. There are no restrictions on *a*, *b* and *c* except that *c* must be nonzero. Division by zero calls *flint_abort*.

8.4.13 Powers and roots

`void qqbar_sqrt(qqbar_t res, const qqbar_t x)`
`void qqbar_sqrt_ui(qqbar_t res, ulong x)`
Sets *res* to the principal square root of *x*.

`void qqbar_rsqr(qqbar_t res, const qqbar_t x)`
Sets *res* to the reciprocal of the principal square root of *x*. Division by zero calls *flint_abort*.

`void qqbar_pow_ui(qqbar_t res, const qqbar_t x, ulong n)`
`void qqbar_pow_si(qqbar_t res, const qqbar_t x, slong n)`
`void qqbar_pow_fmpz(qqbar_t res, const qqbar_t x, const fmpz_t n)`
`void qqbar_pow_fmpq(qqbar_t res, const qqbar_t x, const fmpq_t n)`
Sets *res* to *x* raised to the *n*-th power. Raising zero to a negative power aborts.

`void qqbar_root_ui(qqbar_t res, const qqbar_t x, ulong n)`
`void qqbar_fmpq_root_ui(qqbar_t res, const fmpq_t x, ulong n)`
Sets *res* to the principal *n*-th root of *x*. The order *n* must be positive.

`void qqbar_fmpq_pow_si_ui(qqbar_t res, const fmpq_t x, slong m, ulong n)`
Sets *res* to the principal branch of $x^{m/n}$. The order *n* must be positive. Division by zero calls *flint_abort*.

`int qqbar_pow(qqbar_t res, const qqbar_t x, const qqbar_t y)`
General exponentiation: if x^y is an algebraic number, sets *res* to this value and returns 1. If x^y is transcendental or undefined, returns 0. Note that this function returns 0 instead of aborting on division zero.

8.4.14 Numerical enclosures

The following functions guarantee a polished output in which both the real and imaginary parts are accurate to *prec* bits and exact when exactly representable (that is, when a real or imaginary part is a sufficiently small dyadic number). In some cases, the computations needed to polish the output may be expensive. When polish is unnecessary, `qqbar_enclosure_raw()` may be used instead. Alternatively, `qqbar_cache_enclosure()` can be used to avoid recomputations.

void `qqbar_get_acb`(*acb_t* res, const *qqbar_t* x, *slong* prec)

Sets *res* to an enclosure of *x* rounded to *prec* bits.

void `qqbar_get_arb`(*arb_t* res, const *qqbar_t* x, *slong* prec)

Sets *res* to an enclosure of *x* rounded to *prec* bits, assuming that *x* is a real number. If *x* is not real, *res* is set to $[\text{NaN} \pm \infty]$.

void `qqbar_get_arb_re`(*arb_t* res, const *qqbar_t* x, *slong* prec)

Sets *res* to an enclosure of the real part of *x* rounded to *prec* bits.

void `qqbar_get_arb_im`(*arb_t* res, const *qqbar_t* x, *slong* prec)

Sets *res* to an enclosure of the imaginary part of *x* rounded to *prec* bits.

void `qqbar_cache_enclosure`(*qqbar_t* res, *slong* prec)

Polishes the internal enclosure of *res* to at least *prec* bits of precision in-place. Normally, *qqbar* operations that need high-precision enclosures compute them on the fly without caching the results; if *res* will be used as an invariant operand for many operations, calling this function as a precomputation step can improve performance.

8.4.15 Numerator and denominator

void `qqbar_denominator`(*fmpz_t* res, const *qqbar_t* y)

Sets *res* to the denominator of *y*, i.e. the leading coefficient of the minimal polynomial of *y*.

void `qqbar_numerator`(*qqbar_t* res, const *qqbar_t* y)

Sets *res* to the numerator of *y*, i.e. *y* multiplied by its denominator.

8.4.16 Conjugates

void `qqbar_conjugates`(*qqbar_ptr* res, const *qqbar_t* x)

Sets the entries of the vector *res* to the *d* algebraic conjugates of *x*, including *x* itself, where *d* is the degree of *x*. The output is sorted in a canonical order (as defined by `qqbar_cmp_root_order()`).

8.4.17 Polynomial evaluation

void `_qqbar_evaluate_fmpq_poly`(*qqbar_t* res, const *fmpz_t* *poly, const *fmpz_t* den, *slong* len, const *qqbar_t* x)

void `qqbar_evaluate_fmpq_poly`(*qqbar_t* res, const *fmpq_poly_t* poly, const *qqbar_t* x)

void `_qqbar_evaluate_fmpz_poly`(*qqbar_t* res, const *fmpz_t* *poly, *slong* len, const *qqbar_t* x)

void `qqbar_evaluate_fmpz_poly`(*qqbar_t* res, const *fmpz_poly_t* poly, const *qqbar_t* x)

Sets *res* to the value of the given polynomial *poly* evaluated at the algebraic number *x*. These methods detect simple special cases and automatically reduce *poly* if its degree is greater or equal to that of the minimal polynomial of *x*. In the generic case, evaluation is done by computing minimal polynomials of representation matrices.

int `qqbar_evaluate_fmpz_mpoly_iter`(*qqbar_t* res, const *fmpz_mpoly_t* poly, *qqbar_srcptr* x, *slong* deg_limit, *slong* bits_limit, const *fmpz_mpoly_ctx_t* ctx)

```
int qqbar_evaluate_fmpz_mpoly_horner(qqbar_t res, const fmpz_mpoly_t poly, qqbar_srcptr x,
                                     slong deg_limit, slong bits_limit, const fmpz_mpoly_ctx_t
                                     ctx)
```

```
int qqbar_evaluate_fmpz_mpoly(qqbar_t res, const fmpz_mpoly_t poly, qqbar_srcptr x, slong
                              deg_limit, slong bits_limit, const fmpz_mpoly_ctx_t ctx)
```

Sets *res* to the value of *poly* evaluated at the algebraic numbers given in the vector *x*. The number of variables is defined by the context object *ctx*.

The parameters *deg_limit* and *bits_limit* define evaluation limits: if any temporary result exceeds these limits (not necessarily the final value, in case of cancellation), the evaluation is aborted and 0 (failure) is returned. If evaluation succeeds, 1 is returned.

The *iter* version iterates over all terms in succession and computes the powers that appear. The *horner* version uses a multivariate implementation of the Horner scheme. The default algorithm currently uses the Horner scheme.

8.4.18 Polynomial roots

```
void qqbar_roots_fmpz_poly(qqbar_ptr res, const fmpz_poly_t poly, int flags)
```

```
void qqbar_roots_fmpz_poly(qqbar_ptr res, const fmpz_poly_t poly, int flags)
```

Sets the entries of the vector *res* to the *d* roots of the polynomial *poly*. Roots with multiplicity appear with repetition in the output array. By default, the roots will be sorted in a convenient canonical order (as defined by `qqbar_cmp_root_order()`). Instances of a repeated root always appear consecutively.

The following *flags* are supported:

- `QQBAR_ROOTS_IRREDUCIBLE` - if set, *poly* is assumed to be irreducible (it may still have constant content), and no polynomial factorization is performed internally.
- `QQBAR_ROOTS_UNSORTED` - if set, the roots will not be guaranteed to be sorted (except for repeated roots being listed consecutively).

```
void qqbar_eigenvalues_fmpz_mat(qqbar_ptr res, const fmpz_mat_t mat, int flags)
```

```
void qqbar_eigenvalues_fmpz_mat(qqbar_ptr res, const fmpz_mat_t mat, int flags)
```

Sets the entries of the vector *res* to the eigenvalues of the square matrix *mat*. These functions compute the characteristic polynomial of *mat* and then call `qqbar_roots_fmpz_poly()` with the same flags.

```
int _qqbar_roots_poly_squarefree(qqbar_ptr roots, qqbar_srcptr coeffs, slong len, slong deg_limit,
                                 slong bits_limit)
```

Writes to the vector *roots* the *d* roots of the polynomial with algebraic number coefficients *coeffs* of length *len* ($d = \text{len} - 1$).

Given the polynomial $f = a_0 + \dots + a_d x^d$ with coefficients in $\overline{\mathbb{Q}}$, we construct an annihilating polynomial with coefficients in $\mathbb{Q}$ as $g = \prod (\tilde{a}_0 + \dots + \tilde{a}_d x^d)$ where the product is taken over all combinations of algebraic conjugates $\tilde{a}_k$ of the input coefficients. The polynomial *g* is subsequently factored to find candidate roots.

The leading coefficient a_d must be nonzero and the polynomial *f* polynomial must be squarefree. To compute roots of a general polynomial which may have repeated roots, it is necessary to perform a squarefree factorization before calling this function. An option is to call `gr_poly_roots()` with a `qqbar` context object, which wraps this function and takes care of the initial squarefree factorization.

Since the product *g* can explode in size very quickly, the *deg_limit* and *bits_limit* parameters allow bounding the degree and working precision. The function returns 1 on success and 0 on failure indicating that such a limit has been exceeded. Setting nonpositive values for the limits removes the restrictions; however, the function can still fail and return 0 in that case if *g* exceeds machine size.

Note: to compute algebraic number roots of polynomials of various other types, use `gr_poly_roots_other()`.

8.4.19 Roots of unity and trigonometric functions

The following functions use word-size integers p and q instead of `fmpq_t` instances to express rational numbers. This is to emphasize that the computations are feasible only with small q in this representation of algebraic numbers since the associated minimal polynomials have degree $O(q)$. The input p and q do not need to be reduced *a priori*, but should not be close to the word boundaries (they may be added and subtracted internally).

`void qqbar_root_of_unity(qqbar_t res, slong p, ulong q)`

Sets `res` to the root of unity $e^{2\pi ip/q}$.

`int qqbar_is_root_of_unity(slong *p, ulong *q, const qqbar_t x)`

If x is not a root of unity, returns 0. If x is a root of unity, returns 1. If p and q are not `NULL` and x is a root of unity, this also sets p and q to the minimal integers with $0 \leq p < q$ such that $x = e^{2\pi ip/q}$.

`void qqbar_exp_pi_i(qqbar_t res, slong p, ulong q)`

Sets `res` to the root of unity $e^{\pi ip/q}$.

`void qqbar_cos_pi(qqbar_t res, slong p, ulong q)`

`void qqbar_sin_pi(qqbar_t res, slong p, ulong q)`

`int qqbar_tan_pi(qqbar_t res, slong p, ulong q)`

`int qqbar_cot_pi(qqbar_t res, slong p, ulong q)`

`int qqbar_sec_pi(qqbar_t res, slong p, ulong q)`

`int qqbar_csc_pi(qqbar_t res, slong p, ulong q)`

Sets `res` to the trigonometric function $\cos(\pi x)$, $\sin(\pi x)$, etc., with $x = \frac{p}{q}$. The functions `tan`, `cot`, `sec` and `csc` return the flag 1 if the value exists, and return 0 if the evaluation point is a pole of the function.

`int qqbar_log_pi_i(slong *p, ulong *q, const qqbar_t x)`

If $y = \log(x)/(\pi i)$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $-1 < y \leq 1$ and returns 1. If y is not algebraic, returns 0.

`int qqbar_atan_pi(slong *p, ulong *q, const qqbar_t x)`

If $y = \operatorname{atan}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $|y| < \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

`int qqbar_asin_pi(slong *p, ulong *q, const qqbar_t x)`

If $y = \operatorname{asin}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $|y| \leq \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

`int qqbar_acos_pi(slong *p, ulong *q, const qqbar_t x)`

If $y = \operatorname{acos}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $0 \leq y \leq 1$ and returns 1. If y is not algebraic, returns 0.

`int qqbar_acot_pi(slong *p, ulong *q, const qqbar_t x)`

If $y = \operatorname{acot}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $-\frac{1}{2} < y \leq \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

`int qqbar_asec_pi(slong *p, ulong *q, const qqbar_t x)`

If $y = \operatorname{asec}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $0 \leq y \leq 1$ and returns 1. If y is not algebraic, returns 0.

`int qqbar_acsc_pi(slong *p, ulong *q, const qqbar_t x)`

If $y = \operatorname{acsc}(x)/\pi$ is algebraic, and hence necessarily rational, sets $y = p/q$ to the reduced such fraction with $-\frac{1}{2} \leq y \leq \frac{1}{2}$ and returns 1. If y is not algebraic, returns 0.

8.4.20 Guessing and simplification

int **qqbar_guess**(*qqbar_t* res, const *acb_t* z, *slong* max_deg, *slong* max_bits, int flags, *slong* prec)

Attempts to find an algebraic number *res* of degree at most *max_deg* and height at most *max_bits* bits matching the numerical enclosure *z*. The return flag indicates success. This is only a heuristic method, and the return flag neither implies a rigorous proof that *res* is the correct result, nor a rigorous proof that no suitable algebraic number with the given *max_deg* and *max_bits* exists. (Proof of nonexistence could in principle be computed, but this is not yet implemented.)

The working precision *prec* should normally be the same as the precision used to compute *z*. It does not make much sense to run this algorithm with precision smaller than $O(\max\_deg \cdot \max\_bits)$.

This function does a single iteration at the target *max_deg*, *max_bits*, and *prec*. For best performance, one should invoke this function repeatedly with successively larger parameters when the size of the intended solution is unknown or may be much smaller than a worst-case bound.

int **qqbar_express_in_field**(*fmpq_poly_t* res, const *qqbar_t* alpha, const *qqbar_t* x, *slong* max_bits, int flags, *slong* prec)

Attempts to express *x* in the number field generated by *alpha*, returning success (0 or 1). On success, *res* is set to a polynomial *f* of degree less than the degree of *alpha* and with height (counting both the numerator and the denominator, when the coefficients of *g* are put on a common denominator) bounded by *max_bits* bits, such that $f(\alpha) = x$.

(Exception: the *max_bits* parameter is currently ignored if *x* is rational, in which case *res* is just set to the value of *x*.)

This function looks for a linear relation heuristically using a working precision of *prec* bits. If *x* is expressible in terms of *alpha*, then this function is guaranteed to succeed when *prec* is taken large enough. The identity $f(\alpha) = x$ is checked rigorously, i.e. a return value of 1 implies a proof of correctness. In principle, choosing a sufficiently large *prec* can be used to prove that *x* does not lie in the field generated by *alpha*, but the present implementation does not support doing so automatically.

This function does a single iteration at the target *max_bits* and *prec*. For best performance, one should invoke this function repeatedly with successively larger parameters when the size of the intended solution is unknown or may be much smaller than a worst-case bound.

8.4.21 Symbolic expressions and conversion to radicals

void **qqbar_get_quadratic**(*fmpz_t* a, *fmpz_t* b, *fmpz_t* c, *fmpz_t* q, const *qqbar_t* x, int factoring)

Assuming that *x* has degree 1 or 2, computes integers *a*, *b*, *c* and *q* such that

$$x = \frac{a + b\sqrt{c}}{q}$$

and such that *c* is not a perfect square, *q* is positive, and *q* has no content in common with both *a* and *b*. In other words, this determines a quadratic field $\mathbb{Q}(\sqrt{c})$ containing *x*, and then finds the canonical reduced coefficients *a*, *b* and *q* expressing *x* in this field. For convenience, this function supports rational *x*, for which *b* and *c* will both be set to zero. The following remarks apply to irrationals.

The radicand *c* will not be a perfect square, but will not automatically be squarefree since this would require factoring the discriminant. As a special case, *c* will be set to -1 if *x* is a Gaussian rational number. Otherwise, behavior is controlled by the *factoring* parameter.

- If *factoring* is 0, no factorization is performed apart from removing powers of two.
- If *factoring* is 1, a complete factorization is performed (*c* will be minimal). This can be very expensive if the discriminant is large.

- If *factoring* is 2, a smooth factorization is performed to remove small factors from *c*. This is a tradeoff that provides pretty output in most cases while avoiding extreme worst-case slowdown. The smooth factorization guarantees finding all small factors (up to some trial division limit determined internally by FLINT), but large factors are only found heuristically.

int `qqbar_set_fexpr`(*qqbar_t* res, const *fexpr_t* expr)

Sets *res* to the algebraic number represented by the symbolic expression *expr*, returning 1 on success and 0 on failure.

This function performs a “static” evaluation using *qqbar* arithmetic, supporting only closed-form expressions with explicitly algebraic subexpressions. It can be used to recover values generated by `qqbar_get_expr_formula()` and variants. For evaluating more complex expressions involving other types of values or requiring symbolic simplifications, the user should preprocess *expr* so that it is in a form which can be parsed by `qqbar_set_fexpr()`.

The following expressions are supported:

- Integer constants
- Arithmetic operations with algebraic operands
- Square roots of algebraic numbers
- Powers with algebraic base and exponent an explicit rational number
- NumberI, GoldenRatio, RootOfUnity
- Floor, Ceil, Abs, Sign, Csgn, Conjugate, Re, Im, Max, Min
- Trigonometric functions with argument an explicit rational number times Pi
- Exponentials with argument an explicit rational number times Pi * NumberI
- The Decimal() constructor
- AlgebraicNumberSerialized() (assuming valid data, which is not checked)
- PolynomialRootIndexed()
- PolynomialRootNearest()

Examples of formulas that are not supported, despite the value being an algebraic number:

- $\pi - \pi$ (general transcendental simplifications are not performed)
- $1 / \text{Infinity}$ (only numbers are handled)
- `Sum(n, For(n, 1, 10))` (only static evaluation is performed)

void `qqbar_get_fexpr_repr`(*fexpr_t* res, const *qqbar_t* x)

Sets *res* to a symbolic expression reflecting the exact internal representation of *x*. The output will have the form `AlgebraicNumberSerialized(List(coeffs), enclosure)`. The output can be converted back to a *qqbar_t* value using `qqbar_set_fexpr()`. This is the recommended format for serializing algebraic numbers as it requires minimal computation, but it has the disadvantage of not being human-readable.

void `qqbar_get_fexpr_root_nearest`(*fexpr_t* res, const *qqbar_t* x)

Sets *res* to a symbolic expression unambiguously describing *x* in the form `PolynomialRootNearest(List(coeffs), point)` where *point* is an approximation of *x* guaranteed to be closer to *x* than any conjugate root. The output can be converted back to a *qqbar_t* value using `qqbar_set_fexpr()`. This is a useful format for human-readable presentation, but serialization and deserialization can be expensive.

void `qqbar_get_fexpr_root_indexed`(*fexpr_t* res, const *qqbar_t* x)

Sets *res* to a symbolic expression unambiguously describing *x* in the form `PolynomialRootIndexed(List(coeffs), index)` where *index* is the index of *x* among its conjugate roots in the builtin root sort order. The output can be converted back to a *qqbar_t*

value using `qqbar_set_fexpr()`. This is a useful format for human-readable presentation when the numerical value is important, but serialization and deserialization can be expensive.

int `qqbar_get_fexpr_formula`(*fexpr_t* res, const *qqbar_t* x, *ulong* flags)

Attempts to express the algebraic number x as a closed-form expression using arithmetic operations, radicals, and possibly exponentials or trigonometric functions, but without using `PolynomialRootNearest` or `PolynomialRootIndexed`. Returns 0 on failure and 1 on success.

The *flags* parameter toggles different methods for generating formulas. It can be set to any combination of the following. If *flags* is 0, only rational numbers will be handled.

QQBAR_FORMULA_ALL

Toggles all methods (potentially expensive).

QQBAR_FORMULA_GAUSSIANS

Detect Gaussian rational numbers $a + bi$.

QQBAR_FORMULA_QUADRATICS

Solve quadratics in the form $a + b\sqrt{d}$.

QQBAR_FORMULA_CYCLOTOMICS

Detect elements of cyclotomic fields. This works by trying plausible cyclotomic fields (based on the degree of the input), using LLL to find candidate number field elements, and certifying candidates through an exact computation. Detection is heuristic and is not guaranteed to find all cyclotomic numbers.

QQBAR_FORMULA_CUBICS

QQBAR_FORMULA_QUARTICS

QQBAR_FORMULA_QUINTICS

Solve polynomials of degree 3, 4 and (where applicable) 5 using cubic, quartic and quintic formulas (not yet implemented).

QQBAR_FORMULA_DEPRESSION

Use depression to try to generate simpler numbers.

QQBAR_FORMULA_DEFLATION

Use deflation to try to generate simpler numbers. This allows handling number of the form $a^{1/n}$ where a can be represented in closed form.

QQBAR_FORMULA_SEPARATION

Try separating real and imaginary parts or sign and magnitude of complex numbers. This allows handling numbers of the form $a+bi$ or $m \cdot s$ (with $m > 0$, $|s| = 1$) where a and b or m and s can be represented in closed form. This is only attempted as a fallback after other methods fail: if an explicit Cartesian or magnitude-sign represented is desired, the user should manually separate the number into complex parts before calling `qqbar_get_fexpr_formula()`.

QQBAR_FORMULA_EXP_FORM

QQBAR_FORMULA_TRIG_FORM

QQBAR_FORMULA_RADICAL_FORM

QQBAR_FORMULA_AUTO_FORM

Select output form for cyclotomic numbers. The *auto* form (equivalent to no flags being set) results in radicals for numbers of low degree, trigonometric functions for real numbers, and complex exponentials for nonreal numbers. The other flags (not fully implemented) can be used to force exponential form, trigonometric form, or radical form.

8.4.22 Internal functions

void `qqbar_fmpz_poly_composed_op`(*fmpz_poly_t* res, const *fmpz_poly_t* A, const *fmpz_poly_t* B, int op)

Given nonconstant polynomials A and B , sets res to a polynomial whose roots are $a + b$, $a - b$, ab or a/b for all roots a of A and all roots b of B . The parameter op selects the arithmetic operation: 0 for addition, 1 for subtraction, 2 for multiplication and 3 for division. If op is 3, B must not have zero as a root.

void `qqbar_binary_op`(*qqbar_t* res, const *qqbar_t* x, const *qqbar_t* y, int op)

Performs a binary operation using a generic algorithm. This does not check for special cases.

int `_qqbar_validate_uniqueness`(*acb_t* res, const *fmpz_poly_t* poly, const *acb_t* z, *slong* max_prec)

Given z known to be an enclosure of at least one root of $poly$, certifies that the enclosure contains a unique root, and in that case sets res to a new (possibly improved) enclosure for the same root, returning 1. Returns 0 if uniqueness cannot be certified.

The enclosure is validated by performing a single step with the interval Newton method. The working precision is determined from the accuracy of z , but limited by max_prec bits.

This method slightly inflates the enclosure z to improve the chances that the interval Newton step will succeed. Uniqueness on this larger interval implies uniqueness of the original interval, but not existence; when existence has not been ensured a priori, `_qqbar_validate_existence_uniqueness()` should be used instead.

int `_qqbar_validate_existence_uniqueness`(*acb_t* res, const *fmpz_poly_t* poly, const *acb_t* z, *slong* max_prec)

Given any complex interval z , certifies that the enclosure contains a unique root of $poly$, and in that case sets res to a new (possibly improved) enclosure for the same root, returning 1. Returns 0 if existence and uniqueness cannot be certified.

The enclosure is validated by performing a single step with the interval Newton method. The working precision is determined from the accuracy of z , but limited by max_prec bits.

void `_qqbar_enclosure_raw`(*acb_t* res, const *fmpz_poly_t* poly, const *acb_t* z, *slong* prec)

void `qqbar_enclosure_raw`(*acb_t* res, const *qqbar_t* x, *slong* prec)

Sets res to an enclosure of x accurate to about $prec$ bits (the actual accuracy can be slightly lower, or higher).

This function uses repeated interval Newton steps to polish the initial enclosure z , doubling the working precision each time. If any step fails to improve the accuracy significantly, the root is recomputed from scratch to higher precision.

If the initial enclosure is accurate enough, res is set to this value without rounding and without further computation.

int `_qqbar_acb_linddep`(*fmpz_t* *rel, *acb_srcptr* vec, *slong* len, int check, *slong* prec)

Attempts to find an integer vector rel giving a linear relation between the elements of the real or complex vector vec , using the LLL algorithm.

The working precision is set to the minimum of $prec$ and the relative accuracy of vec (that is, the difference between the largest magnitude and the largest error magnitude within vec). 95% of the bits within the working precision are used for the LLL matrix, and the remaining 5% bits are used to validate the linear relation by evaluating the linear combination and checking that the resulting interval contains zero. This validation does not prove the existence or nonexistence of a linear relation, but it provides a quick heuristic way to eliminate spurious relations.

If $check$ is set, the return value indicates whether the validation was successful; otherwise, the return value simply indicates whether the algorithm was executed normally (failure may occur, for example, if the input vector is non-finite).

In principle, this method can be used to produce a proof that no linear relation exists with coefficients up to a specified bit size, but this has not yet been implemented.

REAL AND COMPLEX NUMBERS

9.1 Feature overview

Ball arithmetic, also known as mid-rad interval arithmetic, is an extension of floating-point arithmetic in which an error bound is attached to each variable. This allows computing with real and complex numbers in a mathematically rigorous way.

With plain floating-point arithmetic, the user must do an error analysis to guarantee that results are correct. Manual error analysis is time-consuming and bug-prone. Ball arithmetic effectively makes error analysis automatic.

In traditional (inf-sup) interval arithmetic, both endpoints of an interval $[a, b]$ are full-precision numbers, which makes interval arithmetic twice as expensive as floating-point arithmetic. In ball arithmetic, only the midpoint m of an interval $[m \pm r]$ is a full-precision number, and a few bits suffice for the radius r . At high precision, ball arithmetic is therefore not more expensive than plain floating-point arithmetic.

Joris van der Hoeven's paper [Hoe2009] is a good introduction to the subject.

Other implementations of ball arithmetic include [iRRAM](#) and [Mathemagix](#). Arb differs from earlier implementations in technical aspects of the implementation, which makes certain computations more efficient. It also provides a more comprehensive low-level interface, giving the user full access to the internals. Finally, it implements a wider range of transcendental functions, covering a large portion of the special functions in standard reference works such as [NIST2012].

The ball arithmetic routines in FLINT (formerly the standalone Arb library) are designed for computer algebra and computational number theory, but may be useful in any area demanding reliable or precise numerical computing. The contents include:

- A module (*arf*) for correctly rounded arbitrary-precision floating-point arithmetic. Arb's floating-point numbers have a few special features, such as arbitrary-size exponents (useful for combinatorics and asymptotics) and dynamic allocation (facilitating implementation of hybrid integer/floating-point and mixed-precision algorithms).
- A module (*mag*) for representing magnitudes (error bounds) more efficiently than with an arbitrary-precision floating-point type.
- A module (*arb*) for real ball arithmetic, where a ball is implemented as an *arf* midpoint and a *mag* radius.
- A module (*acb*) for complex numbers in rectangular form, represented as pairs of real balls.
- Modules (*arb_poly*, *acb_poly*) for polynomials or power series over the real and complex numbers, implemented using balls as coefficients, with asymptotically fast polynomial multiplication and many other operations.
- Modules (*arb_mat*, *acb_mat*) for matrices over the real and complex numbers, implemented using balls as coefficients. At the moment, only rudimentary linear algebra operations are provided.
- Functions for high-precision evaluation of various mathematical constants and special functions, implemented using ball arithmetic with rigorous error bounds.

9.2 Using ball arithmetic

This section gives an introduction to working with real numbers in Arb (see [arb.h – real numbers](#) for the API and technical documentation). The general principles carry over to complex numbers, polynomials and matrices.

9.2.1 Ball semantics

Let $f : A \rightarrow B$ be a function. A ball implementation of f is a function F that maps sets $X \subseteq A$ to sets $F(X) \subseteq B$ subject to the following rule:

For all $x \in X$, we have $f(x) \in F(X)$.

In other words, $F(X)$ is an *enclosure* for the set $\{f(x) : x \in X\}$. This rule is sometimes called the *inclusion principle*.

Throughout the documentation (except where otherwise noted), we will simply write $f(x)$ instead of $F(X)$ when describing ball implementations of pointwise-defined mathematical functions, understanding that the input is a set of point values and that the output is an enclosure.

General subsets of $\mathbb{R}$ are not possible to represent on a computer. Instead, we work with subsets of the form $[m \pm r] = [m - r, m + r]$ where the midpoint m and radius r are binary floating-point numbers, i.e. numbers of the form $u2^v$ with $u, v \in \mathbb{Z}$ (to make this scheme complete, we also need to adjoin the special floating-point values $-\infty$, $+\infty$ and NaN).

Given a ball $[m \pm r]$ with $m \in \mathbb{R}$ (not necessarily a floating-point number), we can always round m to a nearby floating-point number that has at most $prec$ bits in the component u , and add an upper bound for the rounding error to r . In Arb, ball functions that take a $prec$ argument as input (e.g. [arb_add\(\)](#)) always round their output to $prec$ bits. Some functions are always exact (e.g. [arb_neg\(\)](#)), and thus do not take a $prec$ argument.

The programming interface resembles that of GMP. Each [arb_t](#) variable must be initialized with [arb_init\(\)](#) before use (this also sets its value to zero), and deallocated with [arb_clear\(\)](#) after use. Variables have pass-by-reference semantics. In the list of arguments to a function, output variables come first, followed by input variables, and finally the precision:

```
#include "arb.h"

int main()
{
    arb_t x, y;
    arb_init(x); arb_init(y);
    arb_set_ui(x, 3);          /* x = 3 */
    arb_const_pi(y, 128);     /* y = pi, to 128 bits */
    arb_sub(y, y, x, 53);     /* y = y - x, to 53 bits */
    arb_clear(x); arb_clear(y);
}
```

9.2.2 Binary and decimal

While the internal representation uses binary floating-point numbers, it is usually preferable to print numbers in decimal. The binary-to-decimal conversion generally requires rounding. Three different methods are available for printing a number to standard output:

- [arb_print\(\)](#) shows the exact internal representation of a ball, with binary exponents.
- [arb_printd\(\)](#) shows an inexact view of the internal representation, approximated by decimal floating-point numbers.

- `arb_printn()` shows a *decimal ball* that is guaranteed to be an enclosure of the binary floating-point ball. By default, it only prints digits in the midpoint that are certain to be correct, up to an error of at most one unit in the last place. Converting from binary to decimal is generally inexact, and the output of this method takes this rounding into account when printing the radius.

This snippet computes a 53-bit enclosure of π and prints it in three ways:

```
arb_const_pi(x, 53);
arb_print(x); printf("\n");
arb_printd(x, 20); printf("\n");
arb_printn(x, 20, 0); printf("\n");
```

The output is:

```
(884279719003555 * 2^-48) +/- (536870913 * 2^-80)
3.141592653589793116 +/- 4.4409e-16
[3.141592653589793 +/- 5.61e-16]
```

The `arb_get_str()` and `arb_set_str()` methods are useful for converting rigorously between decimal strings and binary balls (`arb_get_str()` produces the same string as `arb_printn()`, and `arb_set_str()` can parse such strings back).

A potential mistake is to create a ball from a `double` constant such as 2.3, when this actually represents 2.2999999999999982236431605997495353221893310546875. To produce a ball containing the rational number 23/10, one of the following can be used:

```
arb_set_str(x, "2.3", prec)

arb_set_ui(x, 23);
arb_div_ui(x, x, 10, prec)

fmpq_set_si(q, 23, 10); /* q is a FLINT fmpq_t */
arb_set_fmpq(x, q, prec);
```

9.2.3 Quality of enclosures

The main problem when working with ball arithmetic (or interval arithmetic) is *overestimation*. In general, the enclosure of a value or set of values as computed with ball arithmetic will be larger than the smallest possible enclosure.

Overestimation results naturally from rounding errors and cancellations in the individual steps of a calculation. As a general principle, formula rewriting techniques that make floating-point code more numerically stable also make ball arithmetic code more numerically stable, in the sense of producing tighter enclosures.

As a result of the *dependency problem*, ball or interval arithmetic can produce error bounds that are much larger than the actual numerical errors resulting from doing floating-point arithmetic. Consider the expression $(x + 1) - x$ as an example. When evaluated in floating-point arithmetic, x may have a large initial error. However, that error will cancel itself out in the subtraction, so that the result equals 1 (except perhaps for a small rounding error left from the operation $x + 1$). In ball arithmetic, dependent errors add up instead of cancelling out. If $x = [3 \pm 0.1]$, the result will be $[1 \pm 0.2]$, where the error bound has doubled. In unfavorable circumstances, error bounds can grow exponentially with the number of steps.

If all inputs to a calculation are “point values”, i.e. exact numbers and known mathematical constants that can be approximated arbitrarily closely (such as π), then an error of order 2^n can typically be overcome by working with n extra bits of precision, increasing the computation time by an amount that is polynomial in n . In certain situations, however, overestimation leads to exponential slowdown or even failure of an algorithm to converge. For example, root-finding algorithms that refine the result iteratively may fail to converge in ball arithmetic, even if they do converge in plain floating-point arithmetic.

Therefore, ball arithmetic is not a silver bullet: there will always be situations where some amount of numerical or mathematical analysis is required. Some experimentation may be required to find whether (and how) it can be used effectively for a given problem.

9.2.4 Predicates

A ball implementation of a predicate $f : \mathbb{R} \rightarrow \{\text{True}, \text{False}\}$ would need to be able to return a third logical value indicating that the result could be either True or False. In most cases, predicates in Arb are implemented as functions that return the *int* value 1 to indicate that the result certainly is True, and the *int* value 0 to indicate that the result could be either True or False. To test whether a predicate certainly is False, the user must test whether the negated predicate certainly is True.

For example, the following code would *not* be correct in general:

```
if (arb_is_positive(x))
{
    ... /* do things assuming that x > 0 */
}
else
{
    ... /* do things assuming that x <= 0 */
}
```

Instead, the following can be used:

```
if (arb_is_positive(x))
{
    ... /* do things assuming that x > 0 */
}
else if (arb_is_nonpositive(x))
{
    ... /* do things assuming that x <= 0 */
}
else
{
    ... /* do things assuming that the sign of x is unknown */
}
```

Likewise, we will write $x \leq y$ in mathematical notation with the meaning that $x \leq y$ holds for all $x \in X, y \in Y$ where X and Y are balls.

Note that some predicates such as `arb_overlaps()` and `arb_contains()` actually are predicates on balls viewed as sets, and not ball implementations of pointwise predicates.

Some predicates are also complementary. For example `arb_contains_zero()` tests whether the input ball contains the point zero. Negated, it is equivalent to `arb_is_nonzero()`, and complementary to `arb_is_zero()` as a pointwise predicate:

```
if (arb_is_zero(x))
{
    ... /* do things assuming that x = 0 */
}
#ifdef 1
else if (arb_is_nonzero(x))
#else
else if (!arb_contains_zero(x)) /* equivalent */
#endif
{
    ... /* do things assuming that x != 0 */
}
```

(continues on next page)

(continued from previous page)

```

}
else
{
    ... /* do things assuming that the sign of x is unknown */
}

```

9.2.5 A worked example: the sine function

We implement the function $\sin(x)$ naively using the Taylor series $\sum_{k=0}^{\infty} (-1)^k x^{2k+1} / (2k+1)!$ and `arb_t` arithmetic. Since there are infinitely many terms, we need to split the series in two parts: a finite sum that can be evaluated directly, and a tail that has to be bounded.

We stop as soon as we reach a term t bounded by $|t| \leq 2^{-prec} < 1$. The terms are alternating and must have decreasing magnitude from that point, so the tail of the series is bounded by $|t|$. We add this magnitude to the radius of the output. Since ball arithmetic automatically bounds the numerical errors resulting from all arithmetic operations, the output `res` is a ball guaranteed to contain $\sin(x)$.

```

#include "arb.h"

void arb_sin_naive(arb_t res, const arb_t x, slong prec)
{
    arb_t s, t, u, tol;
    slong k;
    arb_init(s); arb_init(t); arb_init(u); arb_init(tol);

    arb_one(tol);
    arb_mul_2exp_si(tol, tol, -prec); /* tol = 2^-prec */

    for (k = 0; ; k++)
    {
        arb_pow_ui(t, x, 2 * k + 1, prec);
        arb_fac_ui(u, 2 * k + 1, prec);
        arb_div(t, t, u, prec); /* t = x^(2k+1) / (2k+1)! */

        arb_abs(u, t);
        if (arb_le(u, tol)) /* if |t| <= 2^-prec */
        {
            arb_add_error(s, u); /* add |t| to the radius and stop */
            break;
        }

        if (k % 2 == 0)
            arb_add(s, s, t, prec);
        else
            arb_sub(s, s, t, prec);
    }

    arb_set(res, s);
    arb_clear(s); arb_clear(t); arb_clear(u); arb_clear(tol);
}

```

This algorithm is naive, because the Taylor series is slow to converge and suffers from catastrophic cancellation when $|x|$ is large (we could also improve the efficiency of the code slightly by computing the terms using recurrence relations instead of computing x^k and $k!$ from scratch each iteration).

As a test, we compute $\sin(2016.1)$. The largest term in the Taylor series for $\sin(x)$ reaches a magnitude

of about $x^x/x!$, or about 10^{873} in this case. Therefore, we need over 873 digits (about 3000 bits) of precision to overcome the catastrophic cancellation and determine the result with sufficient accuracy to tell whether it is positive or negative.

```
int main()
{
    arb_t x, y;
    slong prec;
    arb_init(x); arb_init(y);

    for (prec = 64; ; prec *= 2)
    {
        arb_set_str(x, "2016.1", prec);
        arb_sin_naive(y, x, prec);
        printf("Using %5ld bits, sin(x) = ", prec);
        arb_printn(y, 10, 0); printf("\n");
        if (!arb_contains_zero(y)) /* stopping condition */
            break;
    }

    arb_clear(x); arb_clear(y);
}
```

The program produces the following output:

```
Using    64 bits, sin(x) = [+/- 2.67e+859]
Using   128 bits, sin(x) = [+/- 1.30e+840]
Using   256 bits, sin(x) = [+/- 3.60e+801]
Using   512 bits, sin(x) = [+/- 3.01e+724]
Using  1024 bits, sin(x) = [+/- 2.18e+570]
Using  2048 bits, sin(x) = [+/- 1.22e+262]
Using  4096 bits, sin(x) = [-0.7190842207 +/- 1.20e-11]
```

As an exercise, the reader may improve the naive algorithm by making it subtract a well-chosen multiple of 2π from x before invoking the Taylor series (hint: use `arb_const_pi()`, `arb_div()` and `arf_get_fmpz()`). If done correctly, 64 bits of precision should be more than enough to compute $\sin(2016.1)$, and with minor adjustments to the code, the user should be able to compute $\sin(\exp(2016.1))$ quite easily as well.

This example illustrates how ball arithmetic can be used to perform nontrivial calculations. To evaluate an infinite series, the user needs to know how to bound the tail of the series, but everything else is automatic. When evaluating a finite formula that can be expressed completely using built-in functions, all error bounding is automatic from the point of view of the user. In particular, the `arb_sin()` method should be used to compute the sine of a real number; it uses a much more efficient algorithm than the naive code above.

This example also illustrates the “guess-and-verify” paradigm: instead of determining *a priori* the floating-point precision necessary to get a correct result, we *guess* some initial precision, use ball arithmetic to *verify* that the result is accurate enough, and restart with higher precision (or signal failure) if it is not.

If we think of rounding errors as essentially random processes, then a floating-point computation is analogous to a *Monte Carlo algorithm*. Using ball arithmetic to get a verified result effectively turns it into the analog of a *Las Vegas algorithm*, which is a randomized algorithm that always gives a correct result if it terminates, but may fail to terminate (alternatively, instead of actually looping forever, it might signal failure after a certain number of iterations).

The loop will fail to terminate if we attempt to determine the sign of $\sin(\pi)$:


```

Using    64 bits, sin(x) = [+/- 3.96e-18]
Using   128 bits, sin(x) = [+/- 2.17e-37]
Using   256 bits, sin(x) = [+/- 6.10e-76]
Using   512 bits, sin(x) = [+/- 5.13e-153]
Using  1024 bits, sin(x) = [+/- 4.01e-307]
Using  2048 bits, sin(x) = [+/- 2.13e-615]
Using  4096 bits, sin(x) = [+/- 6.85e-1232]
Using  8192 bits, sin(x) = [+/- 6.46e-2465]
Using 16384 bits, sin(x) = [+/- 5.09e-4931]
Using 32768 bits, sin(x) = [+/- 5.41e-9863]
...

```

The sign of a nonzero real number can be decided by computing it to sufficiently high accuracy, but the sign of an expression that is exactly equal to zero cannot be decided by a numerical computation unless the entire computation happens to be exact (in this example, we could use the `arb_sin_pi()` function which computes $\sin(\pi x)$ in one step, with the input $x = 1$).

It is up to the user to implement a stopping criterion appropriate for the circumstances of a given application. For example, breaking when it is clear that $|\sin(x)| < 10^{-10000}$ would allow the program to terminate and convey some meaningful information about the input $x = \pi$, though this would not constitute a mathematical proof that $\sin(\pi) = 0$.

9.2.6 More on precision and accuracy

The relation between the working precision and the accuracy of the output is not always easy predict. The following remarks might help to choose *prec* optimally.

For a ball $[m \pm r]$ it is convenient to define the following notions:

- Absolute error: $e_{abs} = |r|$
- Relative error: $e_{rel} = |r| / \max(0, |m| - |r|)$ (or $e_{rel} = 0$ if $r = m = 0$)
- Absolute accuracy: $a_{abs} = 1/e_{abs}$
- Relative accuracy: $a_{rel} = 1/e_{rel}$

Expressed in bits, one takes the corresponding $\log_2$ values.

Of course, if x is the exact value being approximated, then the “absolute error” so defined is an upper bound for the actual absolute error $|x - m|$ and “absolute accuracy” a lower bound for $1/|x - m|$, etc.

The *prec* argument in Arb should be thought of as controlling the working precision. Generically, when evaluating a fixed expression (that is, when the sequence of operations does not depend on the precision), the absolute or relative error will be bounded by

$$2^{O(1)-prec}$$

where the $O(1)$ term depends on the expression and implementation details of the ball functions used to evaluate it. Accordingly, for an accuracy of p bits, we need to use a working precision $O(1) + p$. If the expression is numerically well-behaved, then the $O(1)$ term will be small, which leads to the heuristic of “adding a few guard bits” (for most basic calculations, 10 or 20 guard bits is enough). If the $O(1)$ term is unknown, then increasing the number of guard bits in exponential steps until the result is accurate enough is generally a good heuristic.

Sometimes, a partially accurate result can be used to estimate the $O(1)$ term. For example, if the goal is to achieve 100 bits of accuracy and a precision of 120 bits yields 80 bits of accuracy, then it is plausible that a precision of just over 140 bits yields 100 bits of accuracy.

Built-in functions in Arb can roughly be characterized as belonging to one of two extremes (though there is actually a spectrum):

- Simple operations, including basic arithmetic operations and many elementary functions. In most cases, for an input $x = [m \pm r]$, $f(x)$ is evaluated by computing $f(m)$ and then separately bounding the *propagated error* $|f(m) - f(m + \varepsilon)|$, $|\varepsilon| \leq r$. The working precision is automatically increased internally so that $f(m)$ is computed to *prec* bits of relative accuracy with an error of at most a few units in the last place (perhaps with rare exceptions). The propagated error can generally be bounded quite tightly as well (see [General formulas and bounds](#)). As a result, the enclosure will be close to the best possible at the given precision, and the user can estimate the precision to use accordingly.
- Complex operations, such as certain higher transcendental functions (for example, the Riemann zeta function). The function is evaluated by performing a sequence of simpler operations, each using ball arithmetic with a working precision of roughly *prec* bits. The sequence of operations might depend on *prec*; for example, an infinite series might be truncated so that the remainder is smaller than 2^{-prec} . The final result can be far from tight, and it is not guaranteed that the error converges to zero as $prec \rightarrow \infty$, though in practice, it should do so in most cases.

In short, the *inclusion principle* is the fundamental contract in Arb. Enclosures computed by built-in functions may or may not be tight enough to be useful, but the hope is that they will be sufficient for most purposes. Tightening the error bounds for more complex operations is a long term optimization goal, which in many cases will require a fair amount of research. A tradeoff also has to be made for efficiency: tighter error bounds allow the user to work with a lower precision, but they may also be much more expensive to compute.

9.2.7 Polynomial time guarantee

Arb provides a soft guarantee that the time used to evaluate a ball function will depend polynomially on *prec* and the bit size of the input, uniformly regardless of the numerical value of the input.

The idea behind this soft guarantee is to allow Arb to be used as a black box to evaluate expressions numerically without potentially slowing down, hanging indefinitely or crashing because of “bad” input such as nested exponentials. By controlling the precision, the user can cancel a computation before it uses up an unreasonable amount of resources, without having to rely on other timeout or exception mechanisms. A result that is feasible but very expensive to compute can still be forced by setting the precision high enough.

As motivation, consider evaluating $\sin(x)$ or $\exp(x)$ with the exact floating-point number $x = 2^{2^n}$ as input. The time and space required to compute an accurate floating-point approximation of $\sin(x)$ or $\exp(x)$ increases as 2^n , in the first case because of the need to subtract an accurate multiple of 2π and in the second case due to the size of the output exponent and the internal subtraction of an accurate multiple of $\log(2)$. This is despite the fact that the size of x as an object in memory only increases linearly with n . Already $n = 33$ would require at least 1 GB of memory, and $n = 100$ would be physically impossible to process. For functions that are computed by direct use of power series expansions, e.g. $f(x) = \sum_{k=0}^{\infty} c_k x^k$, without having fast argument-reduction techniques like those for elementary functions, the time would be exponential in n already when $x = 2^n$.

Therefore, Arb caps internal work parameters (the internal working precision, the number terms of an infinite series to add, etc.) by polynomial, usually linear, functions of *prec*. When the limit is exceeded, the output is set to a crude bound. For example, if x is too large, `arb_sin()` will simply return $[\pm 1]$, and `arb_exp()` will simply return $[\pm \infty]$ if x is positive or $[\pm 2^{-m}]$ if x is negative.

This is not just a failsafe, but occasionally a useful optimization. It is not entirely uncommon to have formulas where one term is modest and another term decreases exponentially, such as:

$$\log(x) + \sin(x) \exp(-x).$$

For example, the reflection formula of the digamma function has a similar structure. When x is large, the right term would be expensive to compute to high relative accuracy. Doing so is unnecessary, however, since a crude bound of $[\pm 1] \cdot [\pm 2^{-m}]$ is enough to evaluate the expression as a whole accurately.

The polynomial time guarantee is “soft” in that there are a few exceptions. For example, the complexity of computing the Riemann zeta function $\zeta(\sigma + it)$ increases linearly with the imaginary height $|t|$ in the

current implementation, and all known algorithms have a complexity of $|t|^\alpha$ where the best known value for α is about 0.3. Input with large $|t|$ is most likely to be given deliberately by users with the explicit intent of evaluating the zeta function itself, so the evaluation is not cut off automatically.

9.3 Technical conventions and potential issues

9.3.1 Integer overflow

When machine-size integers are used for precisions, sizes of integers in bits, lengths of polynomials, and similar quantities that relate to sizes in memory, very few internal checks are performed to verify that such quantities do not overflow.

Precisions and lengths exceeding a small fraction of `LONG_MAX`, say $2^{24} \approx 10^7$ on 32-bit systems, should be regarded as resulting in undefined behavior. On 64-bit systems this should generally not be an issue, since most calculations will exhaust the available memory (or the user's patience waiting for the computation to complete) long before running into integer overflows. However, the user needs to be wary of unintentionally passing input parameters of order `LONG_MAX` or negative parameters where positive parameters are expected, for example due to a runaway loop that repeatedly increases the precision.

Currently, no hard upper limit on the precision is defined, but $2^{24} \approx 10^7$ bits on 32-bit system and $2^{36} \approx 10^{11}$ bits on a 64-bit system can be considered safe for most purposes. The relatively low limit on 64-bit systems is due to the fact that GMP integers are used internally in some algorithms, and GMP integers are limited to 2^{37} bits. The minimum allowed precision is 2 bits.

This caveat does not apply to exponents of floating-point numbers, which are represented as arbitrary-precision integers, nor to integers used as numerical scalars (e.g. `arb_mul_si()`). However, it still applies to conversions and operations where the result is requested exactly and sizes become an issue. For example, trying to convert the floating-point number $2^{2^{100}}$ to an integer could result in anything from a silent wrong value to thrashing followed by a crash, and it is the user's responsibility not to attempt such a thing.

9.3.2 Aliasing

As a rule, Arb allows aliasing of operands. For example, in the function call `arb_add(z, x, y, prec)`, which performs $z \leftarrow x + y$, any two (or all three) of the variables x , y and z are allowed to be the same. Exceptions to this rule are documented explicitly.

The general rule that input and output variables can be aliased with each other only applies to variables of the same type (ignoring `const` qualifiers on input variables – a special case is that `arb_srcptr` is considered the `const` version of `arb_ptr`). This is a natural extension of the so-called *strict aliasing rule* in C.

For example, in `arb_poly_evaluate()` which evaluates $y = f(x)$ for a polynomial f , the output variable y is not allowed to be a pointer to one of the coefficients of f (but aliasing between x and y or between x and the coefficients of f is allowed). This also applies to `_arb_poly_evaluate()`: for the purposes of aliasing, `arb_srcptr` (the type of the coefficient array within f) and `arb_t` (the type of x) are *not* considered to be the same type, and therefore must not be aliased with each other, even though an `arb_ptr/arb_srcptr` variable pointing to a length 1 array would otherwise be interchangeable with an `arb_t/const arb_t`.

Moreover, in functions that allow aliasing between an input array and an output array, the arrays must either be identical or completely disjoint, never partially overlapping.

There are natural exceptions to these aliasing restrictions, which may be used internally without being documented explicitly. However, third party code should avoid relying on such exceptions.

An important caveat applies to **aliasing of input variables**. Identical pointers are understood to give permission for **algebraic simplification**. This assumption is made to improve performance. For

example, the call `arb_mul(z, x, x, prec)` sets z to a ball enclosing the set

$$\{t^2 : t \in x\}$$

and not the (generally larger) set

$$\{tu : t \in x, u \in x\}.$$

If the user knows that two values x and y both lie in the interval $[-1, 1]$ and wants to compute an enclosure for $f(x, y)$, then it would be a mistake to create an `arb_t` variable x enclosing $[-1, 1]$ and reusing the same variable for y , calling $f(x, x)$. Instead, the user has to create a distinct variable y also enclosing $[-1, 1]$.

Algebraic simplification is not guaranteed to occur. For example, `arb_add(z, x, x, prec)` and `arb_sub(z, x, x, prec)` currently do not implement this optimization. It is better to use `arb_mul_2exp_si(z, x, 1)` and `arb_zero(z)`, respectively.

9.3.3 Thread safety and caches

Arb should be fully threadsafe, provided that both MPFR and FLINT have been built in threadsafe mode. Use `flint_set_num_threads()` to set the number of threads that Arb is allowed to use internally for single computations (this is currently only exploited by a handful of operations). Please note that thread safety is only tested minimally, and extra caution when developing multithreaded code is therefore recommended.

Arb may cache some data (such as the value of π and Bernoulli numbers) to speed up various computations. In threadsafe mode, caches use thread-local storage. There is currently no way to save memory and avoid recomputation by having several threads share the same cache. Caches can be freed by calling the `flint_cleanup()` function. To avoid memory leaks, the user should call `flint_cleanup()` when exiting a thread. It is also recommended to call `flint_cleanup()` when exiting the main program (this should result in a clean output when running `Valgrind`, and can help catching memory issues).

There does not seem to be an obvious way to make sure that `flint_cleanup()` is called when exiting a thread using OpenMP. A possible solution to this problem is to use OpenMP sections, or to use C++ and create a thread-local object whose destructor invokes `flint_cleanup()`.

9.3.4 Use of hardware floating-point arithmetic

Arb uses hardware floating-point arithmetic (the `double` type in C) in two different ways.

First, `double` arithmetic as well as transcendental `libm` functions (such as `exp`, `log`) are used to select parameters heuristically in various algorithms. Such heuristic use of approximate arithmetic does not affect correctness: when any error bounds depend on the parameters, the error bounds are evaluated separately using rigorous methods. At worst, flaws in the floating-point arithmetic on a particular machine could cause an algorithm to become inefficient due to inefficient parameters being selected.

Second, `double` arithmetic is used internally for some rigorous error bound calculations. To guarantee correctness, we make the following assumptions. With the stated exceptions, these should hold on all commonly used platforms.

- A `double` uses the standard IEEE 754 format (with a 53-bit significand, 11-bit exponent, encoding of infinities and NaNs, etc.)
- We assume that the compiler does not perform “unsafe” floating-point optimizations, such as reordering of operations. Unsafe optimizations are disabled by default in most modern C compilers, including GCC and Clang. The exception appears to be the Intel C++ compiler, which does some unsafe optimizations by default. These must be disabled by the user.
- We do not assume that floating-point operations are correctly rounded (a counterexample is the x87 FPU), or that rounding is done in any particular direction (the rounding mode may have been changed by the user). We assume that any floating-point operation is done with at most 1.1 ulp error.

- We do not assume that underflow or overflow behaves in a particular way (we only use doubles that fit in the regular exponent range, or explicit infinities).
- We do not use transcendental `libm` functions, since these can have errors of several ulps, and there is unfortunately no way to get guaranteed bounds. However, we do use functions such as `ldexp` and `sqrt`, which we assume to be correctly implemented.

9.3.5 Interface changes

Most of the core API should be stable at this point, and significant compatibility-breaking changes will be specified in the release notes.

In general, Arb does not distinguish between “private” and “public” parts of the API. The implementation is meant to be transparent by design. All methods are intended to be fully documented and tested (exceptions to this are mainly due to lack of time on part of the author). The user should use common sense to determine whether a function is concerned with implementation details, making it likely to change as the implementation changes in the future. The interface of `arb_add()` is probably not going to change in the next version, but `_arb_get_mpn_fixed_mod_pi4()` just might.

9.3.6 General note on correctness

Except where otherwise specified, Arb is designed to produce provably correct error bounds. The code has been written carefully, and the library is extensively tested. However, like any complex mathematical software, Arb is virtually certain to contain bugs, so the usual precautions are advised:

- Do sanity checks. For example, check that the result satisfies an expected mathematical relation, or compute the same result in two different ways, with different settings, and with different levels of precision. Arb’s unit tests already do such checks, but they are not guaranteed to catch every possible bug, and they provide no protection against the user accidentally using the interface incorrectly.
- Compare results with other mathematical software.
- Read the source code to verify that it really does what it is supposed to do.

All bug reports are highly appreciated.

9.4 Arb example programs

See *Examples* for general information about example programs. Running:

```
make examples
```

will compile the programs and place the binaries in `build/examples`. The examples related to the Arb module are documented below.

9.4.1 pi.c

This program computes π to an accuracy of roughly n decimal digits by calling the `arb_const_pi()` function with a working precision of roughly $n \log_2(10)$ bits.

Sample output, computing π to one million digits:

```
> build/examples/pi 1000000
precision = 3321933 bits... cpu/wall(s): 0.243 0.244
virt/peak/res/peak(MB): 24.46 30.44 8.73 14.42
[3.14159265358979323846{...999959 digits...}42209010610577945815 +/- 1.38e-1000000]
```

The program prints an interval guaranteed to contain π , and where all displayed digits are correct up to an error of plus or minus one unit in the last place (see `arb_printn()`). By default, only the first and last few digits are printed. Pass 0 as a second argument to print all digits (or pass m to print $m + 1$ leading and m trailing digits, as above with the default $m = 20$).

The program can optionally compute various other constants, and can use multiple threads:

```
> build/examples/pi 1000000 -threads 4
precision = 3321933 bits... cpu/wall(s): 0.265 0.147
virt/peak/res/peak(MB): 241.95 422.15 13.33 17.54
[3.14159265358979323846{...999959 digits...}42209010610577945815 +/- 1.38e-1000000]
> build/examples/pi 1000000 -constant e
precision = 3321933 bits... cpu/wall(s): 0.09 0.09
virt/peak/res/peak(MB): 25.56 29.19 9.58 13.11
[2.71828182845904523536{...999959 digits...}01379817644769422819 +/- 1.39e-1000000]
```

9.4.2 pi_agm.c

This program implements the Brent-Salamin AGM iteration to compute π . This algorithm is not used by `arb_const_pi()` since the Chudnovsky algorithm is faster in practice.

```
> build/examples/pi_agm 100000 precision = 332197 bits... Iteration 1 / 16: error bound
0.00391 Iteration 2 / 16: error bound 2.98e-8 Iteration 3 / 16: error bound 8.67e-19 Itera-
tion 4 / 16: error bound 1.84e-40 Iteration 5 / 16: error bound 8.24e-84 Iteration 6 / 16:
error bound 8.28e-171 Iteration 7 / 16: error bound 4.18e-345 Iteration 8 / 16: error bound
5.34e-694 Iteration 9 / 16: error bound 2.18e-1392 Iteration 10 / 16: error bound 3.62e-2789
Iteration 11 / 16: error bound 5.01e-5583 Iteration 12 / 16: error bound 2.39e-11171 Iteration
13 / 16: error bound 5.46e-22348 Iteration 14 / 16: error bound 1.42e-44701 Iteration 15 /
16: error bound 4.82e-89409 Iteration 16 / 16: error bound 1.38e-178824 cpu/wall(s): 0.028
0.028 virt/peak/res/peak(MB): 19.19 19.19 7.17 7.20 [3.14159265358979323846{...99959 dig-
its...}76742080565549362465 +/- 3.91e-100000]
```

9.4.3 zeta_zeros.c

This program computes one or several consecutive zeros of the Riemann zeta function on the critical line:

```
> build/examples/zeta_zeros -n 1 -count 10 -digits 30
1 14.1347251417346937904572519836
2 21.0220396387715549926284795939
3 25.0108575801456887632137909926
4 30.4248761258595132103118975306
5 32.9350615877391896906623689641
6 37.5861781588256712572177634807
7 40.9187190121474951873981269146
8 43.3270732809149995194961221654
9 48.0051508811671597279424727494
10 49.7738324776723021819167846786
cpu/wall(s): 0.01 0.01
virt/peak/res/peak(MB): 21.28 21.28 7.29 7.29
```

Five zeros starting with the millionth:

```
> build/examples/zeta_zeros -n 1000000 -count 5 -digits 20
1000000 600269.67701244495552
1000001 600270.30109071169866
1000002 600270.74787059436613
```

(continues on next page)

(continued from previous page)

```
1000003      600271.48637367364820
1000004      600271.76148042593778
cpu/wall(s): 0.03 0.03
virt/peak/res/peak(MB): 21.41 21.41 7.41 7.41
```

The program supports the following options:

```
zeta_zeros [-n n] [-count n] [-prec n] [-digits n] [-threads n] [-platt] [-noplatt] [-
↪v] [-verbose] [-h] [-help]
```

With `-platt`, Platt's algorithm is used, which may be faster when computing many zeros of large index simultaneously.

9.4.4 bernoulli.c

This program benchmarks computing the n th Bernoulli number exactly:

```
> build/examples/bernoulli 1000000 -threads 8
cpu/wall(s): 27.227 5.836
virt/peak/res/peak(MB): 573.47 731.39 73.23 165.13
```

9.4.5 class_poly.c

This program benchmarks computing Hilbert class polynomials:

```
> build/examples/class_poly -1000004 -threads 8
cpu/wall(s): 6.932 1.478
virt/peak/res/peak(MB): 535.27 653.18 71.02 100.65
degree = 624, bits = -37823
```

9.4.6 hilbert_matrix.c

Given an input integer n , this program accurately computes the determinant of the n by n Hilbert matrix. Hilbert matrices are notoriously ill-conditioned: although the entries are close to unit magnitude, the determinant h_n decreases superexponentially (nearly as $1/4^{n^2}$) as a function of n . This program automatically doubles the working precision until the ball computed for h_n by `arb_mat_det()` does not contain zero.

Sample output:

```
$ build/examples/hilbert_matrix 200
prec=20: [ +/- 1.32e-335]
prec=40: [ +/- 1.63e-545]
prec=80: [ +/- 1.30e-933]
prec=160: [ +/- 3.62e-1926]
prec=320: [ +/- 1.81e-4129]
prec=640: [ +/- 3.84e-8838]
prec=1280: [2.955454297e-23924 +/- 8.29e-23935]
success!
cpu/wall(s): 8.494 8.513
virt/peak/res/peak(MB): 134.98 134.98 111.57 111.57
```

Called with `-eig n`, instead of computing the determinant, the program computes the smallest eigenvalue of the Hilbert matrix (in fact, it isolates all eigenvalues and prints the smallest eigenvalue):


```
$ build/examples/hilbert_matrix -eig 50
prec=20: nan
prec=40: nan
prec=80: nan
prec=160: nan
prec=320: nan
prec=640: [1.459157797e-74 +/- 2.49e-84]
success!
cpu/wall(s): 1.84 1.841
virt/peak/res/peak(MB): 33.97 33.97 10.51 10.51
```

9.4.7 keiper_li.c

Given an input integer n , this program rigorously computes numerical values of the Keiper-Li coefficients $\lambda_0, \dots, \lambda_n$. The Keiper-Li coefficients have the property that $\lambda_n > 0$ for all $n > 0$ if and only if the Riemann hypothesis is true. This program was used for the record computations described in [Joh2013] (the paper describes the algorithm in some more detail).

The program takes the following parameters:

```
keiper_li n [-prec prec] [-threads num_threads] [-out out_file]
```

The program prints the first and last few coefficients. It can optionally write all the computed data to a file. The working precision defaults to a value that should give all the coefficients to a few digits of accuracy, but can optionally be set higher (or lower). On a multicore system, using several threads results in faster execution.

Sample output:

```
> build/examples/keiper_li 1000 -threads 2
zeta: cpu/wall(s): 0.4 0.244
virt/peak/res/peak(MB): 167.98 294.69 5.09 7.43
log: cpu/wall(s): 0.03 0.038
gamma: cpu/wall(s): 0.02 0.016
binomial transform: cpu/wall(s): 0.01 0.018
0: -0.69314718055994530941723212145817656807550013436026 +/- 6.5389e-347
1: 0.023095708966121033814310247906495291621932127152051 +/- 2.0924e-345
2: 0.046172867614023335192864243096033943387066108314123 +/- 1.674e-344
3: 0.0692129735181082679304973488726010689942120263932 +/- 5.0219e-344
4: 0.092197619873060409647627872409439018065541673490213 +/- 2.0089e-343
5: 0.11510854289223549048622128109857276671349132303596 +/- 1.0044e-342
6: 0.13792766871372988290416713700341666356138966078654 +/- 6.0264e-342
7: 0.16063715965299421294040287257385366292282442046163 +/- 2.1092e-341
8: 0.18321945964338257908193931774721859848998098273432 +/- 8.4368e-341
9: 0.20565733870917046170289387421343304741236553410044 +/- 7.5931e-340
10: 0.22793393631931577436930340573684453380748385942738 +/- 7.5931e-339
991: 2.3196617961613367928373899656994682562101430813341 +/- 2.461e-11
992: 2.3203766239254884035349896518332550233162909717288 +/- 9.5363e-11
993: 2.321092061239733282811659116333262802034375592414 +/- 1.8495e-10
994: 2.3218073540188462110258826121503870112747188888893 +/- 3.5907e-10
995: 2.3225217392815185726928702951225314023773358152533 +/- 6.978e-10
996: 2.3232344485814623873333223609413703912358283071281 +/- 1.3574e-09
997: 2.3239447114886014522889542667580382034526509232475 +/- 2.6433e-09
998: 2.3246517591032700808344143240352605148856869322209 +/- 5.1524e-09
999: 2.3253548275861382119812576052060526988544993162101 +/- 1.0053e-08
1000: 2.3260531616864664574065046940832238158044982041872 +/- 3.927e-08
virt/peak/res/peak(MB): 170.18 294.69 7.51 7.51
```

9.4.8 logistic.c

This program computes the n -th iterate of the logistic map defined by $x_{n+1} = rx_n(1 - x_n)$ where r and x_0 are given. It takes the following parameters:

```
logistic n [x_0] [r] [digits]
```

The inputs x_0 , r and *digits* default to 0.5, 3.75 and 10 respectively. The computation is automatically restarted with doubled precision until the result is accurate to *digits* decimal digits.

Sample output:

```
> build/examples/logistic 10
Trying prec=64 bits...success!
cpu/wall(s): 0 0.001
x_10 = [0.6453672908 +/- 3.10e-11]

> build/examples/logistic 100
Trying prec=64 bits...ran out of accuracy at step 18
Trying prec=128 bits...ran out of accuracy at step 53
Trying prec=256 bits...success!
cpu/wall(s): 0 0
x_100 = [0.8882939923 +/- 1.60e-11]

> build/examples/logistic 10000
Trying prec=64 bits...ran out of accuracy at step 18
Trying prec=128 bits...ran out of accuracy at step 53
Trying prec=256 bits...ran out of accuracy at step 121
Trying prec=512 bits...ran out of accuracy at step 256
Trying prec=1024 bits...ran out of accuracy at step 525
Trying prec=2048 bits...ran out of accuracy at step 1063
Trying prec=4096 bits...ran out of accuracy at step 2139
Trying prec=8192 bits...ran out of accuracy at step 4288
Trying prec=16384 bits...ran out of accuracy at step 8584
Trying prec=32768 bits...success!
cpu/wall(s): 0.859 0.858
x_10000 = [0.8242048008 +/- 4.35e-11]

> build/examples/logistic 1234 0.1 3.99 30
Trying prec=64 bits...ran out of accuracy at step 0
Trying prec=128 bits...ran out of accuracy at step 10
Trying prec=256 bits...ran out of accuracy at step 76
Trying prec=512 bits...ran out of accuracy at step 205
Trying prec=1024 bits...ran out of accuracy at step 461
Trying prec=2048 bits...ran out of accuracy at step 974
Trying prec=4096 bits...success!
cpu/wall(s): 0.009 0.009
x_1234 = [0.256445391958651410579677945635 +/- 3.92e-31]
```

9.4.9 real_roots.c

This program isolates the roots of a function on the interval (a, b) (where a and b are input as double-precision literals) using the routines in the `arb_calc` module. The program takes the following arguments:

```
real_roots function a b [-refine d] [-verbose] [-maxdepth n] [-maxeval n] [-maxfound n] [-prec n]
```

The following functions (specified by an integer code) are implemented:

- 0 - $Z(x)$ (Riemann-Siegel Z-function)
- 1 - $\sin(x)$
- 2 - $\sin(x^2)$
- 3 - $\sin(1/x)$
- 4 - $\text{Ai}(x)$ (Airy function)
- 5 - $\text{Ai}'(x)$ (Airy function)
- 6 - $\text{Bi}(x)$ (Airy function)
- 7 - $\text{Bi}'(x)$ (Airy function)

The following options are available:

- `-refine d`: If provided, after isolating the roots, attempt to refine the roots to d digits of accuracy using a few bisection steps followed by Newton's method with adaptive precision, and then print them.
- `-verbose`: Print more information.
- `-maxdepth n`: Stop searching after n recursive subdivisions.
- `-maxeval n`: Stop searching after approximately n function evaluations (the actual number evaluations will be a small multiple of this).
- `-maxfound n`: Stop searching after having found n isolated roots.
- `-prec n`: Working precision to use for the root isolation.

With *function* 0, the program isolates roots of the Riemann zeta function on the critical line, and guarantees that no roots are missed (see `zeta_zeros.c` for a far more efficient way to do this):

```
> build/examples/real_roots 0 0.0 50.0 -verbose
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
found isolated root in: [14.111328125, 14.16015625]
found isolated root in: [20.99609375, 21.044921875]
found isolated root in: [25, 25.048828125]
found isolated root in: [30.419921875, 30.4443359375]
found isolated root in: [32.91015625, 32.958984375]
found isolated root in: [37.548828125, 37.59765625]
found isolated root in: [40.91796875, 40.966796875]
found isolated root in: [43.310546875, 43.3349609375]
found isolated root in: [47.998046875, 48.0224609375]
found isolated root in: [49.755859375, 49.7802734375]
-----
Found roots: 10
Subintervals possibly containing undetected roots: 0
Function evaluations: 3058
cpu/wall(s): 0.202 0.202
virt/peak/res/peak(MB): 26.12 26.14 2.76 2.76
```

Find just one root and refine it to approximately 75 digits:

```
> build/examples/real_roots 0 0.0 50.0 -maxfound 1 -refine 75
interval: [0, 50]
maxdepth = 30, maxeval = 100000, maxfound = 1, low_prec = 30
refined root (0/8):
[14.134725141734693790457251983562470270784257115699243175685567460149963429809 +/- 2.
↪57e-76]

-----

Found roots: 1
Subintervals possibly containing undetected roots: 7
Function evaluations: 761
cpu/wall(s): 0.055 0.056
virt/peak/res/peak(MB): 26.12 26.14 2.75 2.75
```

Find the first few roots of an Airy function and refine them to 50 digits each:

```
> build/examples/real_roots 4 -10 0 -refine 50
interval: [-10, 0]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
refined root (0/6):
[-9.022650853340980380158190839880089256524677535156083 +/- 4.85e-52]

refined root (1/6):
[-7.944133587120853123138280555798268532140674396972215 +/- 1.92e-52]

refined root (2/6):
[-6.786708090071758998780246384496176966053882477393494 +/- 3.84e-52]

refined root (3/6):
[-5.520559828095551059129855512931293573797214280617525 +/- 1.05e-52]

refined root (4/6):
[-4.087949444130970616636988701457391060224764699108530 +/- 2.46e-52]

refined root (5/6):
[-2.338107410459767038489197252446735440638540145672388 +/- 1.48e-52]

-----

Found roots: 6
Subintervals possibly containing undetected roots: 0
Function evaluations: 200
cpu/wall(s): 0.003 0.003
virt/peak/res/peak(MB): 26.12 26.14 2.24 2.24
```

Find roots of $\sin(x^2)$ on $(0, 100)$. The algorithm cannot isolate the root at $x = 0$ (it is at the endpoint of the interval, and in any case a root of multiplicity higher than one). The failure is reported:

```
> build/examples/real_roots 2 0 100
interval: [0, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
-----

Found roots: 3183
Subintervals possibly containing undetected roots: 1
Function evaluations: 34058
cpu/wall(s): 0.032 0.032
virt/peak/res/peak(MB): 26.32 26.37 2.04 2.04
```

This does not miss any roots:

```
> build/examples/real_roots 2 1 100
interval: [1, 100]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
-----
Found roots: 3183
Subintervals possibly containing undetected roots: 0
Function evaluations: 34039
cpu/wall(s): 0.023 0.023
virt/peak/res/peak(MB): 26.32 26.37 2.01 2.01
```

Looking for roots of $\sin(1/x)$ on $(0, 1)$, the algorithm finds many roots, but will never find all of them since there are infinitely many:

```
> build/examples/real_roots 3 0.0 1.0
interval: [0, 1]
maxdepth = 30, maxeval = 100000, maxfound = 100000, low_prec = 30
-----
Found roots: 10198
Subintervals possibly containing undetected roots: 24695
Function evaluations: 202587
cpu/wall(s): 0.171 0.171
virt/peak/res/peak(MB): 28.39 30.38 4.05 4.05
```

Remark: the program always computes rigorous containing intervals for the roots, but the accuracy after refinement could be less than d digits.

9.4.10 poly_roots.c

This program finds the complex roots of an integer polynomial by calling `arb_fmpz_poly_complex_roots()`, which in turn calls `acb_poly_find_roots()` with increasing precision until the roots certainly have been isolated. The program takes the following arguments:

```
poly_roots [-refine d] [-print d] <poly>
```

Isolates all the complex roots of a polynomial with integer coefficients.

If `-refine d` is passed, the roots are refined to a relative tolerance better than $10^{-(d)}$. By default, the roots are only computed to sufficient accuracy to isolate them. The refinement is not currently done efficiently.

If `-print d` is passed, the computed roots are printed to d decimals. By default, the roots are not printed.

The polynomial can be specified by passing the following as `<poly>`:

a <n>	Easy polynomial $1 + 2x + \dots + (n+1)x^n$
t <n>	Chebyshev polynomial T_n
u <n>	Chebyshev polynomial U_n
p <n>	Legendre polynomial P_n
c <n>	Cyclotomic polynomial Φ_n
s <n>	Swinerton-Dyer polynomial S_n
b <n>	Bernoulli polynomial B_n
w <n>	Wilkinson polynomial W_n
e <n>	Taylor series of $\exp(x)$ truncated to degree n
m <n> <m>	The Mignotte-like polynomial $x^n + (100x+1)^m$, $n > m$

(continues on next page)

(continued from previous page)

```

coeffs <c0 c1 ... cn>          c0 + c1 x + ... + cn x^n

Concatenate to multiply polynomials, e.g.: p 5 t 6 coeffs 1 2 3
for P_5(x)*T_6(x)*(1+2x+3x^2)

```

This finds the roots of the Wilkinson polynomial with roots at the positive integers 1, 2, ..., 100:

```

> build/examples/poly_roots -print 15 w 100
computing squarefree factorization...
cpu/wall(s): 0.001 0.001
roots with multiplicity 1
searching for 100 roots, 100 deflated
prec=32: 0 isolated roots | cpu/wall(s): 0.098 0.098
prec=64: 0 isolated roots | cpu/wall(s): 0.247 0.247
prec=128: 0 isolated roots | cpu/wall(s): 0.498 0.497
prec=256: 0 isolated roots | cpu/wall(s): 0.713 0.713
prec=512: 100 isolated roots | cpu/wall(s): 0.104 0.105
done!
[1.000000000000000 +/- 3e-20]
[2.000000000000000 +/- 3e-19]
[3.000000000000000 +/- 1e-19]
[4.000000000000000 +/- 1e-19]
[5.000000000000000 +/- 1e-19]
...
[96.0000000000000 +/- 1e-17]
[97.0000000000000 +/- 1e-17]
[98.0000000000000 +/- 3e-17]
[99.0000000000000 +/- 3e-17]
[100.0000000000000 +/- 3e-17]
cpu/wall(s): 1.664 1.664

```

This finds the roots of a Bernoulli polynomial which has both real and complex roots:

```

> build/examples/poly_roots -refine 100 -print 20 b 16
computing squarefree factorization...
cpu/wall(s): 0.001 0
roots with multiplicity 1
searching for 16 roots, 16 deflated
prec=32: 16 isolated roots | cpu/wall(s): 0.006 0.006
prec=64: 16 isolated roots | cpu/wall(s): 0.001 0.001
prec=128: 16 isolated roots | cpu/wall(s): 0.001 0.001
prec=256: 16 isolated roots | cpu/wall(s): 0.001 0.002
prec=512: 16 isolated roots | cpu/wall(s): 0.002 0.001
done!
[-0.94308706466055783383 +/- 2.02e-21]
[-0.75534059252067985752 +/- 2.70e-21]
[-0.24999757119077421009 +/- 4.27e-21]
[0.24999757152512726002 +/- 4.43e-21]
[0.75000242847487273998 +/- 4.43e-21]
[1.2499975711907742101 +/- 1.43e-20]
[1.7553405925206798575 +/- 1.74e-20]
[1.9430870646605578338 +/- 3.21e-20]
[-0.99509334829256233279 +/- 9.42e-22] + [0.44547958157103608805 +/- 3.59e-21]*I
[-0.99509334829256233279 +/- 9.42e-22] + [-0.44547958157103608805 +/- 3.59e-21]*I
[1.9950933482925623328 +/- 1.10e-20] + [0.44547958157103608805 +/- 3.59e-21]*I
[1.9950933482925623328 +/- 1.10e-20] + [-0.44547958157103608805 +/- 3.59e-21]*I

```

(continues on next page)

(continued from previous page)

```

[-0.92177327714429290564 +/- 4.68e-21] + [-1.0954360955079385542 +/- 1.71e-21]*I
[-0.92177327714429290564 +/- 4.68e-21] + [1.0954360955079385542 +/- 1.71e-21]*I
[1.9217732771442929056 +/- 3.54e-20] + [1.0954360955079385542 +/- 1.71e-21]*I
[1.9217732771442929056 +/- 3.54e-20] + [-1.0954360955079385542 +/- 1.71e-21]*I
cpu/wall(s): 0.011 0.012

```

Roots are automatically separated by multiplicity by performing an initial squarefree factorization:

```

> build/examples/poly_roots -print 5 p 5 p 5 t 7 coeffs 1 5 10 10 5 1
computing squarefree factorization...
cpu/wall(s): 0 0
roots with multiplicity 1
searching for 6 roots, 3 deflated
prec=32: 3 isolated roots | cpu/wall(s): 0 0.001
done!
[-0.97493 +/- 2.10e-6]
[-0.78183 +/- 1.49e-6]
[-0.43388 +/- 3.75e-6]
[0.43388 +/- 3.75e-6]
[0.78183 +/- 1.49e-6]
[0.97493 +/- 2.10e-6]
roots with multiplicity 2
searching for 4 roots, 2 deflated
prec=32: 2 isolated roots | cpu/wall(s): 0 0
done!
[-0.90618 +/- 1.56e-7]
[-0.53847 +/- 6.91e-7]
[0.53847 +/- 6.91e-7]
[0.90618 +/- 1.56e-7]
roots with multiplicity 3
searching for 1 roots, 0 deflated
prec=32: 0 isolated roots | cpu/wall(s): 0 0
done!
0
roots with multiplicity 5
searching for 1 roots, 1 deflated
prec=32: 1 isolated roots | cpu/wall(s): 0 0
done!
-1.0000
cpu/wall(s): 0 0.001

```

9.4.11 zeta_zeros.c

This program finds the imaginary parts of consecutive nontrivial zeros of the Riemann zeta function by calling either `acb_dirichlet_hardy_z_zeros()` or `acb_dirichlet_platt_local_hardy_z_zeros()` depending on the height of the zeros and the number of zeros requested. The program takes the following arguments:

```

zeta_zeros [-n n] [-count n] [-prec n] [-threads n] [-platt] [-noplatt] [-v] [-
↪ verbose] [-h] [-help]

> build/examples/zeta_zeros -n 1048449114 -count 2
1048449114      [388858886.0022851217767970582 +/- 7.46e-20]
1048449115      [388858886.0023936897027167201 +/- 7.59e-20]
cpu/wall(s): 0.255 0.255

```

(continues on next page)

(continued from previous page)

```
virt/peak/res/peak(MB): 26.77 26.77 7.88 7.88
```

9.4.12 complex_plot.c

This program plots one of the predefined functions over a complex interval $[x_a, x_b] + [y_a, y_b]i$ using domain coloring, at a resolution of xn times yn pixels.

The program takes the parameters:

```
complex_plot [-range xa xb ya yb] [-size xn yn] [-color n] [-threads n] <func>
```

Defaults parameters are $[-10, 10] + [-10, 10]i$ and $xn = yn = 512$.

A color function can be selected with `-color`. Valid options are 0 (phase=hue, magnitude=brightness) and 1 (phase only, white-gold-black-blue-white counterclockwise).

The output is written to `arbplot.ppm`. If you have ImageMagick, run `convert arbplot.ppm arbplot.png` to get a PNG.

Function codes `<func>` are:

- `gamma` - Gamma function
- `digamma` - Digamma function
- `lgamma` - Logarithmic gamma function
- `zeta` - Riemann zeta function
- `erf` - Error function
- `ai` - Airy function Ai
- `bi` - Airy function Bi
- `besselj` - Bessel function J_0
- `bessely` - Bessel function Y_0
- `besseli` - Bessel function I_0
- `besselk` - Bessel function K_0
- `modj` - Modular j -function
- `modeta` - Dedekind eta function
- `barnesg` - Barnes G -function
- `agm` - Arithmetic geometric mean

The function is just sampled at point values; no attempt is made to resolve small features by adaptive subsampling.

For example, the following plots the Riemann zeta function around a portion of the critical strip with imaginary part between 100 and 140:

```
> build/examples/complex_plot zeta -range -10 10 100 140 -size 256 512
```

For parallel computation on a multicore system, use `-threads n`.

9.4.13 lvalue.c

This program evaluates Dirichlet L-functions. It takes the following input:

```
> build/examples/lvalue
lvalue [-character q n] [-re a] [-im b] [-prec p] [-z] [-deflate] [-len l]

Print value of Dirichlet L-function at  $s = a+bi$ .
Default  $a = 0.5$ ,  $b = 0$ ,  $p = 53$ ,  $(q, n) = (1, 0)$  (Riemann zeta)
[-z]          - compute  $Z(s)$  instead of  $L(s)$ 
[-deflate]    - remove singular term at  $s = 1$ 
[-len l]      - compute  $l$  terms in Taylor series at  $s$ 
```

Evaluating the Riemann zeta function and the Dirichlet beta function at $s = 2$:

```
> build/examples/lvalue -re 2 -prec 128
L(s) = [1.64493406684822643647241516664602518922 +/- 4.37e-39]
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.05 2.05

> build/examples/lvalue -character 4 3 -re 2 -prec 128
L(s) = [0.91596559417721901505460351493238411077 +/- 7.86e-39]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.86 26.88 2.31 2.31
```

Evaluating the L-function for character number 101 modulo 1009 at $s = 1/2$ and $s = 1$:

```
> build/examples/lvalue -character 1009 101
L(s) = [-0.459256562383872 +/- 5.24e-16] + [1.346937111206009 +/- 3.03e-16]*I
cpu/wall(s): 0.012 0.012
virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30

> build/examples/lvalue -character 1009 101 -re 1
L(s) = [0.657952586112728 +/- 6.02e-16] + [1.004145273214022 +/- 3.10e-16]*I
cpu/wall(s): 0.017 0.018
virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30
```

Computing the first few coefficients in the Laurent series of the Riemann zeta function at $s = 1$:

```
> build/examples/lvalue -re 1 -deflate -len 8
L(s) = [0.577215664901532861 +/- 5.29e-19]
L'(s) = [0.072815845483676725 +/- 2.68e-19]
[x^2] L(s+x) = [-0.004845181596436159 +/- 3.87e-19]
[x^3] L(s+x) = [-0.000342305736717224 +/- 4.20e-19]
[x^4] L(s+x) = [9.6890419394471e-5 +/- 2.40e-19]
[x^5] L(s+x) = [-6.6110318108422e-6 +/- 4.51e-20]
[x^6] L(s+x) = [-3.316240908753e-7 +/- 3.85e-20]
[x^7] L(s+x) = [1.0462094584479e-7 +/- 7.78e-21]
cpu/wall(s): 0.003 0.004
virt/peak/res/peak(MB): 26.86 26.88 2.30 2.30
```

Evaluating the Riemann zeta function near the first nontrivial root:

```
> build/examples/lvalue -re 0.5 -im 14.134725
L(s) = [1.76743e-8 +/- 1.93e-14] + [-1.110203e-7 +/- 2.84e-14]*I
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.31 2.31
```

(continues on next page)

(continued from previous page)

```

> build/examples/lvalue -z -re 14.134725 -prec 200
Z(s) = [-1.12418349839417533300111494358128257497862927935658e-7 +/- 4.62e-58]
cpu/wall(s): 0.001 0.001
virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57

> build/examples/lvalue -z -re 14.134725 -len 4
Z(s) = [-1.124184e-7 +/- 7.00e-14]
Z'(s) = [0.793160414884 +/- 4.09e-13]
[x^2] Z(s+x) = [0.065164586492 +/- 5.39e-13]
[x^3] Z(s+x) = [-0.020707762705 +/- 5.37e-13]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.86 26.88 2.57 2.57

```

9.4.14 lcentral.c

This program computes the central value $L(1/2)$ for each Dirichlet L-function character modulo q for each q in the range $qmin$ to $qmax$. Usage:

```

> build/examples/lcentral
Computes central values (s = 0.5) of Dirichlet L-functions.

usage: build/examples/lcentral [--quiet] [--check] [--prec <bits>] qmin qmax

```

The first few values:

```

> build/examples/lcentral 1 8
3,2: [0.48086755769682862618122006324 +/- 7.35e-30]
4,3: [0.66769145718960917665869092930 +/- 1.62e-30]
5,2: [0.76374788011728687822451215264 +/- 2.32e-30] + [0.
↪21696476751886069363858659310 +/- 3.06e-30]*I
5,4: [0.23175094750401575588338366176 +/- 2.21e-30]
5,3: [0.76374788011728687822451215264 +/- 2.32e-30] + [-0.
↪21696476751886069363858659310 +/- 3.06e-30]*I
7,3: [0.71394334376831949285993820742 +/- 1.21e-30] + [0.
↪47490218277139938263745243935 +/- 4.52e-30]*I
7,2: [0.31008936259836766059195052534 +/- 5.29e-30] + [-0.
↪07264193137017790524562171245 +/- 5.48e-30]*I
7,6: [1.14658566690370833367712697646 +/- 1.95e-30]
7,4: [0.31008936259836766059195052534 +/- 5.29e-30] + [0.
↪07264193137017790524562171245 +/- 5.48e-30]*I
7,5: [0.71394334376831949285993820742 +/- 1.21e-30] + [-0.
↪47490218277139938263745243935 +/- 4.52e-30]*I
8,5: [0.37369171291254730738158695002 +/- 4.01e-30]
8,3: [1.10042140952554837756713576997 +/- 3.37e-30]
cpu/wall(s): 0.002 0.003
virt/peak/res/peak(MB): 26.32 26.34 2.35 2.35

```

Testing a large q :

```

> build/examples/lcentral --quiet --check --prec 256 100000 100000
cpu/wall(s): 1.668 1.667
virt/peak/res/peak(MB): 35.67 46.66 11.67 22.61

```

It is conjectured that the central value never vanishes. Running with `--check` verifies that the interval certainly is nonzero. This can fail with insufficient precision:

```
> build/examples/lcentral --check --prec 15 100000 100000
100000,71877: [0.1 +/- 0.0772] + [ +/- 0.136]*I
100000,90629: [2e+0 +/- 0.106] + [ +/- 0.920]*I
100000,28133: [ +/- 0.811] + [-2e+0 +/- 0.501]*I
100000,3141: [0.8 +/- 0.0407] + [-0.1 +/- 0.0243]*I
100000,53189: [4.0 +/- 0.0826] + [ +/- 0.107]*I
100000,53253: [1.9 +/- 0.0855] + [-3.9 +/- 0.0681]*I
Value could be zero!
100000,53381: [ +/- 0.0329] + [ +/- 0.0413]*I
Aborted
```

9.4.15 integrals.c

This program computes integrals using `acb_calc_integrate()`. Invoking the program without parameters shows usage:

```
> build/examples/integrals
Compute integrals using acb_calc_integrate.
Usage: integrals -i n [-prec p] [-tol eps] [-twice] [...]
```

-i n - compute integral n (0 <= n <= 23), or "-i all"

-prec p - precision in bits (default p = 64)

-goal p - approximate relative accuracy goal (default p)

-tol eps - approximate absolute error goal (default 2^{-p})

-twice - run twice (to see overhead of computing nodes)

-heap - use heap for subinterval queue

-verbose - show information

-verbose2 - show more information

-deg n - use quadrature degree up to n

-eval n - limit number of function evaluations to n

-depth n - limit subinterval queue size to n

-threads n - use parallel computation with n threads

Implemented integrals:

I0 = $\int_0^{100} \sin(x) \, dx$

I1 = $4 \int_0^1 \frac{1}{(1+x^2)} \, dx$

I2 = $2 \int_0^{\infty} \frac{1}{(1+x^2)} \, dx$ (using domain truncation)

I3 = $4 \int_0^1 \sqrt{1-x^2} \, dx$

I4 = $\int_0^8 \sin(x+\exp(x)) \, dx$

I5 = $\int_1^{101} \text{floor}(x) \, dx$

I6 = $\int_0^1 |x^4+10x^3+19x^2-6x-6| \exp(x) \, dx$

I7 = $\frac{1}{2} \pi i \int \text{zeta}(s) \, ds$ (closed path around $s = 1$)

I8 = $\int_0^1 \sin(1/x) \, dx$ (slow convergence, use -heap and/or -tol)

I9 = $\int_0^1 x \sin(1/x) \, dx$ (slow convergence, use -heap and/or -tol)

I10 = $\int_0^{10000} x^{1000} \exp(-x) \, dx$

I11 = $\int_1^{\{1+1000i\}} \gamma(x) \, dx$

I12 = $\int_{-10}^{\{10\}} \sin(x) + \exp(-200-x^2) \, dx$

I13 = $\int_{-1020}^{\{-1010\}} \exp(x) \, dx$ (use -tol 0 for relative error)

I14 = $\int_0^{\infty} \exp(-x^2) \, dx$ (using domain truncation)

I15 = $\int_0^1 \text{sech}(10(x-0.2))^2 + \text{sech}(100(x-0.4))^4 + \text{sech}(1000(x-0.6))^6 \, dx$

I16 = $\int_0^8 (\exp(x)-\text{floor}(\exp(x))) \sin(x+\exp(x)) \, dx$ (use higher -eval)

I17 = $\int_0^{\infty} \text{sech}(x) \, dx$ (using domain truncation)

I18 = $\int_0^{\infty} \text{sech}^3(x) \, dx$ (using domain truncation)

I19 = $\int_0^1 -\log(x)/(1+x) \, dx$ (using domain truncation)

I20 = $\int_0^{\infty} x \exp(-x)/(1+\exp(-x)) \, dx$ (using domain truncation)

I21 = $\int_C \wp(x)/x^{(11)} \, dx$ (contour for 10th Laurent coefficient of Weierstrass p-

(continues on next page)

(continued from previous page)

```

↪function)
I22 = N(1000) = count zeros with  $0 < t \leq 1000$  of  $\zeta(s)$  using argument principle
I23 =  $\int_0^{1000} W_0(x) dx$ 
I24 =  $\int_0^\pi \max(\sin(x), \cos(x)) dx$ 
I25 =  $\int_{-1}^1 \operatorname{erf}(x/\sqrt{0.0002}) * 0.5 + 1.5 * \exp(-x) dx$ 
I26 =  $\int_{-10}^{10} \operatorname{Ai}(x) dx$ 
I27 =  $\int_0^{10} (x - \lfloor x \rfloor - 1/2) \max(\sin(x), \cos(x)) dx$ 
I28 =  $\int_{-1-i}^{-1+i} \sqrt{x} dx$ 
I29 =  $\int_0^\infty \exp(-x^2 + ix) dx$  (using domain truncation)
I30 =  $\int_0^\infty \exp(-x) \operatorname{Ai}(-x) dx$  (using domain truncation)
I31 =  $\int_0^\pi x \sin(x) / (1 + \cos(x)^2) dx$ 

```

A few examples:

```

build/examples/integrals -i 4
I4 =  $\int_0^8 \sin(x + \exp(x)) dx \dots$ 
cpu/wall(s): 0.02 0.02
I4 = [0.34740017265725 +/- 3.95e-15]

> build/examples/integrals -i 3 -prec 333 -tol 1e-80
I3 =  $4 \int_0^1 \sqrt{1-x^2} dx \dots$ 
cpu/wall(s): 0.024 0.024
I3 = [3.
↪141592653589793238462643383279502884197169399375105820974944592307816406286209 +/-
↪4.24e-79]

> build/examples/integrals -i 9 -heap
I9 =  $\int_0^1 x \sin(1/x) dx$  (slow convergence, use -heap and/or -tol) ...
cpu/wall(s): 0.019 0.018
I9 = [0.3785300 +/- 3.17e-8]

```

9.4.16 fpwrap.c

This program demonstrates calling the floating-point wrapper:

```

> build/examples/fpwrap
zeta(2) = 1.644934066848226
zeta(0.5 + 123i) = 0.006252861175594465 + 0.08206030514520983i

```

9.4.17 functions_benchmark.c

This program benchmarks performance of some standard functions.

9.5 mag.h – fixed-precision unsigned floating-point numbers for bounds

The `mag_t` type holds an unsigned floating-point number with a fixed-precision mantissa (30 bits) and an arbitrary-precision exponent (represented as an `fmpz_t`), suited for representing magnitude bounds. The special values zero and positive infinity are supported, but not NaN.

Operations that involve rounding will always produce a valid upper bound, or a lower bound if the function name has the suffix *lower*. For performance reasons, no attempt is made to compute the best possible bounds: in general, a bound may be several ulps larger/smaller than the optimal bound. Some

functions such as `mag_set()` and `mag_mul_2exp_si()` are always exact and therefore do not require separate *lower* versions.

A common mistake is to forget computing a lower bound for the argument of a decreasing function that is meant to be bounded from above, or vice versa. For example, to compute an upper bound for $(x+1)/(y+1)$, the parameter x should initially be an upper bound while y should be a lower bound, and one should do:

```
mag_add_ui(tmp1, x, 1);
mag_add_ui_lower(tmp2, y, 1);
mag_div(res, tmp1, tmp2);
```

For a lower bound of the same expression, x should be a lower bound while y should be an upper bound, and one should do:

```
mag_add_ui_lower(tmp1, x, 1);
mag_add_ui(tmp2, y, 1);
mag_div_lower(res, tmp1, tmp2);
```

Applications requiring floating-point arithmetic with more flexibility (such as correct rounding, or higher precision) should use the `arf_t` type instead. For calculations where a complex alternation between upper and lower bounds is necessary, it may be cleaner to use `arb_t` arithmetic and convert to a `mag_t` bound only in the end.

9.5.1 Types, macros and constants

type `mag_struct`

A `mag_struct` holds a mantissa and an exponent. Special values are encoded by the mantissa being set to zero.

type `mag_t`

A `mag_t` is defined as an array of length one of type `mag_struct`, permitting a `mag_t` to be passed by reference.

9.5.2 Memory management

void `mag_init(mag_t x)`

Initializes the variable x for use. Its value is set to zero.

void `mag_clear(mag_t x)`

Clears the variable x , freeing or recycling its allocated memory.

void `mag_swap(mag_t x, mag_t y)`

Swaps x and y efficiently.

`mag_ptr` `_mag_vec_init(slong n)`

Allocates a vector of length n . All entries are set to zero.

void `_mag_vec_clear(mag_ptr v, slong n)`

Clears a vector of length n .

`slong` `mag_allocated_bytes(const mag_t x)`

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(mag_struct)` to get the size of the object as a whole.

9.5.3 Special values

void **mag_zero**(*mag_t* res)

Sets *res* to zero.

void **mag_one**(*mag_t* res)

Sets *res* to one.

void **mag_inf**(*mag_t* res)

Sets *res* to positive infinity.

int **mag_is_special**(const *mag_t* x)

Returns nonzero iff *x* is zero or positive infinity.

int **mag_is_zero**(const *mag_t* x)

Returns nonzero iff *x* is zero.

int **mag_is_inf**(const *mag_t* x)

Returns nonzero iff *x* is positive infinity.

int **mag_is_finite**(const *mag_t* x)

Returns nonzero iff *x* is not positive infinity (since there is no NaN value, this function is exactly the logical negation of *mag_is_inf()*).

9.5.4 Assignment and conversions

void **mag_init_set**(*mag_t* res, const *mag_t* x)

Initializes *res* and sets it to the value of *x*. This operation is always exact.

void **mag_set**(*mag_t* res, const *mag_t* x)

Sets *res* to the value of *x*. This operation is always exact.

void **mag_set_d**(*mag_t* res, double x)

void **mag_set_ui**(*mag_t* res, *ulong* x)

void **mag_set_fmpz**(*mag_t* res, const *fmpz_t* x)

Sets *res* to an upper bound for $|x|$. The operation may be inexact even if *x* is exactly representable.

void **mag_set_d_lower**(*mag_t* res, double x)

void **mag_set_ui_lower**(*mag_t* res, *ulong* x)

void **mag_set_fmpz_lower**(*mag_t* res, const *fmpz_t* x)

Sets *res* to a lower bound for $|x|$. The operation may be inexact even if *x* is exactly representable.

void **mag_set_d_2exp_fmpz**(*mag_t* res, double x, const *fmpz_t* y)

void **mag_set_fmpz_2exp_fmpz**(*mag_t* res, const *fmpz_t* x, const *fmpz_t* y)

void **mag_set_ui_2exp_si**(*mag_t* res, *ulong* x, *slong* y)

Sets *res* to an upper bound for $|x| \cdot 2^y$.

void **mag_set_d_2exp_fmpz_lower**(*mag_t* res, double x, const *fmpz_t* y)

void **mag_set_fmpz_2exp_fmpz_lower**(*mag_t* res, const *fmpz_t* x, const *fmpz_t* y)

Sets *res* to a lower bound for $|x| \cdot 2^y$.

double **mag_get_d**(const *mag_t* x)

Returns a *double* giving an upper bound for *x*.

double **mag_get_d_log2_approx**(const *mag_t* x)

Returns a *double* approximating $\log_2(x)$, suitable for estimating magnitudes (warning: not a rigorous bound). The value is clamped between *COEFF_MIN* and *COEFF_MAX*.

void **mag_get_fmpq**(*fmpq_t* res, const *mag_t* x)

void **mag_get_fmpz**(*fmpz_t* res, const *mag_t* x)

void **mag_get_fmpz_lower**(*fmpz_t* res, const *mag_t* x)

Sets *res*, respectively, to the exact rational number represented by *x*, the integer exactly representing the ceiling function of *x*, or the integer exactly representing the floor function of *x*.

These functions are unsafe: the user must check in advance that *x* is of reasonable magnitude. If *x* is infinite or has a bignum exponent, an abort will be raised. If the exponent otherwise is too large or too small, the available memory could be exhausted resulting in undefined behavior.

9.5.5 Comparisons

int **mag_equal**(const *mag_t* x, const *mag_t* y)

Returns nonzero iff *x* and *y* have the same value.

int **mag_cmp**(const *mag_t* x, const *mag_t* y)

Returns negative, zero, or positive, depending on whether *x* is smaller, equal, or larger than *y*.

int **mag_cmp_2exp_si**(const *mag_t* x, *slong* y)

Returns negative, zero, or positive, depending on whether *x* is smaller, equal, or larger than 2^y .

void **mag_min**(*mag_t* res, const *mag_t* x, const *mag_t* y)

void **mag_max**(*mag_t* res, const *mag_t* x, const *mag_t* y)

Sets *res* respectively to the smaller or the larger of *x* and *y*.

9.5.6 Input and output

void **mag_print**(const *mag_t* x)

Prints *x* to standard output.

void **mag_fprint**(FILE *file, const *mag_t* x)

Prints *x* to the stream *file*.

char ***mag_dump_str**(const *mag_t* x)

Allocates a string and writes a binary representation of *x* to it that can be read by *mag_load_str()*. The returned string needs to be deallocated with *flint_free*.

int **mag_load_str**(*mag_t* x, const char *str)

Parses *str* into *x*. Returns a nonzero value if *str* is not formatted correctly.

int **mag_dump_file**(FILE *stream, const *mag_t* x)

Writes a binary representation of *x* to *stream* that can be read by *mag_load_file()*. Returns a nonzero value if the data could not be written.

int **mag_load_file**(*mag_t* x, FILE *stream)

Reads *x* from *stream*. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with *mag_dump_file()* make sure to insert a whitespace to separate consecutive values.

9.5.7 Random generation

void **mag_randtest**(*mag_t* res, *flint_rand_t* state, *slong* expbits)
 Sets *res* to a random finite value, with an exponent up to *expbits* bits large.

void **mag_randtest_special**(*mag_t* res, *flint_rand_t* state, *slong* expbits)
 Like *mag_randtest()*, but also sometimes sets *res* to infinity.

9.5.8 Arithmetic

void **mag_add**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Adds *x* and *y*.

void **mag_add_ui**(*mag_t* res, const *mag_t* x, *ulong* y)
 Sets *res* to an upper bound for $x + y$.

void **mag_add_lower**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Sets *res* to a lower bound for $x + y$.

void **mag_add_ui_lower**(*mag_t* res, const *mag_t* x, *ulong* y)
 Sets *res* to a lower bound for $x + y$.

void **mag_add_2exp_fmpz**(*mag_t* res, const *mag_t* x, const *fmpz_t* e)
 Sets *res* to an upper bound for $x + 2^e$.

void **mag_add_ui_2exp_si**(*mag_t* res, const *mag_t* x, *ulong* y, *slong* e)
 Sets *res* to an upper bound for $x + y2^e$.

void **mag_sub**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Sets *res* to an upper bound for $\max(x - y, 0)$.

void **mag_sub_lower**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Sets *res* to a lower bound for $\max(x - y, 0)$.

void **mag_mul_2exp_si**(*mag_t* res, const *mag_t* x, *slong* y)
 Sets *res* to $x \cdot 2^y$. This operation is exact.

void **mag_mul**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Multiplies *x* and *y*.

void **mag_mul_ui**(*mag_t* res, const *mag_t* x, *ulong* y)
 Sets *res* to an upper bound for xy .

void **mag_mul_fmpz**(*mag_t* res, const *mag_t* x, const *fmpz_t* y)
 Sets *res* to an upper bound for xy .

void **mag_mul_lower**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Sets *res* to a lower bound for xy .

void **mag_mul_ui_lower**(*mag_t* res, const *mag_t* x, *ulong* y)
 Sets *res* to a lower bound for xy .

void **mag_mul_fmpz_lower**(*mag_t* res, const *mag_t* x, const *fmpz_t* y)
 Sets *res* to a lower bound for xy .

void **mag_addmul**(*mag_t* z, const *mag_t* x, const *mag_t* y)
 Sets *z* to an upper bound for $z + xy$.

void **mag_div**(*mag_t* res, const *mag_t* x, const *mag_t* y)
 Sets *res* to an upper bound for x/y .

void **mag_div_ui**(*mag_t* res, const *mag_t* x, *ulong* y)
 Sets *res* to an upper bound for x/y .

void **mag_div_fmpz**(*mag_t* res, const *mag_t* x, const *fmpz_t* y)
 Sets *res* to an upper bound for x/y .

void **mag_div_lower**(*mag_t* res, const *mag_t* x, const *mag_t* y)

Sets *res* to a lower bound for x/y .

void **mag_inv**(*mag_t* res, const *mag_t* x)

Sets *res* to an upper bound for $1/x$.

void **mag_inv_lower**(*mag_t* res, const *mag_t* x)

Sets *res* to a lower bound for $1/x$.

9.5.9 Fast, unsafe arithmetic

The following methods assume that all inputs are finite and that all exponents (in all inputs as well as the final result) fit as *fmpz* inline values. They also assume that the output variables do not have promoted exponents, as they will be overwritten directly (thus leaking memory).

void **mag_fast_init_set**(*mag_t* x, const *mag_t* y)

Initialises *x* and sets it to the value of *y*.

void **mag_fast_zero**(*mag_t* res)

Sets *res* to zero.

int **mag_fast_is_zero**(const *mag_t* x)

Returns nonzero iff *x* is zero.

void **mag_fast_mul**(*mag_t* res, const *mag_t* x, const *mag_t* y)

Sets *res* to an upper bound for xy .

void **mag_fast_addmul**(*mag_t* z, const *mag_t* x, const *mag_t* y)

Sets *z* to an upper bound for $z + xy$.

void **mag_fast_add_2exp_si**(*mag_t* res, const *mag_t* x, *slong* e)

Sets *res* to an upper bound for $x + 2^e$.

void **mag_fast_mul_2exp_si**(*mag_t* res, const *mag_t* x, *slong* e)

Sets *res* to an upper bound for $x2^e$.

9.5.10 Powers and logarithms

void **mag_pow_ui**(*mag_t* res, const *mag_t* x, *ulong* e)

void **mag_pow_fmpz**(*mag_t* res, const *mag_t* x, const *fmpz_t* e)

Sets *res* to an upper bound for x^e .

void **mag_pow_ui_lower**(*mag_t* res, const *mag_t* x, *ulong* e)

void **mag_pow_fmpz_lower**(*mag_t* res, const *mag_t* x, const *fmpz_t* e)

Sets *res* to a lower bound for x^e .

void **mag_sqrt**(*mag_t* res, const *mag_t* x)

Sets *res* to an upper bound for $\sqrt{x}$.

void **mag_sqrt_lower**(*mag_t* res, const *mag_t* x)

Sets *res* to a lower bound for $\sqrt{x}$.

void **mag_rsqrt**(*mag_t* res, const *mag_t* x)

Sets *res* to an upper bound for $1/\sqrt{x}$.

void **mag_rsqrt_lower**(*mag_t* res, const *mag_t* x)

Sets *res* to a lower bound for $1/\sqrt{x}$.

```

void mag_hypot(mag_t res, const mag_t x, const mag_t y)
    Sets res to an upper bound for  $\sqrt{x^2 + y^2}$ .

void mag_root(mag_t res, const mag_t x, ulong n)
    Sets res to an upper bound for  $x^{1/n}$ .

void mag_log(mag_t res, const mag_t x)
    Sets res to an upper bound for  $\log(\max(1, x))$ .

void mag_log_lower(mag_t res, const mag_t x)
    Sets res to a lower bound for  $\log(\max(1, x))$ .

void mag_neg_log(mag_t res, const mag_t x)
    Sets res to an upper bound for  $-\log(\min(1, x))$ , i.e. an upper bound for  $|\log(x)|$  for  $x \leq 1$ .

void mag_neg_log_lower(mag_t res, const mag_t x)
    Sets res to a lower bound for  $-\log(\min(1, x))$ , i.e. a lower bound for  $|\log(x)|$  for  $x \leq 1$ .

void mag_log_ui(mag_t res, ulong n)
    Sets res to an upper bound for  $\log(n)$ .

void mag_log1p(mag_t res, const mag_t x)
    Sets res to an upper bound for  $\log(1 + x)$ . The bound is computed accurately for small  $x$ .

void mag_exp(mag_t res, const mag_t x)
    Sets res to an upper bound for  $\exp(x)$ .

void mag_exp_lower(mag_t res, const mag_t x)
    Sets res to a lower bound for  $\exp(x)$ .

void mag_expinv(mag_t res, const mag_t x)
    Sets res to an upper bound for  $\exp(-x)$ .

void mag_expinv_lower(mag_t res, const mag_t x)
    Sets res to a lower bound for  $\exp(-x)$ .

void mag_expm1(mag_t res, const mag_t x)
    Sets res to an upper bound for  $\exp(x) - 1$ . The bound is computed accurately for small  $x$ .

void mag_exp_tail(mag_t res, const mag_t x, ulong N)
    Sets res to an upper bound for  $\sum_{k=N}^{\infty} x^k/k!$ .

void mag_binpow_uiui(mag_t res, ulong m, ulong n)
    Sets res to an upper bound for  $(1 + 1/m)^n$ .

void mag_geom_series(mag_t res, const mag_t x, ulong N)
    Sets res to an upper bound for  $\sum_{k=N}^{\infty} x^k$ .

```

9.5.11 Special functions

```

void mag_const_pi(mag_t res)

void mag_const_pi_lower(mag_t res)
    Sets res to an upper (respectively lower) bound for  $\pi$ .

void mag_atan(mag_t res, const mag_t x)

void mag_atan_lower(mag_t res, const mag_t x)
    Sets res to an upper (respectively lower) bound for  $\operatorname{atan}(x)$ .

```

void **mag_cosh**(*mag_t* res, const *mag_t* x)

void **mag_cosh_lower**(*mag_t* res, const *mag_t* x)

void **mag_sinh**(*mag_t* res, const *mag_t* x)

void **mag_sinh_lower**(*mag_t* res, const *mag_t* x)

Sets *res* to an upper or lower bound for $\cosh(x)$ or $\sinh(x)$.

void **mag_fac_ui**(*mag_t* res, *ulong* n)

Sets *res* to an upper bound for $n!$.

void **mag_rfac_ui**(*mag_t* res, *ulong* n)

Sets *res* to an upper bound for $1/n!$.

void **mag_bin_uiui**(*mag_t* res, *ulong* n, *ulong* k)

Sets *res* to an upper bound for the binomial coefficient $\binom{n}{k}$.

void **mag_bernoulli_div_fac_ui**(*mag_t* res, *ulong* n)

Sets *res* to an upper bound for $|B_n|/n!$ where B_n denotes a Bernoulli number.

void **mag_polylog_tail**(*mag_t* res, const *mag_t* z, *slong* s, *ulong* d, *ulong* N)

Sets *res* to an upper bound for

$$\sum_{k=N}^{\infty} \frac{z^k \log^d(k)}{k^s}.$$

The bounding strategy is described in *Algorithms for polylogarithms*. Note: in applications where s in this formula may be real or complex, the user can simply substitute any convenient integer s' such that $s' \leq \operatorname{Re}(s)$.

void **mag_hurwitz_zeta_uiui**(*mag_t* res, *ulong* s, *ulong* a)

Sets *res* to an upper bound for $\zeta(s, a) = \sum_{k=0}^{\infty} (k+a)^{-s}$. We use the formula

$$\zeta(s, a) \leq \frac{1}{a^s} + \frac{1}{(s-1)a^{s-1}}$$

which is obtained by estimating the sum by an integral. If $s \leq 1$ or $a = 0$, the bound is infinite.

9.6 nfloat.h – packed floating-point numbers with n-word precision

This module provides binary floating-point numbers in a flat representation with precision in small fixed multiples of the word size (64, 128, 192, 256, ... bits on a 64-bit machine). The exponent range is close to a full word. A number with n -limb precision is stored as $n+2$ contiguous limbs as follows:

exponent limb
sign limb
mantissa[0]
...
mantissa[n-1]

For normal (nonzero and finite) values x , the most significant limb of the mantissa is always normalised to have its most significant bit set, and the exponent e is the unique integer such that $|x| \in [0.5, 1) \cdot 2^e$. Special (zero or nonfinite) values are encoded using special values of the exponent field, with junk data in the mantissa.

This type has the advantage that floating-point numbers with the same precision can be packed together tightly in vectors and created on the stack without heap allocation. The precision of an **nfloat** context object and its elements cannot be changed; to switch precision, one must convert to a different context

object. For higher precision than supported by `nfloating` and for calculations that require fine-grained precision adjustments, one should use `arf_t` instead.

The focus is on fast calculation, not bitwise-defined results. Atomic operations typically give slightly worse than correct rounding, e.g. with 1-2 ulp error. The rounding is not guaranteed to be identical on 64-bit and 32-bit machines. Planned features include:

- Support for special values (partially implemented)
- Optional (but slower) IEEE 754-like semantics
- Complex and ball types

This module is designed to use the *generics* interface. As such, the domain is represented by a `gr_ctx_t` context object, methods return status flags (`GR_SUCCESS`, `GR_UNABLE`, `GR_DOMAIN`), and one can use generic structures such as `gr_poly_t` for polynomials and `gr_mat_t` for matrices.

9.6.1 Types, macros and constants

`NFLOAT_MIN_LIMBS`

`NFLOAT_MAX_LIMBS`

The number of limbs n permitted as precision. The current limits are $1 \leq n \leq 66$ on a 64-bit machine and $1 \leq n \leq 132$ on a 32-bit machine, permitting precision up to 4224 bits. The upper limit exists so that elements and temporary buffers are safe to allocate on the stack and so that simple operations like swapping are not too expensive.

type `nfloating_ptr`

type `nfloating_srcptr`

Pointer to an `nfloating` element or vector of elements of any precision. Since this is a void type, one must cast to the correct size before doing pointer arithmetic, e.g. via the `GR_ENTRY` macro.

type `nfloating64_struct`

type `nfloating128_struct`

type `nfloating192_struct`

type `nfloating256_struct`

type `nfloating384_struct`

type `nfloating512_struct`

type `nfloating1024_struct`

type `nfloating2048_struct`

type `nfloating4096_struct`

type `nfloating64_t`

type `nfloating128_t`

type `nfloating192_t`

type `nfloating256_t`

type `nfloating384_t`

type `nfloating512_t`

type `nfloating1024_t`

type `nfloating2048_t`

type `nfloating4096_t`

For convenience we define types of the correct structure size for some common levels of bit precision. An `nfloatingX_t` is defined as a length-one array of `nfloatingX_struct`, permitting it to be passed by reference.

Sample usage:

```
gr_ctx_t ctx;
nfloat256_t x, y;

nfloat_ctx_init(ctx, 256, 0);  /* precision must match the type */
gr_init(x, ctx);
gr_init(y, ctx);

gr_ctx_println(ctx);

GR_MUST_SUCCEED(gr_set_ui(x, 5, ctx));
GR_MUST_SUCCEED(gr_set_ui(y, 7, ctx));
GR_MUST_SUCCEED(gr_div(x, x, y, ctx));
GR_MUST_SUCCEED(gr_println(x, ctx));

gr_clear(x, ctx);
gr_clear(y, ctx);
gr_ctx_clear(ctx);
```

NFLOAT_HEADER_LIMBS

NFLOAT_EXP(x)

NFLOAT_SGNBIT(x)

NFLOAT_D(x)

NFLOAT_DATA(x)

NFLOAT_MAX_ALLOC

NFLOAT_MIN_EXP

NFLOAT_MAX_EXP

NFLOAT_EXP_ZERO

NFLOAT_EXP_POS_INF

NFLOAT_EXP_NEG_INF

NFLOAT_EXP_NAN

NFLOAT_IS_SPECIAL(x)

NFLOAT_IS_ZERO(x)

NFLOAT_IS_POS_INF(x)

NFLOAT_IS_NEG_INF(x)

NFLOAT_IS_INF(x)

NFLOAT_IS_NAN(x)

9.6.2 Context objects

int **nfloat_ctx_init**(*gr_ctx_t* ctx, *long* prec, int flags)

Initializes *ctx* to represent a domain of floating-point numbers with bit precision *prec* rounded up to a full word (for example, *prec* = 53 actually creates a domain with 64-bit precision).

Returns **GR_UNABLE** without initializing the context object if the given precision is too large to be supported, otherwise returns **GR_SUCCESS**.

Admissible flags are listed below.

NFLOAT_ALLOW_UNDERFLOW

By default, operations that would underflow the exponent range output a garbage value and return **GR_UNABLE**. Setting this flag allows such operations to output zero and return **GR_SUCCESS** instead.

NFLOAT_ALLOW_INF

Allow creation of infinities. By default, operations that would overflow the exponent range output a garbage value and return `GR_UNABLE` or `GR_DOMAIN`. Setting this flag allows such operations to output an infinity and return `GR_SUCCESS` instead.

NFLOAT_ALLOW_NAN

Allow creation of NaNs. By default, operations that are meaningless output a garbage value and return `GR_UNABLE` or `GR_DOMAIN`. Setting this flag allows such operations to output NaN and return `GR_SUCCESS` instead.

NFLOAT_RND_FLOOR**NFLOAT_RND_CEIL**

Enable directed rounding towards $-\infty$ or $+\infty$ respectively. At most one of these flags may be set. If neither is set, rounding is done in an arbitrary direction for efficiency (usually but not always truncating towards zero). See below for a list of operations which support directed rounding.

Infinities and NaNs are disabled by default to improve performance, as this allows certain functions to skip checks for such values.

9.6.3 Basic operations and arithmetic

Basic functionality for the `gr` method table. These methods are interchangeable with their `gr` counterparts.

```
int nfloat_ctx_write(gr_stream_t out, gr_ctx_t ctx)
```

```
void nfloat_init(nfloat_ptr res, gr_ctx_t ctx)
```

Initializes *res* to the zero element.

```
void nfloat_clear(nfloat_ptr res, gr_ctx_t ctx)
```

Since `nfloat` elements do no allocation, this is a no-op.

```
void nfloat_swap(nfloat_ptr x, nfloat_ptr y, gr_ctx_t ctx)
```

```
int nfloat_set(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
```

```
truth_t nfloat_equal(nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
```

```
int nfloat_ctx_set_real_prec(gr_ctx_t ctx, slong prec)
```

Since `nfloat` contexts do not allow variable precision, this does nothing and returns `GR_UNABLE`.

```
int nfloat_ctx_get_real_prec(slong *res, gr_ctx_t ctx)
```

Sets *res* to the precision in bits and returns `GR_SUCCESS`.

```
int nfloat_zero(nfloat_ptr res, gr_ctx_t ctx)
```

```
int nfloat_one(nfloat_ptr res, gr_ctx_t ctx)
```

```
int nfloat_neg_one(nfloat_ptr res, gr_ctx_t ctx)
```

```
int nfloat_pos_inf(nfloat_ptr res, gr_ctx_t ctx)
```

```
int nfloat_neg_inf(nfloat_ptr res, gr_ctx_t ctx)
```

```
int nfloat_nan(nfloat_ptr res, gr_ctx_t ctx)
```

```
truth_t nfloat_is_zero(nfloat_srcptr x, gr_ctx_t ctx)
```

```
truth_t nfloat_is_one(nfloat_srcptr x, gr_ctx_t ctx)
```

```
truth_t nfloat_is_neg_one(nfloat_srcptr x, gr_ctx_t ctx)
```

```
int nfloat_set_ui(nfloat_ptr res, ulong x, gr_ctx_t ctx)
```

```
int nfloat_set_si(nfloat_ptr res, slong x, gr_ctx_t ctx)
```

```
int nfloat_set_fmpz(nfloat_ptr res, const fmpz_t x, gr_ctx_t ctx)

int _nfloat_set_mpn_2exp(nfloat_ptr res, nn_srcptr x, slong xn, slong exp, int xsgnbit, gr_ctx_t
    ctx)

int nfloat_set_mpn_2exp(nfloat_ptr res, nn_srcptr x, slong xn, slong exp, int xsgnbit, gr_ctx_t ctx)

int nfloat_set_arf(nfloat_ptr res, const arf_t x, gr_ctx_t ctx)
int nfloat_get_arf(arf_t res, nfloat_srcptr x, gr_ctx_t ctx)

int nfloat_get_d_2exp_si(double *m, slong *e, nfloat_srcptr x, gr_ctx_t ctx)

int nfloat_get_fmpz(fmpz_t res, nfloat_srcptr x, gr_ctx_t ctx)
    Succeeds only if x is integer-valued.

int nfloat_set_fmpq(nfloat_ptr res, const fmpq_t v, gr_ctx_t ctx)
int nfloat_set_d(nfloat_ptr res, double x, gr_ctx_t ctx)
int nfloat_set_str(nfloat_ptr res, const char *x, gr_ctx_t ctx)
int nfloat_set_other(nfloat_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t ctx)

int nfloat_write(gr_stream_t out, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_randtest(nfloat_ptr res, flint_rand_t state, gr_ctx_t ctx)

int nfloat_cmp(int *res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_cmpabs(int *res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)

int nfloat_neg(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_abs(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_add(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_sub(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_mul(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_submul(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_addmul(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_sqr(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)

int nfloat_mul_2exp_si(nfloat_ptr res, nfloat_srcptr x, slong y, gr_ctx_t ctx)

int nfloat_inv(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_div(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int nfloat_div_ui(nfloat_ptr res, nfloat_srcptr x, ulong y, gr_ctx_t ctx)
int nfloat_div_si(nfloat_ptr res, nfloat_srcptr x, slong y, gr_ctx_t ctx)

int nfloat_sqrt(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_rsqr(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)

int nfloat_sgn(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_im(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)

int nfloat_floor(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_ceil(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_trunc(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_nint(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
```

```

int nfloat_pow(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)

int nfloat_pi(nfloat_ptr res, gr_ctx_t ctx)
int nfloat_exp(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_expml(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_log(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_log1p(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_sin(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_cos(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_tan(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_sinh(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_cosh(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_tanh(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_atan(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_gamma(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)
int nfloat_zeta(nfloat_ptr res, nfloat_srcptr x, gr_ctx_t ctx)

```

9.6.4 Vector functions

Overrides for generic `gr` vector operations with inlined or partially inlined code for reduced overhead.

```

void _nfloat_vec_init(nfloat_ptr res, slong len, gr_ctx_t ctx)
void _nfloat_vec_clear(nfloat_ptr res, slong len, gr_ctx_t ctx)
int _nfloat_vec_set(nfloat_ptr res, nfloat_srcptr x, slong len, gr_ctx_t ctx)
int _nfloat_vec_zero(nfloat_ptr res, slong len, gr_ctx_t ctx)

int _nfloat_vec_add(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, slong len, gr_ctx_t ctx)
int _nfloat_vec_sub(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, slong len, gr_ctx_t ctx)
int _nfloat_vec_mul(nfloat_ptr res, nfloat_srcptr x, nfloat_srcptr y, slong len, gr_ctx_t ctx)
int _nfloat_vec_mul_scalar(nfloat_ptr res, nfloat_srcptr x, slong len, nfloat_srcptr y, gr_ctx_t ctx)
int _nfloat_vec_addmul_scalar(nfloat_ptr res, nfloat_srcptr x, slong len, nfloat_srcptr y, gr_ctx_t
                             ctx)
int _nfloat_vec_submul_scalar(nfloat_ptr res, nfloat_srcptr x, slong len, nfloat_srcptr y, gr_ctx_t
                             ctx)

int _nfloat_vec_dot(nfloat_ptr res, nfloat_srcptr initial, int subtract, nfloat_srcptr x, nfloat_srcptr
                   y, slong len, gr_ctx_t ctx)
int _nfloat_vec_dot_rev(nfloat_ptr res, nfloat_srcptr initial, int subtract, nfloat_srcptr x,
                       nfloat_srcptr y, slong len, gr_ctx_t ctx)

```

9.6.5 Matrix functions

```

int nfloat_mat_mul_fixed(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, slong
                        max_extra_prec, gr_ctx_t ctx)
int nfloat_mat_mul_block(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, slong
                        min_block_size, gr_ctx_t ctx)
int nfloat_mat_mul(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)

```

Different implementations of matrix multiplication.

```

int nfloat_mat_nonsingular_solve_tril(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int
                                     unit, gr_ctx_t ctx)

```

```
int nfloat_mat_nonsingular_solve_triu(gr_mat_t X, const gr_mat_t L, const gr_mat_t B, int
                                     unit, gr_ctx_t ctx)
int nfloat_mat_lu(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check,
                 gr_ctx_t ctx)
int nfloat_mat_lq(gr_mat_t L, gr_mat_t Q, const gr_mat_t A, gr_ctx_t ctx)
```

9.6.6 Internal functions

```
int _nfloat_underflow(nfloat_ptr res, int sgnbit, gr_ctx_t ctx)
int _nfloat_overflow(nfloat_ptr res, int sgnbit, gr_ctx_t ctx)

int _nfloat_cmp(nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int _nfloat_cmpabs(nfloat_srcptr x, nfloat_srcptr y, gr_ctx_t ctx)
int _nfloat_add_1(nfloat_ptr res, ulong x0, slong xexp, int xsgnbit, ulong y0, slong delta, gr_ctx_t
                 ctx)
int _nfloat_sub_1(nfloat_ptr res, ulong x0, slong xexp, int xsgnbit, ulong y0, slong delta, gr_ctx_t
                 ctx)
int _nfloat_add_2(nfloat_ptr res, nn_srcptr xd, slong xexp, int xsgnbit, nn_srcptr yd, slong delta,
                 gr_ctx_t ctx)
int _nfloat_sub_2(nfloat_ptr res, nn_srcptr xd, slong xexp, int xsgnbit, nn_srcptr yd, slong delta,
                 gr_ctx_t ctx)
int _nfloat_add_3(nfloat_ptr res, nn_srcptr x, slong xexp, int xsgnbit, nn_srcptr y, slong delta,
                 gr_ctx_t ctx)
int _nfloat_sub_3(nfloat_ptr res, nn_srcptr x, slong xexp, int xsgnbit, nn_srcptr y, slong delta,
                 gr_ctx_t ctx)
int _nfloat_add_4(nfloat_ptr res, nn_srcptr x, slong xexp, int xsgnbit, nn_srcptr y, slong delta,
                 gr_ctx_t ctx)
int _nfloat_sub_4(nfloat_ptr res, nn_srcptr x, slong xexp, int xsgnbit, nn_srcptr y, slong delta,
                 gr_ctx_t ctx)
int _nfloat_add_n(nfloat_ptr res, nn_srcptr xd, slong xexp, int xsgnbit, nn_srcptr yd, slong delta,
                 slong nlimbs, gr_ctx_t ctx)
int _nfloat_sub_n(nfloat_ptr res, nn_srcptr xd, slong xexp, int xsgnbit, nn_srcptr yd, slong delta,
                 slong nlimbs, gr_ctx_t ctx)
```

9.6.7 Complex numbers

Complex floating-point numbers have the obvious representation as real pairs.

```
type nfloat_complex_ptr
type nfloat_complex_srcptr

int nfloat_complex_ctx_init(gr_ctx_t ctx, slong prec, int flags)

NFLOAT_COMPLEX_CTX_DATA_NLIMBS(ctx)
NFLOAT_COMPLEX_RE(ptr, ctx)
NFLOAT_COMPLEX_IM(ptr, ctx)
NFLOAT_COMPLEX_IS_SPECIAL(x, ctx)
NFLOAT_COMPLEX_IS_ZERO(x, ctx)

void nfloat_complex_init(nfloat_complex_ptr res, gr_ctx_t ctx)
void nfloat_complex_clear(nfloat_complex_ptr res, gr_ctx_t ctx)
```

```

int nfloat_complex_zero(nfloat_complex_ptr res, gr_ctx_t ctx)
int nfloat_complex_get_acf(acf_t res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_set_acf(nfloat_complex_ptr res, const acf_t x, gr_ctx_t ctx)
int nfloat_complex_get_acb(acb_t res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_set_acb(nfloat_complex_ptr res, const acb_t x, gr_ctx_t ctx)
int nfloat_complex_write(gr_stream_t out, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_randtest(nfloat_complex_ptr res, flint_rand_t state, gr_ctx_t ctx)
void nfloat_complex_swap(nfloat_complex_ptr x, nfloat_complex_ptr y, gr_ctx_t ctx)
int nfloat_complex_set(nfloat_complex_ptr res, nfloat_complex_ptr x, gr_ctx_t ctx)
int nfloat_complex_one(nfloat_complex_ptr res, gr_ctx_t ctx)
int nfloat_complex_neg_one(nfloat_complex_ptr res, gr_ctx_t ctx)
truth_t nfloat_complex_is_zero(nfloat_complex_srcptr x, gr_ctx_t ctx)
truth_t nfloat_complex_is_one(nfloat_complex_srcptr x, gr_ctx_t ctx)
truth_t nfloat_complex_is_neg_one(nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_i(nfloat_complex_ptr res, gr_ctx_t ctx)
int nfloat_complex_pi(nfloat_complex_ptr res, gr_ctx_t ctx)
int nfloat_complex_conj(nfloat_complex_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_re(nfloat_complex_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_im(nfloat_complex_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
truth_t nfloat_complex_equal(nfloat_complex_srcptr x, nfloat_complex_srcptr y, gr_ctx_t ctx)
int nfloat_complex_set_si(nfloat_complex_ptr res, slong x, gr_ctx_t ctx)
int nfloat_complex_set_ui(nfloat_complex_ptr res, ulong x, gr_ctx_t ctx)
int nfloat_complex_set_fmpz(nfloat_complex_ptr res, const fmpz_t x, gr_ctx_t ctx)
int nfloat_complex_set_fmpq(nfloat_complex_ptr res, const fmpq_t x, gr_ctx_t ctx)
int nfloat_complex_set_d(nfloat_complex_ptr res, double x, gr_ctx_t ctx)
int nfloat_complex_set_other(nfloat_complex_ptr res, gr_srcptr x, gr_ctx_t x_ctx, gr_ctx_t ctx)
int nfloat_complex_neg(nfloat_complex_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_add(nfloat_complex_ptr res, nfloat_complex_srcptr x, nfloat_complex_srcptr y,
    gr_ctx_t ctx)
int nfloat_complex_sub(nfloat_complex_ptr res, nfloat_complex_srcptr x, nfloat_complex_srcptr y,
    gr_ctx_t ctx)
int _nfloat_complex_sqr_naive(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr b,
    gr_ctx_t ctx)
int _nfloat_complex_sqr_standard(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr
    b, gr_ctx_t ctx)
int _nfloat_complex_sqr_karatsuba(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr
    b, gr_ctx_t ctx)
int _nfloat_complex_sqr(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr b, gr_ctx_t
    ctx)
int nfloat_complex_sqr(nfloat_complex_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int _nfloat_complex_mul_naive(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr b,
    nfloat_srcptr c, nfloat_srcptr d, gr_ctx_t ctx)
int _nfloat_complex_mul_standard(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr
    b, nfloat_srcptr c, nfloat_srcptr d, gr_ctx_t ctx)
int _nfloat_complex_mul_karatsuba(nfloat_ptr res1, nfloat_ptr res2, nfloat_srcptr a, nfloat_srcptr
    b, nfloat_srcptr c, nfloat_srcptr d, gr_ctx_t ctx)
int nfloat_complex_mul(nfloat_complex_ptr res, nfloat_complex_srcptr x, nfloat_complex_srcptr y,
    gr_ctx_t ctx)

```

```

int nfloat_complex_inv(nfloat_complex_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
int nfloat_complex_div(nfloat_complex_ptr res, nfloat_complex_srcptr x, nfloat_complex_srcptr y,
    gr_ctx_t ctx)
int nfloat_complex_mul_2exp_si(nfloat_complex_ptr res, nfloat_complex_srcptr x, slong y,
    gr_ctx_t ctx)
int nfloat_complex_cmp(int *res, nfloat_complex_srcptr x, nfloat_complex_srcptr y, gr_ctx_t ctx)
int nfloat_complex_cmpabs(int *res, nfloat_complex_srcptr x, nfloat_complex_srcptr y, gr_ctx_t
    ctx)
int nfloat_complex_abs(nfloat_ptr res, nfloat_complex_srcptr x, gr_ctx_t ctx)
void _nfloat_complex_vec_init(nfloat_complex_ptr res, slong len, gr_ctx_t ctx)
void _nfloat_complex_vec_clear(nfloat_complex_ptr res, slong len, gr_ctx_t ctx)
int _nfloat_complex_vec_zero(nfloat_complex_ptr res, slong len, gr_ctx_t ctx)
int _nfloat_complex_vec_set(nfloat_complex_ptr res, nfloat_complex_srcptr x, slong len, gr_ctx_t
    ctx)
int _nfloat_complex_vec_add(nfloat_complex_ptr res, nfloat_complex_srcptr x,
    nfloat_complex_srcptr y, slong len, gr_ctx_t ctx)
int _nfloat_complex_vec_sub(nfloat_complex_ptr res, nfloat_complex_srcptr x,
    nfloat_complex_srcptr y, slong len, gr_ctx_t ctx)
int nfloat_complex_mat_mul_fixed(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, slong
    max_extra_prec, gr_ctx_t ctx)
int nfloat_complex_mat_mul_block(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, slong
    min_block_size, gr_ctx_t ctx)
int nfloat_complex_mat_mul_reorder(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t
    ctx)
int nfloat_complex_mat_mul(gr_mat_t C, const gr_mat_t A, const gr_mat_t B, gr_ctx_t ctx)
int nfloat_complex_mat_nonsingular_solve_tril(gr_mat_t X, const gr_mat_t L, const gr_mat_t
    B, int unit, gr_ctx_t ctx)
int nfloat_complex_mat_nonsingular_solve_triu(gr_mat_t X, const gr_mat_t L, const gr_mat_t
    B, int unit, gr_ctx_t ctx)
int nfloat_complex_mat_lu(slong *rank, slong *P, gr_mat_t LU, const gr_mat_t A, int rank_check,
    gr_ctx_t ctx)

```

9.6.8 Packed fixed-point arithmetic

A fixed-point number in the range $(-1, 1)$ with n -limb precision is represented as $n + 1$ contiguous limbs as follows:

sign limb
mantissa[0]
...
mantissa[n-1]

In the following method signatures, `nlimbs` always refers to the precision n while the storage is `nlimbs + 1`.

There is no overflow handling: all methods assume that inputs have been scaled to a range $[-\varepsilon, \varepsilon]$ so that all intermediate results (including rounding errors) lie in $(-1, 1)$.

```
void _nfixed_print(nn_srcptr x, slong nlimbs, slong exp)
```

Print the fixed-point number

```
void _nfixed_vec_add(nn_ptr res, nn_srcptr a, nn_srcptr b, slong len, slong nlimbs)
```



```
void _nfixed_vec_sub(nn_ptr res, nn_srcptr a, nn_srcptr b, slong len, slong nlimbs)
```

Vectorized addition or subtraction of *len* fixed-point numbers.

```
void _nfixed_dot_2(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

```
void _nfixed_dot_3(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

```
void _nfixed_dot_4(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

```
void _nfixed_dot_5(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

```
void _nfixed_dot_6(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

```
void _nfixed_dot_7(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

```
void _nfixed_dot_8(nn_ptr res, nn_srcptr x, slong xstride, nn_srcptr y, slong ystride, slong len)
```

Dot product with a fixed number of limbs. The *xstride* and *ystride* parameters indicate the offset in number of limbs between consecutive entries and may be negative.

```
void _nfixed_mat_mul_classical_precise(nn_ptr C, nn_srcptr A, nn_srcptr B, slong m, slong n,
                                       slong p, slong nlimbs)
```

```
void _nfixed_mat_mul_classical(nn_ptr C, nn_srcptr A, nn_srcptr B, slong m, slong n, slong p,
                               slong nlimbs)
```

```
void _nfixed_mat_mul_waksman(nn_ptr C, nn_srcptr A, nn_srcptr B, slong m, slong n, slong p,
                             slong nlimbs)
```

```
void _nfixed_mat_mul_strassen(nn_ptr C, nn_srcptr A, nn_srcptr B, slong m, slong n, slong p,
                              slong cutoff, slong nlimbs)
```

```
void _nfixed_mat_mul(nn_ptr C, nn_srcptr A, nn_srcptr B, slong m, slong n, slong p, slong nlimbs)
```

Matrix multiplication using various algorithms. The *strassen* variant takes a *cutoff* parameter specifying where to switch from basecase multiplication to Strassen multiplication. The *classical_precise* version computes with one extra limb of internal precision; this is only intended for testing purposes.

```
void _nfixed_mat_mul_bound_classical(double *bound, double *error, slong m, slong n, slong p,
                                     double A, double B, slong nlimbs)
```

```
void _nfixed_mat_mul_bound_waksman(double *bound, double *error, slong m, slong n, slong p,
                                   double A, double B, slong nlimbs)
```

```
void _nfixed_mat_mul_bound_strassen(double *bound, double *error, slong m, slong n, slong p,
                                    double A, double B, slong cutoff, slong nlimbs)
```

```
void _nfixed_mat_mul_bound(double *bound, double *error, slong m, slong n, slong p, double A,
                           double B, slong nlimbs)
```

```
void _nfixed_complex_mat_mul_bound(double *bound, double *error, slong m, slong n, slong p,
                                   double A, double B, double C, double D, slong nlimbs)
```

For the respective matrix multiplication algorithm, computes bounds for a size $m \times n \times p$ product at precision *nlimbs* given entrywise bounds *A* and *B*.

The *bound* output is set to a bound for the entries in all intermediate variables of the computation. This should be < 1 to ensure correctness. The *error* output is set to a bound for the output error, measured in ulp. The caller can assume that the computed bounds are nondecreasing functions of *A* and *B*.

For complex multiplication, the entrywise bounds are for $A + Bi$ and $C + Di$.

9.6.9 Directed rounding (experimental)

Currently, the following operations respect the rounding mode toggled with `NFLOAT_CTX_RND_FLOOR` `NFLOAT_CTX_RND_CEIL`. Warning: directed rounding may not function correctly in combination with the `NFLOAT_ALLOW_UNDERFLOW` flag.

- Operations that are always exact, including `set`, `neg`, `abs`, `floor`, `ceil`, `nint`, `trunc`, `sgn`.
- Simple conversions from other types:
 - `set_si`, `set_ui`
 - `set_d`
 - `set_arf`
 - `set_fmpz`
 - `set_fmpq`
 - `set_other` with an `fmpz`, `fmpq`, `arf` or `nfloat` operand
- Basic arithmetic
 - `add`, `add_ui`, `add_si`
 - `sub`, `sub_ui`, `sub_si`
 - `mul`, `mul_ui`, `mul_si`
 - `mul_2exp_si`
 - `sqr`
 - `addmul`, `addmul_ui`, `addmul_si`
 - `submul`
 - `inv`
 - `div`, `div_ui`, `div_si`
 - `sqrt`
 - `rsqrt`
 - Vector versions of any of the above, e.g. `_gr_vec_mul_scalar_ui`
 - `vec_dot`
 - `vec_dot_rev`
 - `mat_mul`

The following operations currently **do not** respect the rounding mode:

- Fused operations and generic algorithms other than those above
- `set_str`
- `set_other` except for the types listed above
- Mixed arithmetic operations with an `other`, `fmpz` or `fmpq` operand
- Transcendental functions
- `nfloat_complex` operations

9.7 arf.h – arbitrary-precision floating-point numbers

A variable of type `arf_t` holds an arbitrary-precision binary floating-point number: that is, a rational number of the form $x \cdot 2^y$ where $x, y \in \mathbb{Z}$ and x is odd, or one of the special values zero, plus infinity, minus infinity, or NaN (not-a-number). There is currently no support for negative zero, unsigned infinity, or a NaN with a payload.

The *exponent* of a finite and nonzero floating-point number can be defined in different ways: for example, as the component y above, or as the unique integer e such that $x \cdot 2^y = m \cdot 2^e$ where $0.5 \leq |m| < 1$. The internal representation of an `arf_t` stores the exponent in the latter format.

Except where otherwise noted, functions have the following semantics:

- Functions taking *prec* and *rnd* parameters at the end of the argument list and returning an `int` flag round the result in the output variable to *prec* bits in the direction specified by *rnd*. The return flag is 0 if the result is exact (not rounded) and 1 if the result is inexact (rounded). Correct rounding is guaranteed: the result is the floating-point number obtained by viewing the inputs as exact numbers, in principle carrying out the mathematical operation exactly, and rounding the resulting real number to the nearest representable floating-point number whose mantissa has at most the specified number of bits, in the specified direction of rounding. In particular, the error is at most 1 ulp with directed rounding modes and 0.5 ulp when rounding to nearest.
- Other functions perform the operation exactly.

Since exponents are bignums, overflow or underflow cannot occur.

9.7.1 Types, macros and constants

type `arf_struct`

type `arf_t`

An `arf_struct` contains four words: an *fmpz* exponent (*exp*), a *size* field tracking the number of limbs used (one bit of this field is also used for the sign of the number), and two more words. The last two words hold the value directly if there are at most two limbs, and otherwise contain one *alloc* field (tracking the total number of allocated limbs, not all of which might be used) and a pointer to the actual limbs. Thus, up to 128 bits on a 64-bit machine and 64 bits on a 32-bit machine, no space outside of the `arf_struct` is used.

An `arf_t` is defined as an array of length one of type `arf_struct`, permitting an `arf_t` to be passed by reference.

type `arf_rnd_t`

Specifies the rounding mode for the result of an approximate operation.

`ARF_RND_DOWN`

Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards zero.

`ARF_RND_UP`

Specifies that the result of an operation should be rounded to the nearest representable number in the direction away from zero.

`ARF_RND_FLOOR`

Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards minus infinity.

`ARF_RND_CEIL`

Specifies that the result of an operation should be rounded to the nearest representable number in the direction towards plus infinity.

ARF_RND_NEAR

Specifies that the result of an operation should be rounded to the nearest representable number, rounding to even if there is a tie between two values.

ARF_PREC_EXACT

If passed as the precision parameter to a function, indicates that no rounding is to be performed.

Warning: use of this value is unsafe in general. It must only be passed as input under the following two conditions:

- The operation in question can inherently be viewed as an exact operation in $\mathbb{Z}[\frac{1}{2}]$ for all possible inputs, provided that the precision is large enough. Examples include addition, multiplication, conversion from integer types to arbitrary-precision floating-point types, and evaluation of some integer-valued functions.
- The exact result of the operation will certainly fit in memory. Note that, for example, adding two numbers whose exponents are far apart can easily produce an exact result that is far too large to store in memory.

The typical use case is to work with small integer values, double precision constants, and the like. It is also useful when writing test code. If in doubt, simply try with some convenient high precision instead of using this special value, and check that the result is exact.

9.7.2 Memory management

void **arf_init**(*arf_t* x)

Initializes the variable *x* for use. Its value is set to zero.

void **arf_clear**(*arf_t* x)

Clears the variable *x*, freeing or recycling its allocated memory.

slong **arf_allocated_bytes**(const *arf_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(arf_struct)` to get the size of the object as a whole.

9.7.3 Special values

void **arf_zero**(*arf_t* res)

void **arf_one**(*arf_t* res)

void **arf_pos_inf**(*arf_t* res)

void **arf_neg_inf**(*arf_t* res)

void **arf_nan**(*arf_t* res)

Sets *res* respectively to 0, 1, $+\infty$, $-\infty$, NaN.

int **arf_is_zero**(const *arf_t* x)

int **arf_is_one**(const *arf_t* x)

int **arf_is_pos_inf**(const *arf_t* x)

int **arf_is_neg_inf**(const *arf_t* x)

int **arf_is_nan**(const *arf_t* x)

Returns nonzero iff *x* respectively equals 0, 1, $+\infty$, $-\infty$, NaN.

int **arf_is_inf**(const *arf_t* x)

Returns nonzero iff *x* equals either $+\infty$ or $-\infty$.

int **arf_is_normal**(const *arf_t* x)

Returns nonzero iff x is a finite, nonzero floating-point value, i.e. not one of the special values 0, $+\infty$, $-\infty$, NaN.

int **arf_is_special**(const *arf_t* x)

Returns nonzero iff x is one of the special values 0, $+\infty$, $-\infty$, NaN, i.e. not a finite, nonzero floating-point value.

int **arf_is_finite**(const *arf_t* x)

Returns nonzero iff x is a finite floating-point value, i.e. not one of the values $+\infty$, $-\infty$, NaN. (Note that this is not equivalent to the negation of *arf_is_inf()*.)

9.7.4 Assignment, rounding and conversions

void **arf_set**(*arf_t* res, const *arf_t* x)

void **arf_set_mpmz**(*arf_t* res, const *mpz_t* x)

void **arf_set_fmpz**(*arf_t* res, const *fmpz_t* x)

void **arf_set_ui**(*arf_t* res, *ulong* x)

void **arf_set_si**(*arf_t* res, *slong* x)

void **arf_set_mpfr**(*arf_t* res, const *mpfr_t* x)

void **arf_set_d**(*arf_t* res, double x)

Sets *res* to the exact value of x .

void **arf_swap**(*arf_t* x, *arf_t* y)

Swaps x and y efficiently.

void **arf_init_set_ui**(*arf_t* res, *ulong* x)

void **arf_init_set_si**(*arf_t* res, *slong* x)

Initializes *res* and sets it to x in a single operation.

int **arf_set_round**(*arf_t* res, const *arf_t* x, *slong* prec, *arf_rnd_t* rnd)

int **arf_set_round_si**(*arf_t* res, *slong* x, *slong* prec, *arf_rnd_t* rnd)

int **arf_set_round_ui**(*arf_t* res, *ulong* x, *slong* prec, *arf_rnd_t* rnd)

int **arf_set_round_mpmz**(*arf_t* res, const *mpz_t* x, *slong* prec, *arf_rnd_t* rnd)

int **arf_set_round_fmpz**(*arf_t* res, const *fmpz_t* x, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to x , rounded to *prec* bits in the direction specified by *rnd*.

void **arf_set_si_2exp_si**(*arf_t* res, *slong* m, *slong* e)

void **arf_set_ui_2exp_si**(*arf_t* res, *ulong* m, *slong* e)

void **arf_set_fmpz_2exp**(*arf_t* res, const *fmpz_t* m, const *fmpz_t* e)

Sets *res* to $m \cdot 2^e$.

int **arf_set_round_fmpz_2exp**(*arf_t* res, const *fmpz_t* x, const *fmpz_t* e, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to $x \cdot 2^e$, rounded to *prec* bits in the direction specified by *rnd*.

void **arf_get_fmpz_2exp**(*fmpz_t* m, *fmpz_t* e, const *arf_t* x)

Sets m and e to the unique integers such that $x = m \cdot 2^e$ and m is odd, provided that x is a nonzero finite fraction. If x is zero, both m and e are set to zero. If x is infinite or NaN, the result is undefined.

void **arf_frexp**(*arf_t* m, *fmpz_t* e, const *arf_t* x)

Writes x as $m \cdot 2^e$, where $0.5 \leq |m| < 1$ if x is a normal value. If x is a special value, copies this to m and sets e to zero. Note: for the inverse operation (*ldexp*), use *arf_mul_2exp_fmpz()*.

double **arf_get_d**(const *arf_t* x, *arf_rnd_t* rnd)

Returns x rounded to a double in the direction specified by *rnd*. This method rounds correctly when overflowing or underflowing the double exponent range (this was not the case in an earlier version).

int **arf_get_mpfr**(*mpfr_t* res, const *arf_t* x, *mpfr_rnd_t* rnd)

Sets the MPFR variable *res* to the value of x . If the precision of x is too small to allow *res* to be represented exactly, it is rounded in the specified MPFR rounding mode. The return value (-1, 0 or 1) indicates the direction of rounding, following the convention of the MPFR library.

If x has an exponent too large or small to fit in the MPFR type, the result overflows to an infinity or underflows to a (signed) zero, and the corresponding MPFR exception flags are set.

int **arf_get_fmpz**(*fmpz_t* res, const *arf_t* x, *arf_rnd_t* rnd)

Sets *res* to x rounded to the nearest integer in the direction specified by *rnd*. If *rnd* is *ARF_RND_NEAR*, rounds to the nearest even integer in case of a tie. Returns inexact (beware: accordingly returns whether x is *not* an integer).

This method aborts if x is infinite or NaN, or if the exponent of x is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if the exponent of x is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that x is within a reasonable range before calling this method.

slong **arf_get_si**(const *arf_t* x, *arf_rnd_t* rnd)

Returns x rounded to the nearest integer in the direction specified by *rnd*. If *rnd* is *ARF_RND_NEAR*, rounds to the nearest even integer in case of a tie. Aborts if x is infinite, NaN, or the value is too large to fit in a slong.

int **arf_get_fmpz_fixed_fmpz**(*fmpz_t* res, const *arf_t* x, const *fmpz_t* e)

int **arf_get_fmpz_fixed_si**(*fmpz_t* res, const *arf_t* x, slong e)

Converts x to a mantissa with predetermined exponent, i.e. sets *res* to an integer y such that $y \times 2^e \approx x$, truncating if necessary. Returns 0 if exact and 1 if truncation occurred.

The warnings for *arf_get_fmpz()* apply.

void **arf_floor**(*arf_t* res, const *arf_t* x)

void **arf_ceil**(*arf_t* res, const *arf_t* x)

void **arf_nint**(*arf_t* res, const *arf_t* x)

Sets *res* respectively to the floor $\lfloor x \rfloor$, ceiling $\lceil x \rceil$ or nearest integer $\lfloor x \rfloor$ (with ties-to-even). The result is always represented exactly, requiring no more bits to store than the input. To round the result to a floating-point number with a lower precision, call *arf_set_round()* afterwards.

void **arf_get_fmpq**(*fmpq_t* res, const *arf_t* x)

Set *res* to the exact rational value of x . This method aborts if x is infinite or NaN, or if the exponent of x is so large that allocating memory for the result fails.

9.7.5 Comparisons and bounds

int **arf_equal**(const *arf_t* x, const *arf_t* y)

int **arf_equal_si**(const *arf_t* x, *slong* y)

int **arf_equal_ui**(const *arf_t* x, *ulong* y)

int **arf_equal_d**(const *arf_t* x, double y)

Returns nonzero iff x and y are exactly equal. NaN is not treated specially, i.e. NaN compares as equal to itself.

For comparison with a *double*, the values -0 and $+0$ are both treated as zero, and all NaN values are treated as identical.

int **arf_cmp**(const *arf_t* x, const *arf_t* y)

int **arf_cmp_si**(const *arf_t* x, *slong* y)

int **arf_cmp_ui**(const *arf_t* x, *ulong* y)

int **arf_cmp_d**(const *arf_t* x, double y)

Returns negative, zero, or positive, depending on whether x is respectively smaller, equal, or greater compared to y . Comparison with NaN is undefined.

int **arf_cmpabs**(const *arf_t* x, const *arf_t* y)

int **arf_cmpabs_ui**(const *arf_t* x, *ulong* y)

int **arf_cmpabs_d**(const *arf_t* x, double y)

int **arf_cmpabs_mag**(const *arf_t* x, const *mag_t* y)

Compares the absolute values of x and y .

int **arf_cmp_2exp_si**(const *arf_t* x, *slong* e)

int **arf_cmpabs_2exp_si**(const *arf_t* x, *slong* e)

Compares x (respectively its absolute value) with 2^e .

int **arf_sgn**(const *arf_t* x)

Returns -1 , 0 or $+1$ according to the sign of x . The sign of NaN is undefined.

void **arf_min**(*arf_t* res, const *arf_t* a, const *arf_t* b)

void **arf_max**(*arf_t* res, const *arf_t* a, const *arf_t* b)

Sets res respectively to the minimum and the maximum of a and b .

slong **arf_bits**(const *arf_t* x)

Returns the number of bits needed to represent the absolute value of the mantissa of x , i.e. the minimum precision sufficient to represent x exactly. Returns 0 if x is a special value.

int **arf_is_int**(const *arf_t* x)

Returns nonzero iff x is integer-valued.

int **arf_is_int_2exp_si**(const *arf_t* x, *slong* e)

Returns nonzero iff x equals $n2^e$ for some integer n .

void **arf_abs_bound_lt_2exp_fmpz**(*fmpz_t* res, const *arf_t* x)

Sets res to the smallest integer b such that $|x| < 2^b$. If x is zero, infinity or NaN, the result is undefined.

void **arf_abs_bound_le_2exp_fmpz**(*fmpz_t* res, const *arf_t* x)

Sets res to the smallest integer b such that $|x| \leq 2^b$. If x is zero, infinity or NaN, the result is undefined.

slong **arf_abs_bound_lt_2exp_si**(const *arf_t* x)

Returns the smallest integer b such that $|x| < 2^b$, clamping the result to lie between $-ARF\_PREC\_EXACT$ and ARF_PREC_EXACT inclusive. If x is zero, $-ARF\_PREC\_EXACT$ is returned, and if x is infinity or NaN, ARF_PREC_EXACT is returned.

9.7.6 Magnitude functions

void **arf_get_mag**(*mag_t* res, const *arf_t* x)

Sets *res* to an upper bound for the absolute value of x .

void **arf_get_mag_lower**(*mag_t* res, const *arf_t* x)

Sets *res* to a lower bound for the absolute value of x .

void **arf_set_mag**(*arf_t* res, const *mag_t* x)

Sets *res* to x . This operation is exact.

void **mag_init_set_arf**(*mag_t* res, const *arf_t* x)

Initializes *res* and sets it to an upper bound for x .

void **mag_fast_init_set_arf**(*mag_t* res, const *arf_t* x)

Initializes *res* and sets it to an upper bound for x . Assumes that the exponent of *res* is small (this function is unsafe).

void **arf_mag_set_ulp**(*mag_t* res, const *arf_t* x, *slong* prec)

Sets *res* to the magnitude of the unit in the last place (ulp) of x at precision *prec*.

void **arf_mag_add_ulp**(*mag_t* res, const *mag_t* x, const *arf_t* y, *slong* prec)

Sets *res* to an upper bound for the sum of x and the magnitude of the unit in the last place (ulp) of y at precision *prec*.

void **arf_mag_fast_add_ulp**(*mag_t* res, const *mag_t* x, const *arf_t* y, *slong* prec)

Sets *res* to an upper bound for the sum of x and the magnitude of the unit in the last place (ulp) of y at precision *prec*. Assumes that all exponents are small.

9.7.7 Shallow assignment

void **arf_init_set_shallow**(*arf_t* z, const *arf_t* x)

void **arf_init_set_mag_shallow**(*arf_t* z, const *mag_t* x)

Initializes z to a shallow copy of x . A shallow copy just involves copying struct data (no heap allocation is performed).

The target variable z may not be cleared or modified in any way (it can only be used as constant input to functions), and may not be used after x has been cleared. Moreover, after x has been assigned shallowly to z , no modification of x is permitted as long as z is in use.

void **arf_init_neg_shallow**(*arf_t* z, const *arf_t* x)

void **arf_init_neg_mag_shallow**(*arf_t* z, const *mag_t* x)

Initializes z shallowly to the negation of x .

9.7.8 Random number generation

void **arf_randtest**(*arf_t* res, *flint_rand_t* state, *slong* bits, *slong* mag_bits)

Generates a finite random number whose mantissa has precision at most *bits* and whose exponent has at most *mag_bits* bits. The values are distributed non-uniformly: special bit patterns are generated with high probability in order to allow the test code to exercise corner cases.

void **arf_randtest_not_zero**(*arf_t* res, *flint_rand_t* state, *slong* bits, *slong* mag_bits)

Identical to *arf_randtest()*, except that zero is never produced as an output.

void **arf_randtest_special**(*arf_t* res, *flint_rand_t* state, *slong* bits, *slong* mag_bits)

Identical to *arf_randtest()*, except that the output occasionally is set to an infinity or NaN.

void **arf_urandom**(*arf_t* res, *flint_rand_t* state, *slong* bits, *arf_rnd_t* rnd)

Sets *res* to a uniformly distributed random number in the interval $[0, 1]$. The method uses rounding from integers to floats based on the rounding mode *rnd*.

9.7.9 Input and output

void **arf_debug**(const *arf_t* x)

Prints information about the internal representation of *x*.

void **arf_print**(const *arf_t* x)

Prints *x* as an integer mantissa and exponent.

void **arf_printd**(const *arf_t* x, *slong* d)

Prints *x* as a decimal floating-point number, rounding to *d* digits. Rounding is faithful (at most 1 ulp error).

char ***arf_get_str**(const *arf_t* x, *slong* d)

Returns *x* as a decimal floating-point number, rounding to *d* digits. Rounding is faithful (at most 1 ulp error).

void **arf_fprint**(FILE *file, const *arf_t* x)

Prints *x* as an integer mantissa and exponent to the stream *file*.

void **arf_fprintd**(FILE *file, const *arf_t* y, *slong* d)

Prints *x* as a decimal floating-point number to the stream *file*, rounding to *d* digits. Rounding is faithful (at most 1 ulp error).

char ***arf_dump_str**(const *arf_t* x)

Allocates a string and writes a binary representation of *x* to it that can be read by *arf_load_str()*. The returned string needs to be deallocated with *flint_free*.

int **arf_load_str**(*arf_t* x, const char *str)

Parses *str* into *x*. Returns a nonzero value if *str* is not formatted correctly.

int **arf_dump_file**(FILE *stream, const *arf_t* x)

Writes a binary representation of *x* to *stream* that can be read by *arf_load_file()*. Returns a nonzero value if the data could not be written.

int **arf_load_file**(*arf_t* x, FILE *stream)

Reads *x* from *stream*. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with *arf_dump_file()* make sure to insert a whitespace to separate consecutive values.

9.7.10 Addition and multiplication

void **arf_abs**(*arf_t* res, const *arf_t* x)

Sets *res* to the absolute value of *x* exactly.

void **arf_neg**(*arf_t* res, const *arf_t* x)

Sets *res* to $-x$ exactly.

int **arf_neg_round**(*arf_t* res, const *arf_t* x, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to $-x$.

int **arf_add**(*arf_t* res, const *arf_t* x, const *arf_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_add_si**(*arf_t* res, const *arf_t* x, *slong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_add_ui**(*arf_t* res, const *arf_t* x, *ulong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_add_fmpz**(*arf_t* res, const *arf_t* x, const *fmpz_t* y, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to $x + y$.

int **arf_add_fmpz_2exp**(*arf_t* res, const *arf_t* x, const *fmpz_t* y, const *fmpz_t* e, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to $x + y2^e$.

int **arf_sub**(*arf_t* res, const *arf_t* x, const *arf_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_sub_si**(*arf_t* res, const *arf_t* x, *slong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_sub_ui**(*arf_t* res, const *arf_t* x, *ulong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_sub_fmpz**(*arf_t* res, const *arf_t* x, const *fmpz_t* y, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to $x - y$.

void **arf_mul_2exp_si**(*arf_t* res, const *arf_t* x, *slong* e)

void **arf_mul_2exp_fmpz**(*arf_t* res, const *arf_t* x, const *fmpz_t* e)

Sets *res* to $x2^e$ exactly.

int **arf_mul**(*arf_t* res, const *arf_t* x, const *arf_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_mul_ui**(*arf_t* res, const *arf_t* x, *ulong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_mul_si**(*arf_t* res, const *arf_t* x, *slong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_mul_mpz**(*arf_t* res, const *arf_t* x, const *mpz_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_mul_fmpz**(*arf_t* res, const *arf_t* x, const *fmpz_t* y, *slong* prec, *arf_rnd_t* rnd)

Sets *res* to $x \cdot y$.

int **arf_addmul**(*arf_t* z, const *arf_t* x, const *arf_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_addmul_ui**(*arf_t* z, const *arf_t* x, *ulong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_addmul_si**(*arf_t* z, const *arf_t* x, *slong* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_addmul_mpz**(*arf_t* z, const *arf_t* x, const *mpz_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_addmul_fmpz**(*arf_t* z, const *arf_t* x, const *fmpz_t* y, *slong* prec, *arf_rnd_t* rnd)

Performs a fused multiply-add $z = z + x \cdot y$, updating *z* in-place.

int **arf_submul**(*arf_t* z, const *arf_t* x, const *arf_t* y, *slong* prec, *arf_rnd_t* rnd)

int **arf_submul_ui**(*arf_t* z, const *arf_t* x, *ulong* y, *slong* prec, *arf_rnd_t* rnd)

```
int arf_submul_si(arf_t z, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_submul_mpz(arf_t z, const arf_t x, const mpz_t y, slong prec, arf_rnd_t rnd)
int arf_submul_fmpz(arf_t z, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
    Performs a fused multiply-subtract  $z = z - x \cdot y$ , updating  $z$  in-place.
int arf_fma(arf_t res, const arf_t x, const arf_t y, const arf_t z, slong prec, arf_rnd_t rnd)
    Sets  $res$  to  $x \cdot y + z$ . This is equivalent to an addmul except that  $res$  and  $z$  can be separate variables.
int arf_sosq(arf_t res, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
    Sets  $res$  to  $x^2 + y^2$ , rounded to  $prec$  bits in the direction specified by  $rnd$ .
```

9.7.11 Summation

```
int arf_sum(arf_t res, arf_srcptr terms, slong len, slong prec, arf_rnd_t rnd)
    Sets  $res$  to the sum of the array terms of length  $len$ , rounded to  $prec$  bits in the direction specified by  $rnd$ . The sum is computed as if done without any intermediate rounding error, with only a single rounding applied to the final result. Unlike repeated calls to arf_add() with infinite precision, this function does not overflow if the magnitudes of the terms are far apart. Warning: this function is implemented naively, and the running time is quadratic with respect to  $len$  in the worst case.
```

9.7.12 Dot products

```
int arf_dot(arf_t res, const arf_t initial, int subtract, arf_srcptr x, slong xstep, arf_srcptr y, slong ystep, slong len, slong prec, arf_rnd_t rnd)
    Computes a dot product, with the same meaning of the parameters as arb_dot(). The result is correctly rounded in the direction specified by  $rnd$ . This function is currently not optimized for speed.
void arf_approx_dot(arf_t res, const arf_t initial, int subtract, arf_srcptr x, slong xstep, arf_srcptr y, slong ystep, slong len, slong prec, arf_rnd_t rnd)
    Computes an approximate dot product, with the same meaning of the parameters as arb_dot(). This operation is not correctly rounded: the final rounding is done in the direction  $rnd$  but intermediate roundings are implementation-defined.
```

9.7.13 Division

```
int arf_div(arf_t res, const arf_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_div_ui(arf_t res, const arf_t x, ulong y, slong prec, arf_rnd_t rnd)
int arf_ui_div(arf_t res, ulong x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_div_si(arf_t res, const arf_t x, slong y, slong prec, arf_rnd_t rnd)
int arf_si_div(arf_t res, slong x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_div_fmpz(arf_t res, const arf_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
int arf_fmpz_div(arf_t res, const fmpz_t x, const arf_t y, slong prec, arf_rnd_t rnd)
int arf_fmpz_div_fmpz(arf_t res, const fmpz_t x, const fmpz_t y, slong prec, arf_rnd_t rnd)
    Sets  $res$  to  $x/y$ , rounded to  $prec$  bits in the direction specified by  $rnd$ , returning nonzero iff the operation is inexact. The result is NaN if  $y$  is zero.
```

9.7.14 Square roots

int **arf_sqrt**(arf_t res, const arf_t x, slong prec, arf_rnd_t rnd)

int **arf_sqrt_ui**(arf_t res, ulong x, slong prec, arf_rnd_t rnd)

int **arf_sqrt_fmpz**(arf_t res, const fmpz_t x, slong prec, arf_rnd_t rnd)

Sets *res* to $\sqrt{x}$. The result is NaN if *x* is negative.

int **arf_rsqr**(arf_t res, const arf_t x, slong prec, arf_rnd_t rnd)

Sets *res* to $1/\sqrt{x}$. The result is NaN if *x* is negative, and $+\infty$ if *x* is zero.

int **arf_root**(arf_t res, const arf_t x, ulong k, slong prec, arf_rnd_t rnd)

Sets *res* to $x^{1/k}$. The result is NaN if *x* is negative. Warning: this function is a wrapper around the MPFR root function. It gets slow and uses much memory for large *k*. Consider working with **arb_root_ui**() for large *k* instead of using this function directly.

9.7.15 Complex arithmetic

int **arf_complex_mul**(arf_t e, arf_t f, const arf_t a, const arf_t b, const arf_t c, const arf_t d, slong prec, arf_rnd_t rnd)

int **arf_complex_mul_fallback**(arf_t e, arf_t f, const arf_t a, const arf_t b, const arf_t c, const arf_t d, slong prec, arf_rnd_t rnd)

Computes the complex product $e + fi = (a + bi)(c + di)$, rounding both *e* and *f* correctly to *prec* bits in the direction specified by *rnd*. The first bit in the return code indicates inexactness of *e*, and the second bit indicates inexactness of *f*.

If any of the components *a*, *b*, *c*, *d* is zero, two real multiplications and no additions are done. This convention is used even if any other part contains an infinity or NaN, and the behavior with infinite/NaN input is defined accordingly.

The *fallback* version is implemented naively, for testing purposes. No squaring optimization is implemented.

int **arf_complex_sqr**(arf_t e, arf_t f, const arf_t a, const arf_t b, slong prec, arf_rnd_t rnd)

Computes the complex square $e + fi = (a + bi)^2$. This function has identical semantics to **arf_complex_mul**() (with *c* = *a*, *b* = *d*), but is faster.

9.7.16 Low-level methods

int **_arf_get_integer_mpn**(nn_ptr y, nn_srcptr xp, slong xn, slong exp)

Given a floating-point number *x* represented by *xn* limbs at *xp* and an exponent *exp*, writes the integer part of *x* to *y*, returning whether the result is inexact. The correct number of limbs is written (no limbs are written if the integer part of *x* is zero). Assumes that *xp*[0] is nonzero and that the top bit of *xp*[*xn*-1] is set.

int **_arf_set_mpn_fixed**(arf_t z, nn_srcptr xp, slong xn, slong fixn, int negative, slong prec, arf_rnd_t rnd)

Sets *z* to the fixed-point number having *xn* total limbs and *fixn* fractional limbs, negated if *negative* is set, rounding *z* to *prec* bits in the direction *rnd* and returning whether the result is inexact. Both *xn* and *fixn* must be nonnegative and not so large that the bit shift would overflow an *slong*, but otherwise no assumptions are made about the input.

int **_arf_set_round_ui**(arf_t z, ulong x, int sgnbit, slong prec, arf_rnd_t rnd)

Sets *z* to the integer *x*, negated if *sgnbit* is 1, rounded to *prec* bits in the direction specified by *rnd*. There are no assumptions on *x*.

int **_arf_set_round_uiui**(*arf_t* z, *slong* *fix, *ulong* hi, *ulong* lo, int sgnbit, *slong* prec, *arf_rnd_t* rnd)

Sets the mantissa of z to the two-limb mantissa given by hi and lo , negated if $sgnbit$ is 1, rounded to $prec$ bits in the direction specified by rnd . Requires that not both hi and lo are zero. Writes the exponent shift to fix without writing the exponent of z directly.

int **_arf_set_round_mpn**(*arf_t* z, *slong* *exp_shift, *nn_srcptr* x, *slong* xn, int sgnbit, *slong* prec, *arf_rnd_t* rnd)

Sets the mantissa of z to the mantissa given by the xn limbs in x , negated if $sgnbit$ is 1, rounded to $prec$ bits in the direction specified by rnd . Returns the inexact flag. Requires that xn is positive and that the top limb of x is nonzero. If x has leading zero bits, writes the shift to exp_shift . This method does not write the exponent of z directly. Requires that x does not point to the limbs of z .

9.8 acf.h – complex floating-point numbers

9.8.1 Types, macros and constants

type **acf_struct**

type **acf_t**

An *acf_struct* consists of a pair of *arf_struct*s. An *acf_t* is defined as an array of length one of type *acf_struct*, permitting an *acf_t* to be passed by reference.

type **acf_ptr**

Alias for **acf_struct ***, used for vectors of numbers.

type **acf_srcptr**

Alias for **const acf_struct ***, used for vectors of numbers when passed as constant input to functions.

acf_realref(x)

Macro returning a pointer to the real part of x as an *arf_t*.

acf_imagref(x)

Macro returning a pointer to the imaginary part of x as an *arf_t*.

9.8.2 Memory management

void **acf_init**(*acf_t* x)

Initializes the variable x for use, and sets its value to zero.

void **acf_clear**(*acf_t* x)

Clears the variable x , freeing or recycling its allocated memory.

void **acf_swap**(*acf_t* z, *acf_t* x)

Swaps z and x efficiently.

slong **acf_allocated_bytes**(const *acf_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add **sizeof(acf_struct)** to get the size of the object as a whole.

9.8.3 Basic manipulation

`arf_ptr acf_real_ptr(acf_t z)`

`arf_ptr acf_imag_ptr(acf_t z)`

Returns a pointer to the real or imaginary part of z .

`void acf_set(acf_t z, const acf_t x)`

Sets z to the value x .

`int acf_equal(const acf_t x, const acf_t y)`

Returns whether x and y are equal.

9.8.4 Arithmetic

`int acf_add(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)`

`int acf_sub(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)`

`int acf_mul(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)`

Sets res to the sum, difference or product of x or y , correctly rounding the real and imaginary parts in direction rnd . The return flag has the least significant bit set if the real part is inexact, and the second least significant bit set if the imaginary part is inexact.

9.8.5 Approximate arithmetic

The following operations are *not* correctly rounded. The `rnd` parameter specifies the final direction of rounding, but intermediate roundings are implementation-defined.

`void acf_approx_inv(acf_t res, const acf_t x, slong prec, arf_rnd_t rnd)`

`void acf_approx_div(acf_t res, const acf_t x, const acf_t y, slong prec, arf_rnd_t rnd)`

`void acf_approx_sqrt(acf_t res, const acf_t x, slong prec, arf_rnd_t rnd)`

Computes an approximate inverse, quotient or square root.

`void acf_approx_dot(acf_t res, const acf_t initial, int subtract, acf_srcptr x, slong xstep, acf_srcptr y, slong ystep, slong len, slong prec, arf_rnd_t rnd)`

Computes an approximate dot product, with the same meaning of the parameters as `arb_dot()`.

9.9 arb.h – real numbers

An `arb_t` represents a ball over the real numbers, that is, an interval $[m \pm r] \equiv [m - r, m + r]$ where the midpoint m and the radius r are (extended) real numbers and r is nonnegative (possibly infinite). The result of an (approximate) operation done on `arb_t` variables is a ball which contains the result of the (mathematically exact) operation applied to any choice of points in the input balls. In general, the output ball is not the smallest possible.

The precision parameter passed to each function roughly indicates the precision to which calculations on the midpoint are carried out (operations on the radius are always done using a fixed, small precision.)

For arithmetic operations, the precision parameter currently simply specifies the precision of the corresponding `arf_t` operation. In the future, the arithmetic might be made faster by incorporating sloppy rounding (typically equivalent to a loss of 1-2 bits of effective working precision) when the result is known to be inexact (while still propagating errors rigorously, of course). Arithmetic operations done on exact input with exactly representable output are always guaranteed to produce exact output.

For more complex operations, the precision parameter indicates a minimum working precision (algorithms might allocate extra internal precision to attempt to produce an output accurate to the requested number of bits, especially when the required precision can be estimated easily, but this is not generally required).

If the precision is increased and the inputs either are exact or are computed with increased accuracy as well, the output should converge proportionally, absent any bugs. The general intended strategy for using ball arithmetic is to add a few guard bits, and then repeat the calculation as necessary with an exponentially increasing number of guard bits (Ziv's strategy) until the result is exact enough for one's purposes (typically the first attempt will be successful).

The following balls with an infinite or NaN component are permitted, and may be returned as output from functions.

- The ball $[+\infty \pm c]$, where c is finite, represents the point at positive infinity. Such a ball can always be replaced by $[+\infty \pm 0]$ while preserving mathematical correctness (this is currently not done automatically by the library).
- The ball $[-\infty \pm c]$, where c is finite, represents the point at negative infinity. Such a ball can always be replaced by $[-\infty \pm 0]$ while preserving mathematical correctness (this is currently not done automatically by the library).
- The ball $[c \pm \infty]$, where c is finite or infinite, represents the whole extended real line $[-\infty, +\infty]$. Such a ball can always be replaced by $[0 \pm \infty]$ while preserving mathematical correctness (this is currently not done automatically by the library). Note that there is no way to represent a half-infinite interval such as $[0, \infty]$.
- The ball $[\text{NaN} \pm c]$, where c is finite or infinite, represents an indeterminate value (the value could be any extended real number, or it could represent a function being evaluated outside its domain of definition, for example where the result would be complex). Such an indeterminate ball can always be replaced by $[\text{NaN} \pm \infty]$ while preserving mathematical correctness (this is currently not done automatically by the library).

9.9.1 Types, macros and constants

type **arb_struct**

type **arb_t**

An *arb_struct* consists of an *arf_struct* (the midpoint) and a *mag_struct* (the radius). An *arb_t* is defined as an array of length one of type *arb_struct*, permitting an *arb_t* to be passed by reference.

type **arb_ptr**

Alias for *arb_struct* *, used for vectors of numbers.

type **arb_srcptr**

Alias for *const arb_struct* *, used for vectors of numbers when passed as constant input to functions.

arb_midref(x)

Macro returning a pointer to the midpoint of x as an *arf_t*.

arb_radref(x)

Macro returning a pointer to the radius of x as a *mag_t*.

9.9.2 Memory management

void **arb_init**(*arb_t* x)

Initializes the variable x for use. Its midpoint and radius are both set to zero.

void **arb_clear**(*arb_t* x)

Clears the variable x , freeing or recycling its allocated memory.

arb_ptr **_arb_vec_init**(*slong* n)

Returns a pointer to an array of n initialized *arb_struct* entries.

void **_arb_vec_clear**(*arb_ptr* v, *slong* n)

Clears an array of n initialized *arb_struct* entries.

void **arb_swap**(*arb_t* x, *arb_t* y)

Swaps x and y efficiently.

slong **arb_allocated_bytes**(const *arb_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(arb_struct)` to get the size of the object as a whole.

slong **_arb_vec_allocated_bytes**(*arb_srcptr* vec, *slong* len)

Returns the total number of bytes allocated for this vector, i.e. the space taken up by the vector itself plus the sum of the internal heap allocation sizes for all its member elements.

double **_arb_vec_estimate_allocated_bytes**(*slong* len, *slong* prec)

Estimates the number of bytes that need to be allocated for a vector of len elements with $prec$ bits of precision, including the space for internal limb data. This function returns a *double* to avoid overflow issues when both len and $prec$ are large.

This is only an approximation of the physical memory that will be used by an actual vector. In practice, the space varies with the content of the numbers; for example, zeros and small integers require no internal heap allocation even if the precision is huge. The estimate assumes that exponents will not be bignums. The actual amount may also be higher or lower due to overhead in the memory allocator or overcommitment by the operating system.

9.9.3 Assignment and rounding

void **arb_set**(*arb_t* y, const *arb_t* x)

void **arb_set_arf**(*arb_t* y, const *arf_t* x)

void **arb_set_si**(*arb_t* y, *slong* x)

void **arb_set_ui**(*arb_t* y, *ulong* x)

void **arb_set_fmpz**(*arb_t* y, const *fmpz_t* x)

void **arb_set_d**(*arb_t* y, double x)

Sets y to the value of x without rounding.

Note: Be cautious when using *arb_set_d()* as it does not impose any error bounds and will only convert a *double* to an *arb_t*. For instance, *arb_set_d*(x , 1.1) and *arb_set_str*(x , "1.1", *prec*) work very differently, where the former will first create a *double* whose value is the approximation of 1.1 (without any error bounds) which then sets x to this approximated value with no error. This differs from *arb_set_str* which will impose an error bound based on the precision.

void **arb_set_fmpz_2exp**(*arb_t* y, const *fmpz_t* x, const *fmpz_t* e)

Sets y to $x \cdot 2^e$.

void **arb_set_round**(*arb_t* y, const *arb_t* x, *slong* prec)

void **arb_set_round_fmpz**(*arb_t* y, const *fmpz_t* x, *slong* prec)

Sets *y* to the value of *x*, rounded to *prec* bits in the direction towards zero.

void **arb_set_round_fmpz_2exp**(*arb_t* y, const *fmpz_t* x, const *fmpz_t* e, *slong* prec)

Sets *y* to $x \cdot 2^e$, rounded to *prec* bits in the direction towards zero.

void **arb_set_fmpq**(*arb_t* y, const *fmpq_t* x, *slong* prec)

Sets *y* to the rational number *x*, rounded to *prec* bits in the direction towards zero.

int **arb_set_str**(*arb_t* res, const char *inp, *slong* prec)

Sets *res* to the value specified by the human-readable string *inp*. The input may be a decimal floating-point literal, such as “25”, “0.001”, “7e+141” or “-31.4159e-1”, and may also consist of two such literals separated by the symbol “+/-” and optionally enclosed in brackets, e.g. “[3.25 +/- 0.0001]”, or simply “[+/- 10]” with an implicit zero midpoint. The output is rounded to *prec* bits, and if the binary-to-decimal conversion is inexact, the resulting error is added to the radius.

The symbols “inf” and “nan” are recognized (a nan midpoint results in an indeterminate interval, with infinite radius).

Returns 0 if successful and nonzero if unsuccessful. If unsuccessful, the result is set to an indeterminate interval.

char ***arb_get_str**(const *arb_t* x, *slong* n, *ulong* flags)

Returns a nice human-readable representation of *x*, with at most *n* digits of the midpoint printed.

With default flags, the output can be parsed back with **arb_set_str()**, and this is guaranteed to produce an interval containing the original interval *x*.

By default, the output is rounded so that the value given for the midpoint is correct up to 1 ulp (unit in the last decimal place).

If **ARB_STR_MORE** is added to *flags*, more (possibly incorrect) digits may be printed.

If **ARB_STR_NO_RADIUS** is added to *flags*, the radius is not included in the output. Unless **ARB_STR_MORE** is set, the output is rounded so that the midpoint is correct to 1 ulp. As a special case, if there are no significant digits after rounding, the result will be shown as **0e+n**, meaning that the result is between $-1e+n$ and $1e+n$ (following the contract that the output is correct to within one unit in the only shown digit).

By adding a multiple *m* of **ARB_STR_CONDENSE** to *flags*, strings of more than three times *m* consecutive digits are condensed, only printing the leading and trailing *m* digits along with brackets indicating the number of digits omitted (useful when computing values to extremely high precision).

9.9.4 Assignment of special values

void **arb_zero**(*arb_t* x)

Sets *x* to zero.

void **arb_one**(*arb_t* x)

Sets *x* to the exact integer 1.

void **arb_pos_inf**(*arb_t* x)

Sets *x* to positive infinity, with a zero radius.

void **arb_neg_inf**(*arb_t* x)

Sets *x* to negative infinity, with a zero radius.

void **arb_zero_pm_inf**(*arb_t* x)

Sets *x* to $[0 \pm \infty]$, representing the whole extended real line.

void **arb_indeterminate**(*arb_t* x)

Sets x to $[\text{NaN} \pm \infty]$, representing an indeterminate result.

void **arb_zero_pm_one**(*arb_t* x)

Sets x to the interval $[0 \pm 1]$.

void **arb_unit_interval**(*arb_t* x)

Sets x to the interval $[0, 1]$.

9.9.5 Input and output

The *arb_print...* functions print to standard output, while *arb_fprint...* functions print to the stream *file*.

void **arb_print**(const *arb_t* x)

void **arb_fprint**(FILE *file, const *arb_t* x)

Prints the internal representation of x .

void **arb_printd**(const *arb_t* x, *slong* digits)

void **arb_fprintd**(FILE *file, const *arb_t* x, *slong* digits)

Prints x in decimal. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.

void **arb_printn**(const *arb_t* x, *slong* digits, *ulong* flags)

void **arb_fprintn**(FILE *file, const *arb_t* x, *slong* digits, *ulong* flags)

Prints a nice decimal representation of x . By default, the output shows the midpoint with a guaranteed error of at most one unit in the last decimal place. In addition, an explicit error bound is printed so that the displayed decimal interval is guaranteed to enclose x . See *arb_get_str()* for details.

char ***arb_dump_str**(const *arb_t* x)

Returns a serialized representation of x as a null-terminated ASCII string that can be read by *arb_load_str()*. The format consists of four hexadecimal integers representing the midpoint mantissa, midpoint exponent, radius mantissa and radius exponent (with special values to indicate zero, infinity and NaN values), separated by single spaces. The returned string needs to be deallocated with *flint_free*.

int **arb_load_str**(*arb_t* x, const char *str)

Sets x to the serialized representation given in *str*. Returns a nonzero value if *str* is not formatted correctly (see *arb_dump_str()*).

int **arb_dump_file**(FILE *stream, const *arb_t* x)

Writes a serialized ASCII representation of x to *stream* in a form that can be read by *arb_load_file()*. Returns a nonzero value if the data could not be written.

int **arb_load_file**(*arb_t* x, FILE *stream)

Reads x from a serialized ASCII representation in *stream*. Returns a nonzero value if the data is not formatted correctly or the read failed. Note that the data is assumed to be delimited by a whitespace or end-of-file, i.e., when writing multiple values with *arb_dump_file()* make sure to insert a whitespace to separate consecutive values.

It is possible to serialize and deserialize a vector as follows (warning: without error handling):

```
fp = fopen("data.txt", "w");
for (i = 0; i < n; i++)
{
    arb_dump_file(fp, vec + i);
```

(continues on next page)

(continued from previous page)

```

    fprintf(fp, "\n");    // or any whitespace character
}
fclose(fp);

fp = fopen("data.txt", "r");
for (i = 0; i < n; i++)
{
    arb_load_file(vec + i, fp);
}
fclose(fp);

```

9.9.6 Random number generation

void **arb_randtest**(*arb_t* x, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random ball. The midpoint and radius will both be finite.

void **arb_randtest_exact**(*arb_t* x, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random number with zero radius.

void **arb_randtest_precise**(*arb_t* x, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random number with radius around $2^{-\text{prec}}$ the magnitude of the midpoint.

void **arb_randtest_positive**(*arb_t* x, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random precise number which is guaranteed to be positive.

void **arb_randtest_wide**(*arb_t* x, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random number with midpoint and radius chosen independently, possibly giving a very large interval.

void **arb_randtest_special**(*arb_t* x, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random interval, possibly having NaN or an infinity as the midpoint and possibly having an infinite radius.

void **arb_get_rand_fmpq**(*fmpq_t* q, *flint_rand_t* state, const *arb_t* x, *slong* bits)

Sets q to a random rational number from the interval represented by x . A denominator is chosen by multiplying the binary denominator of x by a random integer up to bits bits.

The outcome is undefined if the midpoint or radius of x is non-finite, or if the exponent of the midpoint or radius is so large or small that representing the endpoints as exact rational numbers would cause overflows.

void **arb_urandom**(*arb_t* x, *flint_rand_t* state, *slong* prec)

Sets x to a uniformly distributed random number in the interval $[0, 1]$. The method uses rounding from integers to floats, hence the radius might not be 0.

9.9.7 Radius and interval operations

void **arb_get_mid_arb**(*arb_t* m, const *arb_t* x)

Sets m to the midpoint of x .

void **arb_get_rad_arb**(*arb_t* r, const *arb_t* x)

Sets r to the radius of x .

void **arb_add_error_arf**(*arb_t* x, const *arf_t* err)

void **arb_add_error_mag**(*arb_t* x, const *mag_t* err)

void **arb_add_error**(*arb_t* x, const *arb_t* err)

Adds the absolute value of *err* to the radius of *x* (the operation is done in-place).

void **arb_add_error_2exp_si**(*arb_t* x, *slong* e)

void **arb_add_error_2exp_fmpz**(*arb_t* x, const *fmpz_t* e)

Adds 2^e to the radius of *x*.

void **arb_union**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets *z* to a ball containing both *x* and *y*.

int **arb_intersection**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

If *x* and *y* overlap according to **arb_overlaps()**, then *z* is set to a ball containing the intersection of *x* and *y* and a nonzero value is returned. Otherwise zero is returned and the value of *z* is undefined. If *x* or *y* contains NaN, the result is NaN.

void **arb_nonnegative_part**(*arb_t* res, const *arb_t* x)

Sets *res* to the intersection of *x* with $[0, \infty]$. If *x* is nonnegative, an exact copy is made. If *x* is finite and contains negative numbers, an interval of the form $[r/2 \pm r/2]$ is produced, which certainly contains no negative points. In the special case when *x* is strictly negative, *res* is set to zero.

void **arb_get_abs_ubound_arf**(*arf_t* u, const *arb_t* x, *slong* prec)

Sets *u* to the upper bound for the absolute value of *x*, rounded up to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_abs_lbound_arf**(*arf_t* u, const *arb_t* x, *slong* prec)

Sets *u* to the lower bound for the absolute value of *x*, rounded down to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_ubound_arf**(*arf_t* u, const *arb_t* x, *slong* prec)

Sets *u* to the upper bound for the value of *x*, rounded up to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_lbound_arf**(*arf_t* u, const *arb_t* x, *slong* prec)

Sets *u* to the lower bound for the value of *x*, rounded down to *prec* bits. If *x* contains NaN, the result is NaN.

void **arb_get_mag**(*mag_t* z, const *arb_t* x)

Sets *z* to an upper bound for the absolute value of *x*. If *x* contains NaN, the result is positive infinity.

void **arb_get_mag_lower**(*mag_t* z, const *arb_t* x)

Sets *z* to a lower bound for the absolute value of *x*. If *x* contains NaN, the result is zero.

void **arb_get_mag_lower_nonnegative**(*mag_t* z, const *arb_t* x)

Sets *z* to a lower bound for the signed value of *x*, or zero if *x* overlaps with the negative half-axis. If *x* contains NaN, the result is zero.

void **arb_get_interval_fmpz_2exp**(*fmpz_t* a, *fmpz_t* b, *fmpz_t* exp, const *arb_t* x)

Computes the exact interval represented by *x*, in the form of an integer interval multiplied by a power of two, i.e. $x = [a, b] \times 2^{\text{exp}}$. The result is normalized by removing common trailing zeros from *a* and *b*.

This method aborts if *x* is infinite or NaN, or if the difference between the exponents of the midpoint and the radius is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if the exponent difference is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that the midpoint and radius of *x* both are within a reasonable range before calling this method.

void **arb_set_interval_mag**(*arb_t* x, const *mag_t* a, const *mag_t* b, *slong* prec)

void **arb_set_interval_arf**(*arb_t* x, const *arf_t* a, const *arf_t* b, *slong* prec)

void **arb_set_interval_mpfr**(*arb_t* x, const *mpfr_t* a, const *mpfr_t* b, *slong* prec)

Sets x to a ball containing the interval $[a, b]$. We require that $a \leq b$.

void **arb_set_interval_neg_pos_mag**(*arb_t* x, const *mag_t* a, const *mag_t* b, *slong* prec)

Sets x to a ball containing the interval $[-a, b]$.

void **arb_get_interval_arf**(*arf_t* a, *arf_t* b, const *arb_t* x, *slong* prec)

void **arb_get_interval_mpfr**(*mpfr_t* a, *mpfr_t* b, const *arb_t* x)

Constructs an interval $[a, b]$ containing the ball x . The MPFR version uses the precision of the output variables.

slong **arb_rel_error_bits**(const *arb_t* x)

Returns the effective relative error of x measured in bits, defined as the difference between the position of the top bit in the radius and the top bit in the midpoint, plus one. The result is clamped between plus/minus *ARF_PREC_EXACT*.

slong **arb_rel_accuracy_bits**(const *arb_t* x)

Returns the effective relative accuracy of x measured in bits, equal to the negative of the return value from *arb_rel_error_bits*().

slong **arb_rel_one_accuracy_bits**(const *arb_t* x)

Given a ball with midpoint m and radius r , returns an approximation of the relative accuracy of $[\max(1, |m|) \pm r]$ measured in bits.

slong **arb_bits**(const *arb_t* x)

Returns the number of bits needed to represent the absolute value of the mantissa of the midpoint of x , i.e. the minimum precision sufficient to represent x exactly. Returns 0 if the midpoint of x is a special value.

void **arb_trim**(*arb_t* y, const *arb_t* x)

Sets y to a trimmed copy of x : rounds x to a number of bits equal to the accuracy of x (as indicated by its radius), plus a few guard bits. The resulting ball is guaranteed to contain x , but is more economical if x has less than full accuracy.

int **arb_get_unique_fmpz**(*fmpz_t* z, const *arb_t* x)

If x contains a unique integer, sets z to that value and returns nonzero. Otherwise (if x represents no integers or more than one integer), returns zero.

This method aborts if there is a unique integer but that integer is so large that allocating memory for the result fails.

Warning: this method will allocate a huge amount of memory to store the result if there is a unique integer and that integer is huge. Memory allocation could succeed even if the required space is far larger than the physical memory available on the machine, resulting in swapping. It is recommended to check that the midpoint of x is within a reasonable range before calling this method.

void **arb_floor**(*arb_t* y, const *arb_t* x, *slong* prec)

void **arb_ceil**(*arb_t* y, const *arb_t* x, *slong* prec)

void **arb_trunc**(*arb_t* y, const *arb_t* x, *slong* prec)

void **arb_nint**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets y to a ball containing respectively, $\lfloor x \rfloor$ and $\lceil x \rceil$, *trunc*(x), *nint*(x), with the midpoint of y rounded to at most *prec* bits.

void **arb_get_fmpz_mid_rad_10exp**(*fmpz_t* mid, *fmpz_t* rad, *fmpz_t* exp, const *arb_t* x, *slong* n)

Assuming that x is finite and not exactly zero, computes integers *mid*, *rad*, *exp* such that $x \in [m - r, m + r] \times 10^e$ and such that the larger out of *mid* and *rad* has at least *n* digits plus a few guard digits. If x is infinite or exactly zero, the outputs are all set to zero.

int **arb_can_round_arf**(const *arb_t* x, *slong* prec, *arf_rnd_t* rnd)

int **arb_can_round_mpfr**(const *arb_t* x, *slong* prec, *mpfr_rnd_t* rnd)

Returns nonzero if rounding the midpoint of x to $prec$ bits in the direction rnd is guaranteed to give the unique correctly rounded floating-point approximation for the real number represented by x .

In other words, if this function returns nonzero, applying *arf_set_round()*, or *arf_get_mpfr()*, or *arf_get_d()* to the midpoint of x is guaranteed to return a correctly rounded *arf_t*, *mpfr_t* (provided that $prec$ is the precision of the output variable), or *double* (provided that $prec$ is 53). Moreover, *arf_get_mpfr()* is guaranteed to return the correct ternary value according to MPFR semantics.

Note that the *mpfr* version of this function takes an MPFR rounding mode symbol as input, while the *arf* version takes an *arf* rounding mode symbol. Otherwise, the functions are identical.

This function may perform a fast, inexact test; that is, it may return zero in some cases even when correct rounding actually is possible.

To be conservative, zero is returned when x is non-finite, even if it is an “exact” infinity.

9.9.8 Comparisons

int **arb_is_zero**(const *arb_t* x)

Returns nonzero iff the midpoint and radius of x are both zero.

int **arb_is_nonzero**(const *arb_t* x)

Returns nonzero iff zero is not contained in the interval represented by x .

int **arb_is_one**(const *arb_t* f)

Returns nonzero iff x is exactly 1.

int **arb_is_finite**(const *arb_t* x)

Returns nonzero iff the midpoint and radius of x are both finite floating-point numbers, i.e. not infinities or NaN.

int **arb_is_exact**(const *arb_t* x)

Returns nonzero iff the radius of x is zero.

int **arb_is_int**(const *arb_t* x)

Returns nonzero iff x is an exact integer.

int **arb_is_int_2exp_si**(const *arb_t* x, *slong* e)

Returns nonzero iff x exactly equals $n2^e$ for some integer n .

int **arb_equal**(const *arb_t* x, const *arb_t* y)

Returns nonzero iff x and y are equal as balls, i.e. have both the same midpoint and radius.

Note that this is not the same thing as testing whether both x and y certainly represent the same real number, unless either x or y is exact (and neither contains NaN). To test whether both operands *might* represent the same mathematical quantity, use *arb_overlaps()* or *arb_contains()*, depending on the circumstance.

int **arb_equal_si**(const *arb_t* x, *slong* y)

Returns nonzero iff x is equal to the integer y .

int **arb_is_positive**(const *arb_t* x)

int **arb_is_nonnegative**(const *arb_t* x)

int **arb_is_negative**(const *arb_t* x)

int **arb_is_nonpositive**(const *arb_t* x)

Returns nonzero iff all points p in the interval represented by x satisfy, respectively, $p > 0$, $p \geq 0$, $p < 0$, $p \leq 0$. If x contains NaN, returns zero.

int **arb_overlaps**(const *arb_t* x, const *arb_t* y)

Returns nonzero iff x and y have some point in common. If either x or y contains NaN, this function always returns nonzero (as a NaN could be anything, it could in particular contain any number that is included in the other operand).

int **arb_contains_arf**(const *arb_t* x, const *arf_t* y)

int **arb_contains_fmpq**(const *arb_t* x, const *fmpq_t* y)

int **arb_contains_fmpz**(const *arb_t* x, const *fmpz_t* y)

int **arb_contains_si**(const *arb_t* x, *slong* y)

int **arb_contains_mpfr**(const *arb_t* x, const *mpfr_t* y)

int **arb_contains**(const *arb_t* x, const *arb_t* y)

Returns nonzero iff the given number (or ball) y is contained in the interval represented by x .

If x contains NaN, this function always returns nonzero (as it could represent anything, and in particular could represent all the points included in y). If y contains NaN and x does not, it always returns zero.

int **arb_contains_int**(const *arb_t* x)

Returns nonzero iff the interval represented by x contains an integer.

int **arb_contains_zero**(const *arb_t* x)

int **arb_contains_negative**(const *arb_t* x)

int **arb_contains_nonpositive**(const *arb_t* x)

int **arb_contains_positive**(const *arb_t* x)

int **arb_contains_nonnegative**(const *arb_t* x)

Returns nonzero iff there is any point p in the interval represented by x satisfying, respectively, $p = 0$, $p < 0$, $p \leq 0$, $p > 0$, $p \geq 0$. If x contains NaN, returns nonzero.

int **arb_contains_interior**(const *arb_t* x, const *arb_t* y)

Tests if y is contained in the interior of x ; that is, contained in x and not touching either endpoint.

int **arb_eq**(const *arb_t* x, const *arb_t* y)

int **arb_ne**(const *arb_t* x, const *arb_t* y)

int **arb_lt**(const *arb_t* x, const *arb_t* y)

int **arb_le**(const *arb_t* x, const *arb_t* y)

int **arb_gt**(const *arb_t* x, const *arb_t* y)

int **arb_ge**(const *arb_t* x, const *arb_t* y)

Respectively performs the comparison $x = y$, $x \neq y$, $x < y$, $x \leq y$, $x > y$, $x \geq y$ in a mathematically meaningful way. If the comparison $t(\text{op})u$ holds for all $t \in x$ and all $u \in y$, returns 1. Otherwise, returns 0.

The balls x and y are viewed as subintervals of the extended real line. Note that balls that are formally different can compare as equal under this definition: for example, $[-\infty \pm 3] = [-\infty \pm 0]$. Also $[-\infty] \leq [\infty \pm \infty]$.

The output is always 0 if either input has NaN as midpoint.

9.9.9 Arithmetic

void **arb_neg**(*arb_t* y, const *arb_t* x)

void **arb_neg_round**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets y to the negation of x .

void **arb_abs**(*arb_t* y, const *arb_t* x)

Sets y to the absolute value of x . No attempt is made to improve the interval represented by x if it contains zero.

void **arb_nonnegative_abs**(*arb_t* y, const *arb_t* x)

Sets y to the absolute value of x . If x is finite and it contains zero, sets y to some interval $[r \pm r]$ that contains the absolute value of x .

void **arb_sgn**(*arb_t* y, const *arb_t* x)

Sets y to the sign function of x . The result is $[0 \pm 1]$ if x contains both zero and nonzero numbers.

int **arb_sgn_nonzero**(const *arb_t* x)

Returns 1 if x is strictly positive, -1 if x is strictly negative, and 0 if x is zero or a ball containing zero so that its sign is not determined.

void **arb_min**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

void **arb_max**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets z respectively to the minimum and the maximum of x and y .

void **arb_minmax**(*arb_t* z1, *arb_t* z2, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets $z1$ and $z2$ respectively to the minimum and the maximum of x and y .

void **arb_add**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

void **arb_add_arf**(*arb_t* z, const *arb_t* x, const *arf_t* y, *slong* prec)

void **arb_add_ui**(*arb_t* z, const *arb_t* x, *ulong* y, *slong* prec)

void **arb_add_si**(*arb_t* z, const *arb_t* x, *slong* y, *slong* prec)

void **arb_add_fmpz**(*arb_t* z, const *arb_t* x, const *fmpz_t* y, *slong* prec)

Sets $z = x + y$, rounded to $prec$ bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_add_fmpz_2exp**(*arb_t* z, const *arb_t* x, const *fmpz_t* m, const *fmpz_t* e, *slong* prec)

Sets $z = x + m \cdot 2^e$, rounded to $prec$ bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_sub**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

void **arb_sub_arf**(*arb_t* z, const *arb_t* x, const *arf_t* y, *slong* prec)

void **arb_sub_ui**(*arb_t* z, const *arb_t* x, *ulong* y, *slong* prec)

void **arb_sub_si**(*arb_t* z, const *arb_t* x, *slong* y, *slong* prec)

void **arb_sub_fmpz**(*arb_t* z, const *arb_t* x, const *fmpz_t* y, *slong* prec)

Sets $z = x - y$, rounded to $prec$ bits. The precision can be *ARF_PREC_EXACT* provided that the result fits in memory.

void **arb_mul**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

void **arb_mul_arf**(*arb_t* z, const *arb_t* x, const *arf_t* y, *slong* prec)

void **arb_mul_si**(*arb_t* z, const *arb_t* x, *slong* y, *slong* prec)

```

void arb_mul_ui(arb_t z, const arb_t x, ulong y, slong prec)

void arb_mul_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)
    Sets  $z = x \cdot y$ , rounded to  $prec$  bits. The precision can be ARF_PREC_EXACT provided that
    the result fits in memory.

void arb_mul_2exp_si(arb_t y, const arb_t x, slong e)

void arb_mul_2exp_fmpz(arb_t y, const arb_t x, const fmpz_t e)
    Sets  $y$  to  $x$  multiplied by  $2^e$ .

void arb_addmul(arb_t z, const arb_t x, const arb_t y, slong prec)

void arb_addmul_arf(arb_t z, const arb_t x, const arf_t y, slong prec)

void arb_addmul_si(arb_t z, const arb_t x, slong y, slong prec)

void arb_addmul_ui(arb_t z, const arb_t x, ulong y, slong prec)

void arb_addmul_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)
    Sets  $z = z + x \cdot y$ , rounded to  $prec$  bits. The precision can be ARF_PREC_EXACT provided that
    the result fits in memory.

void arb_submul(arb_t z, const arb_t x, const arb_t y, slong prec)

void arb_submul_arf(arb_t z, const arb_t x, const arf_t y, slong prec)

void arb_submul_si(arb_t z, const arb_t x, slong y, slong prec)

void arb_submul_ui(arb_t z, const arb_t x, ulong y, slong prec)

void arb_submul_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)
    Sets  $z = z - x \cdot y$ , rounded to  $prec$  bits. The precision can be ARF_PREC_EXACT provided that
    the result fits in memory.

void arb_fma(arb_t res, const arb_t x, const arb_t y, const arb_t z, slong prec)
void arb_fma_arf(arb_t res, const arb_t x, const arf_t y, const arb_t z, slong prec)
void arb_fma_si(arb_t res, const arb_t x, slong y, const arb_t z, slong prec)
void arb_fma_ui(arb_t res, const arb_t x, ulong y, const arb_t z, slong prec)
void arb_fma_fmpz(arb_t res, const arb_t x, const fmpz_t y, const arb_t z, slong prec)
    Sets  $res$  to  $x \cdot y + z$ . This is equivalent to an addmul except that  $res$  and  $z$  can be separate variables.

void arb_inv(arb_t z, const arb_t x, slong prec)
    Sets  $z$  to  $1/x$ .

void arb_div(arb_t z, const arb_t x, const arb_t y, slong prec)

void arb_div_arf(arb_t z, const arb_t x, const arf_t y, slong prec)

void arb_div_si(arb_t z, const arb_t x, slong y, slong prec)

void arb_div_ui(arb_t z, const arb_t x, ulong y, slong prec)

void arb_div_fmpz(arb_t z, const arb_t x, const fmpz_t y, slong prec)

void arb_fmpz_div_fmpz(arb_t z, const fmpz_t x, const fmpz_t y, slong prec)

```

void **arb_ui_div**(*arb_t* z, *ulong* x, const *arb_t* y, *slong* prec)

Sets $z = x/y$, rounded to *prec* bits. If *y* contains zero, *z* is set to $0 \pm \infty$. Otherwise, error propagation uses the rule

$$\left| \frac{x}{y} - \frac{x + \xi_1 a}{y + \xi_2 b} \right| = \left| \frac{x\xi_2 b - y\xi_1 a}{y(y + \xi_2 b)} \right| \leq \frac{|xb| + |ya|}{|y|(|y| - b)}$$

where $-1 \leq \xi_1, \xi_2 \leq 1$, and where the triangle inequality has been applied to the numerator and the reverse triangle inequality has been applied to the denominator.

void **arb_div_2expm1_ui**(*arb_t* z, const *arb_t* x, *ulong* n, *slong* prec)

Sets $z = x/(2^n - 1)$, rounded to *prec* bits.

9.9.10 Dot product

void **arb_dot_precise**(*arb_t* res, const *arb_t* s, int subtract, *arb_srcptr* x, *slong* xstep, *arb_srcptr* y, *slong* ystep, *slong* len, *slong* prec)

void **arb_dot_simple**(*arb_t* res, const *arb_t* s, int subtract, *arb_srcptr* x, *slong* xstep, *arb_srcptr* y, *slong* ystep, *slong* len, *slong* prec)

void **arb_dot**(*arb_t* res, const *arb_t* s, int subtract, *arb_srcptr* x, *slong* xstep, *arb_srcptr* y, *slong* ystep, *slong* len, *slong* prec)

Computes the dot product of the vectors *x* and *y*, setting *res* to $s + (-1)^{\text{subtract}} \sum_{i=0}^{\text{len}-1} x_i y_i$.

The initial term *s* is optional and can be omitted by passing *NULL* (equivalently, $s = 0$). The parameter *subtract* must be 0 or 1. The length *len* is allowed to be negative, which is equivalent to a length of zero. The parameters *xstep* or *ystep* specify a step length for traversing subsequences of the vectors *x* and *y*; either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between *res* and *s* but not between *res* and the entries of *x* and *y*.

The default version determines the optimal precision for each term and performs all internal calculations using mpn arithmetic with minimal overhead. This is the preferred way to compute a dot product; it is generally much faster and more precise than a simple loop.

The *simple* version performs fused multiply-add operations in a simple loop. This can be used for testing purposes and is also used as a fallback by the default version when the exponents are out of range for the optimized code.

The *precise* version computes the dot product exactly up to the final rounding. This can be extremely slow and is only intended for testing.

void **arb_approx_dot**(*arb_t* res, const *arb_t* s, int subtract, *arb_srcptr* x, *slong* xstep, *arb_srcptr* y, *slong* ystep, *slong* len, *slong* prec)

Computes an approximate dot product *without error bounds*. The radii of the inputs are ignored (only the midpoints are read) and only the midpoint of the output is written.

void **arb_dot_ui**(*arb_t* res, const *arb_t* initial, int subtract, *arb_srcptr* x, *slong* xstep, const *ulong* *y, *slong* ystep, *slong* len, *slong* prec)

void **arb_dot_si**(*arb_t* res, const *arb_t* initial, int subtract, *arb_srcptr* x, *slong* xstep, const *slong* *y, *slong* ystep, *slong* len, *slong* prec)

void **arb_dot_uiui**(*arb_t* res, const *arb_t* initial, int subtract, *arb_srcptr* x, *slong* xstep, const *ulong* *y, *slong* ystep, *slong* len, *slong* prec)

void **arb_dot_siui**(*arb_t* res, const *arb_t* initial, int subtract, *arb_srcptr* x, *slong* xstep, const *ulong* *y, *slong* ystep, *slong* len, *slong* prec)

void **arb_dot_fmpz**(*arb_t* res, const *arb_t* initial, int subtract, *arb_srcptr* x, *slong* xstep, const *fmpz* *y, *slong* ystep, *slong* len, *slong* prec)

Equivalent to **arb_dot()**, but with integers in the array *y*. The *uiui* and *siui* versions take an array of double-limb integers as input; the *siui* version assumes that these represent signed integers in two's complement form.

9.9.11 Powers and roots

void **arb_sqrt**(*arb_t* z, const *arb_t* x, *slong* prec)

void **arb_sqrt_arf**(*arb_t* z, const *arf_t* x, *slong* prec)

void **arb_sqrt_fmpz**(*arb_t* z, const *fmpz_t* x, *slong* prec)

void **arb_sqrt_ui**(*arb_t* z, *ulong* x, *slong* prec)

Sets z to the square root of x , rounded to $prec$ bits.

If $x = m \pm x$ where $m \geq r \geq 0$, the propagated error is bounded by $\sqrt{m} - \sqrt{m-r} = \sqrt{m}(1 - \sqrt{1-r/m}) \leq \sqrt{m}(r/m + (r/m)^2)/2$.

void **arb_sqrtpos**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets z to the square root of x , assuming that x represents a nonnegative number (i.e. discarding any negative numbers in the input interval).

void **arb_hypot**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets z to $\sqrt{x^2 + y^2}$.

void **arb_rsqr**(*arb_t* z, const *arb_t* x, *slong* prec)

void **arb_rsqr_ui**(*arb_t* z, *ulong* x, *slong* prec)

Sets z to the reciprocal square root of x , rounded to $prec$ bits. At high precision, this is faster than computing a square root.

void **arb_sqrt1pm1**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \sqrt{1+x} - 1$, computed accurately when $x \approx 0$.

void **arb_root_ui**(*arb_t* z, const *arb_t* x, *ulong* k, *slong* prec)

Sets z to the k -th root of x , rounded to $prec$ bits. This function selects between different algorithms. For large k , it evaluates $\exp(\log(x)/k)$. For small k , it uses **arf_root()** at the midpoint and computes a propagated error bound as follows: if input interval is $[m-r, m+r]$ with $r \leq m$, the error is largest at $m-r$ where it satisfies

$$\begin{aligned} m^{1/k} - (m-r)^{1/k} &= m^{1/k} [1 - (1-r/m)^{1/k}] \\ &= m^{1/k} [1 - \exp(\log(1-r/m)/k)] \\ &\leq m^{1/k} \min(1, -\log(1-r/m)/k) \\ &= m^{1/k} \min(1, \log(1+r/(m-r))/k). \end{aligned}$$

This is evaluated using **mag_log1p()**.

void **arb_root**(*arb_t* z, const *arb_t* x, *ulong* k, *slong* prec)

Alias for **arb_root_ui()**, provided for backwards compatibility.

void **arb_sqr**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets y to be the square of x .

void **arb_pow_fmpz_binexp**(*arb_t* y, const *arb_t* b, const *fmpz_t* e, *slong* prec)

void **arb_pow_fmpz**(*arb_t* y, const *arb_t* b, const *fmpz_t* e, *slong* prec)

void **arb_pow_ui**(*arb_t* y, const *arb_t* b, *ulong* e, *slong* prec)

void **arb_ui_pow_ui**(*arb_t* y, *ulong* b, *ulong* e, *slong* prec)

void **arb_si_pow_ui**(*arb_t* y, *slong* b, *ulong* e, *slong* prec)

Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Provided that b and e are small enough and the exponent is positive, the exact power can be computed by setting the precision to `ARF_PREC_EXACT`.

Note that these functions can get slow if the exponent is extremely large (in such cases `arb_pow()` may be superior).

void **arb_pow_fmpq**(*arb_t* y, const *arb_t* x, const *fmpq_t* a, *slong* prec)

Sets $y = b^e$, computed as $y = (b^{1/q})^p$ if the denominator of $e = p/q$ is small, and generally as $y = \exp(e \log b)$.

Note that this function can get slow if the exponent is extremely large (in such cases `arb_pow()` may be superior).

void **arb_pow**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets $z = x^y$, computed using binary exponentiation if y is a small exact integer, as $z = (x^{1/2})^{2y}$ if y is a small exact half-integer, and generally as $z = \exp(y \log x)$, except giving the obvious finite result if x is $a \pm a$ and y is positive.

9.9.12 Exponentials and logarithms

void **arb_log_ui**(*arb_t* z, *ulong* x, *slong* prec)

void **arb_log_fmpz**(*arb_t* z, const *fmpz_t* x, *slong* prec)

void **arb_log_arf**(*arb_t* z, const *arf_t* x, *slong* prec)

void **arb_log**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \log(x)$.

At low to medium precision (up to about 4096 bits), `arb_log_arf()` uses table-based argument reduction and fast Taylor series evaluation via `_arb_atan_taylor_rs()`. At high precision, it falls back to MPFR. The function `arb_log()` simply calls `arb_log_arf()` with the midpoint as input, and separately adds the propagated error.

void **arb_log_ui_from_prev**(*arb_t* log_k1, *ulong* k1, *arb_t* log_k0, *ulong* k0, *slong* prec)

Computes $\log(k_1)$, given $\log(k_0)$ where $k_0 < k_1$. At high precision, this function uses the formula $\log(k_1) = \log(k_0) + 2 \operatorname{atanh}((k_1 - k_0)/(k_1 + k_0))$, evaluating the inverse hyperbolic tangent using binary splitting (for best efficiency, k_0 should be large and $k_1 - k_0$ should be small). Otherwise, it ignores $\log(k_0)$ and evaluates the logarithm the usual way.

void **arb_log1p**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \log(1 + x)$, computed accurately when $x \approx 0$.

void **arb_log_base_ui**(*arb_t* res, const *arb_t* x, *ulong* b, *slong* prec)

Sets res to $\log_b(x)$. The result is computed exactly when possible.

void **arb_log_hypot**(*arb_t* res, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets res to $\log(\sqrt{x^2 + y^2})$.

void **arb_exp**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \exp(x)$. Error propagation is done using the following rule: assuming $x = m \pm r$, the error is largest at $m + r$, and we have $\exp(m + r) - \exp(m) = \exp(m)(\exp(r) - 1) \leq r \exp(m + r)$.

void **arb_exp1**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \exp(x) - 1$, using a more accurate method when $x \approx 0$.

void **arb_exp_invexp**(*arb_t* z, *arb_t* w, const *arb_t* x, *slong* prec)

Sets $z = \exp(x)$ and $w = \exp(-x)$. The second exponential is computed from the first using a division, but propagated error bounds are computed separately.

9.9.13 Trigonometric functions

void **arb_sin**(*arb_t* s, const *arb_t* x, *slong* prec)

void **arb_cos**(*arb_t* c, const *arb_t* x, *slong* prec)

void **arb_sin_cos**(*arb_t* s, *arb_t* c, const *arb_t* x, *slong* prec)

Sets $s = \sin(x)$, $c = \cos(x)$.

void **arb_sin_pi**(*arb_t* s, const *arb_t* x, *slong* prec)

void **arb_cos_pi**(*arb_t* c, const *arb_t* x, *slong* prec)

void **arb_sin_cos_pi**(*arb_t* s, *arb_t* c, const *arb_t* x, *slong* prec)

Sets $s = \sin(\pi x)$, $c = \cos(\pi x)$.

void **arb_tan**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets $y = \tan(x) = \sin(x)/\cos(x)$.

void **arb_cot**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets $y = \cot(x) = \cos(x)/\sin(x)$.

void **arb_sin_cos_pi_fmpq**(*arb_t* s, *arb_t* c, const *fmpq_t* x, *slong* prec)

void **arb_sin_pi_fmpq**(*arb_t* s, const *fmpq_t* x, *slong* prec)

void **arb_cos_pi_fmpq**(*arb_t* c, const *fmpq_t* x, *slong* prec)

Sets $s = \sin(\pi x)$, $c = \cos(\pi x)$ where x is a rational number (whose numerator and denominator are assumed to be reduced). We first use trigonometric symmetries to reduce the argument to the octant $[0, 1/4]$. Then we either multiply by a numerical approximation of π and evaluate the trigonometric function the usual way, or we use algebraic methods, depending on which is estimated to be faster. Since the argument has been reduced to the first octant, the first of these two methods gives full accuracy even if the original argument is close to some root other the origin.

void **arb_tan_pi**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets $y = \tan(\pi x)$.

void **arb_cot_pi**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets $y = \cot(\pi x)$.

void **arb_sec**(*arb_t* res, const *arb_t* x, *slong* prec)

Computes $\sec(x) = 1/\cos(x)$.

void **arb_csc**(*arb_t* res, const *arb_t* x, *slong* prec)

Computes $\csc(x) = 1/\sin(x)$.

void **arb_csc_pi**(*arb_t* res, const *arb_t* x, *slong* prec)

Computes $\csc(\pi x) = 1/\sin(\pi x)$.

void **arb_sinc**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \text{sinc}(x) = \sin(x)/x$.

void **arb_sinc_pi**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \text{sinc}(\pi x) = \sin(\pi x)/(\pi x)$.

9.9.14 Inverse trigonometric functions

void **arb_atan_arf**(*arb_t* z, const *arf_t* x, *slong* prec)

void **arb_atan**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \operatorname{atan}(x)$.

At low to medium precision (up to about 4096 bits), **arb_atan_arf()** uses table-based argument reduction and fast Taylor series evaluation via **_arb_atan_taylor_rs()**. At high precision, it falls back to MPFR. The function **arb_atan()** simply calls **arb_atan_arf()** with the midpoint as input, and separately adds the propagated error.

The function **arb_atan_arf()** uses lookup tables if possible, and otherwise falls back to **arb_atan_arf_bb()**.

void **arb_atan2**(*arb_t* z, const *arb_t* b, const *arb_t* a, *slong* prec)

Sets r to an the argument (phase) of the complex number $a + bi$, with the branch cut discontinuity on $(-\infty, 0]$. We define $\operatorname{atan2}(0, 0) = 0$, and for $a < 0$, $\operatorname{atan2}(0, a) = \pi$.

void **arb_asin**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \operatorname{asin}(x) = \operatorname{atan}(x/\sqrt{1-x^2})$. If x is not contained in the domain $[-1, 1]$, the result is an indeterminate interval.

void **arb_acos**(*arb_t* z, const *arb_t* x, *slong* prec)

Sets $z = \operatorname{acos}(x) = \pi/2 - \operatorname{asin}(x)$. If x is not contained in the domain $[-1, 1]$, the result is an indeterminate interval.

9.9.15 Hyperbolic functions

void **arb_sinh**(*arb_t* s, const *arb_t* x, *slong* prec)

void **arb_cosh**(*arb_t* c, const *arb_t* x, *slong* prec)

void **arb_sinh_cosh**(*arb_t* s, *arb_t* c, const *arb_t* x, *slong* prec)

Sets $s = \sinh(x)$, $c = \cosh(x)$. If the midpoint of x is close to zero and the hyperbolic sine is to be computed, evaluates $(e^{2x} \pm 1)/(2e^x)$ via **arb_exp1()** to avoid loss of accuracy. Otherwise evaluates $(e^x \pm e^{-x})/2$.

void **arb_tanh**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets $y = \tanh(x) = \sinh(x)/\cosh(x)$, evaluated via **arb_exp1()** as $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$ if $|x|$ is small, and as $\tanh(\pm x) = 1 - 2e^{\mp 2x}/(1 + e^{\mp 2x})$ if $|x|$ is large.

void **arb_coth**(*arb_t* y, const *arb_t* x, *slong* prec)

Sets $y = \operatorname{coth}(x) = \cosh(x)/\sinh(x)$, evaluated using the same strategy as **arb_tanh()**.

void **arb_sech**(*arb_t* res, const *arb_t* x, *slong* prec)

Computes $\operatorname{sech}(x) = 1/\cosh(x)$.

void **arb_csch**(*arb_t* res, const *arb_t* x, *slong* prec)

Computes $\operatorname{csch}(x) = 1/\sinh(x)$.

9.9.16 Inverse hyperbolic functions

void **arb_atanh**(*arb_t* z, const *arb_t* x, *slong* prec)
 Sets $z = \operatorname{atanh}(x)$.

void **arb_asinh**(*arb_t* z, const *arb_t* x, *slong* prec)
 Sets $z = \operatorname{asinh}(x)$.

void **arb_acosh**(*arb_t* z, const *arb_t* x, *slong* prec)
 Sets $z = \operatorname{acosh}(x)$. If $x < 1$, the result is an indeterminate interval.

9.9.17 Constants

The following functions cache the computed values to speed up repeated calls at the same or lower precision. For further implementation details, see *Algorithms for mathematical constants*.

void **arb_const_pi**(*arb_t* z, *slong* prec)
 Computes π .

void **arb_const_sqrt_pi**(*arb_t* z, *slong* prec)
 Computes $\sqrt{\pi}$.

void **arb_const_log_sqrt2pi**(*arb_t* z, *slong* prec)
 Computes $\log \sqrt{2\pi}$.

void **arb_const_log2**(*arb_t* z, *slong* prec)
 Computes $\log(2)$.

void **arb_const_log10**(*arb_t* z, *slong* prec)
 Computes $\log(10)$.

void **arb_const_euler**(*arb_t* z, *slong* prec)
 Computes Euler's constant $\gamma = \lim_{k \rightarrow \infty} (H_k - \log k)$ where $H_k = 1 + 1/2 + \dots + 1/k$.

void **arb_const_catalan**(*arb_t* z, *slong* prec)
 Computes Catalan's constant $C = \sum_{n=0}^{\infty} (-1)^n / (2n+1)^2$.

void **arb_const_e**(*arb_t* z, *slong* prec)
 Computes $e = \exp(1)$.

void **arb_const_khinchin**(*arb_t* z, *slong* prec)
 Computes Khinchin's constant K_0 .

void **arb_const_glaisher**(*arb_t* z, *slong* prec)
 Computes the Glaisher-Kinkelin constant $A = \exp(1/12 - \zeta'(-1))$.

void **arb_const_apery**(*arb_t* z, *slong* prec)
 Computes Apery's constant $\zeta(3)$.

void **arb_const_reciprocal_fibonacci**(*arb_t* z, *slong* prec)
 Computes the reciprocal Fibonacci constant $\sum_{n=1}^{\infty} 1/F_n$.

9.9.18 Lambert W function

void **arb_lambertw**(*arb_t* res, const *arb_t* x, int flags, *slong* prec)

Computes the Lambert W function, which solves the equation $we^w = x$.

The Lambert W function has infinitely many complex branches $W_k(x)$, two of which are real on a part of the real line. The principal branch $W_0(x)$ is selected by setting *flags* to 0, and the W_{-1} branch is selected by setting *flags* to 1. The principal branch is real-valued for $x \geq -1/e$ (taking values in $[-1, +\infty)$) and the W_{-1} branch is real-valued for $-1/e \leq x < 0$ and takes values in $(-\infty, -1]$. Elsewhere, the Lambert W function is complex and **acb_lambertw**() should be used.

The implementation first computes a floating-point approximation heuristically and then computes a rigorously certified enclosure around this approximation. Some asymptotic cases are handled specially. The algorithm used to compute the Lambert W function is described in [Joh2017b], which follows the main ideas in [CGHJK1996].

9.9.19 Gamma function and factorials

void **arb_rising_ui**(*arb_t* z, const *arb_t* x, *ulong* n, *slong* prec)

void **arb_rising**(*arb_t* z, const *arb_t* x, const *arb_t* n, *slong* prec)

Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$. These functions are aliases for **arb_hypgeom_rising_ui**() and **arb_hypgeom_rising**() .

void **arb_rising_fmpq_ui**(*arb_t* z, const *fmpq_t* x, *ulong* n, *slong* prec)

Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$ using binary splitting. If the denominator or numerator of x is large compared to *prec*, it is more efficient to convert x to an approximation and use **arb_rising_ui**() .

void **arb_rising2_ui**(*arb_t* u, *arb_t* v, const *arb_t* x, *ulong* n, *slong* prec)

Letting $u(x) = x(x+1)(x+2)\cdots(x+n-1)$, simultaneously compute $u(x)$ and $v(x) = u'(x)$. This function is a wrapper of **arb_hypgeom_rising_ui_jet**() .

void **arb_fac_ui**(*arb_t* z, *ulong* n, *slong* prec)

Computes the factorial $z = n!$ via the gamma function.

void **arb_doublefac_ui**(*arb_t* z, *ulong* n, *slong* prec)

Computes the double factorial $z = n!!$ via the gamma function.

void **arb_bin_ui**(*arb_t* z, const *arb_t* n, *ulong* k, *slong* prec)

void **arb_bin_uiui**(*arb_t* z, *ulong* n, *ulong* k, *slong* prec)

Computes the binomial coefficient $z = \binom{n}{k}$, via the rising factorial as $\binom{n}{k} = (n-k+1)_k/k!$.

void **arb_gamma**(*arb_t* z, const *arb_t* x, *slong* prec)

void **arb_gamma_fmpq**(*arb_t* z, const *fmpq_t* x, *slong* prec)

void **arb_gamma_fmpz**(*arb_t* z, const *fmpz_t* x, *slong* prec)

Computes the gamma function $z = \Gamma(x)$.

These functions are aliases for **arb_hypgeom_gamma**(), **arb_hypgeom_gamma_fmpq**(), **arb_hypgeom_gamma_fmpz**() .

void **arb_lgamma**(*arb_t* z, const *arb_t* x, *slong* prec)

Computes the logarithmic gamma function $z = \log \Gamma(x)$. The complex branch structure is assumed, so if $x \leq 0$, the result is an indeterminate interval. This function is an alias for **arb_hypgeom_lgamma**() .

void **arb_rgamma**(*arb_t* z, const *arb_t* x, *slong* prec)

Computes the reciprocal gamma function $z = 1/\Gamma(x)$, avoiding division by zero at the poles of the gamma function. This function is an alias for **arb_hypgeom_rgamma**() .

void **arb_digamma**(*arb_t* y, const *arb_t* x, *slong* prec)

Computes the digamma function $z = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x)$.

9.9.20 Zeta function

void **arb_zeta_ui_vec_borwein**(*arb_ptr* z, *ulong* start, *slong* num, *ulong* step, *slong* prec)

Evaluates $\zeta(s)$ at num consecutive integers s beginning with *start* and proceeding in increments of *step*. Uses Borwein's formula ([Bor2000], [GS2003]), implemented to support fast multi-evaluation (but also works well for a single s).

Requires $\text{start} \geq 2$. For efficiency, the largest s should be at most about as large as *prec*. Arguments approaching *LONG_MAX* will cause overflows. One should therefore only use this function for s up to about *prec*, and then switch to the Euler product.

The algorithm for single s is basically identical to the one used in MPFR (see [MPFR2012] for a detailed description). In particular, we evaluate the sum backwards to avoid storing more than one d_k coefficient, and use integer arithmetic throughout since it is convenient and the terms turn out to be slightly larger than 2^{prec} . The only numerical error in the main loop comes from the division by k^s , which adds less than 1 unit of error per term. For fast multi-evaluation, we repeatedly divide by k^{step} . Each division reduces the input error and adds at most 1 unit of additional rounding error, so by induction, the error per term is always smaller than 2 units.

void **arb_zeta_ui_asymp**(*arb_t* x, *ulong* s, *slong* prec)

void **arb_zeta_ui_euler_product**(*arb_t* z, *ulong* s, *slong* prec)

Computes $\zeta(s)$ using the Euler product. This is fast only if s is large compared to the precision. Both methods are trivial wrappers for `_acb_dirichlet_euler_product_real_ui()`.

void **arb_zeta_ui_bernoulli**(*arb_t* x, *ulong* s, *slong* prec)

Computes $\zeta(s)$ for even s via the corresponding Bernoulli number.

void **arb_zeta_ui_borwein_bsplitt**(*arb_t* x, *ulong* s, *slong* prec)

Computes $\zeta(s)$ for arbitrary $s \geq 2$ using a binary splitting implementation of Borwein's algorithm. This has quasilinear complexity with respect to the precision (assuming that s is fixed).

void **arb_zeta_ui_vec**(*arb_ptr* x, *ulong* start, *slong* num, *slong* prec)

void **arb_zeta_ui_vec_even**(*arb_ptr* x, *ulong* start, *slong* num, *slong* prec)

void **arb_zeta_ui_vec_odd**(*arb_ptr* x, *ulong* start, *slong* num, *slong* prec)

Computes $\zeta(s)$ at num consecutive integers (respectively num even or num odd integers) beginning with $s = \text{start} \geq 2$, automatically choosing an appropriate algorithm.

void **arb_zeta_ui**(*arb_t* x, *ulong* s, *slong* prec)

Computes $\zeta(s)$ for nonnegative integer $s \neq 1$, automatically choosing an appropriate algorithm. This function is intended for numerical evaluation of isolated zeta values; for multi-evaluation, the vector versions are more efficient.

void **arb_zeta**(*arb_t* z, const *arb_t* s, *slong* prec)

Sets z to the value of the Riemann zeta function $\zeta(s)$.

For computing derivatives with respect to s , use `arb_poly_zeta_series()`.

void **arb_hurwitz_zeta**(*arb_t* z, const *arb_t* s, const *arb_t* a, *slong* prec)

Sets z to the value of the Hurwitz zeta function $\zeta(s, a)$.

For computing derivatives with respect to s , use `arb_poly_zeta_series()`.

9.9.21 Bernoulli numbers and polynomials

void **arb_bernoulli_ui**(*arb_t* b, *ulong* n, *slong* prec)

void **arb_bernoulli_fmpz**(*arb_t* b, const *fmpz_t* n, *slong* prec)

Sets b to the numerical value of the Bernoulli number B_n approximated to $prec$ bits.

The internal precision is increased automatically to give an accurate result. Note that, with huge *fmpz* input, the output will have a huge exponent and evaluation will accordingly be slower.

A single division from the exact fraction of B_n is used if this value is in the global cache or the exact numerator roughly is larger than $prec$ bits. Otherwise, the Riemann zeta function is used (see *arb_bernoulli_ui_zeta()*).

This function reads B_n from the global cache if the number is already cached, but does not automatically extend the cache by itself.

void **arb_bernoulli_ui_zeta**(*arb_t* b, *ulong* n, *slong* prec)

Sets b to the numerical value of B_n accurate to $prec$ bits, computed using the formula $B_{2n} = (-1)^{n+1} 2(2n)! \zeta(2n) / (2\pi)^n$.

To avoid potential infinite recursion, we explicitly call the Euler product implementation of the zeta function. This method will only give high accuracy if the precision is small enough compared to n for the Euler product to converge rapidly.

void **arb_bernoulli_poly_ui**(*arb_t* res, *ulong* n, const *arb_t* x, *slong* prec)

Sets res to the value of the Bernoulli polynomial $B_n(x)$.

Warning: this function is only fast if either n or x is a small integer.

This function reads Bernoulli numbers from the global cache if they are already cached, but does not automatically extend the cache by itself.

void **arb_power_sum_vec**(*arb_ptr* res, const *arb_t* a, const *arb_t* b, *slong* len, *slong* prec)

For n from 0 to $len - 1$, sets entry n in the output vector res to

$$S_n(a, b) = \frac{1}{n+1} (B_{n+1}(b) - B_{n+1}(a))$$

where $B_n(x)$ is a Bernoulli polynomial. If a and b are integers and $b \geq a$, this is equivalent to

$$S_n(a, b) = \sum_{k=a}^{b-1} k^n.$$

The computation uses the generating function for Bernoulli polynomials.

9.9.22 Polylogarithms

void **arb_polylog**(*arb_t* w, const *arb_t* s, const *arb_t* z, *slong* prec)

void **arb_polylog_si**(*arb_t* w, *slong* s, const *arb_t* z, *slong* prec)

Sets w to the polylogarithm $\text{Li}_s(z)$.

9.9.23 Other special functions

void **arb_fib_fmpz**(*arb_t* z, const *fmpz_t* n, *slong* prec)

void **arb_fib_ui**(*arb_t* z, *ulong* n, *slong* prec)

Computes the Fibonacci number F_n using binary squaring.

void **arb_agm**(*arb_t* z, const *arb_t* x, const *arb_t* y, *slong* prec)

Sets z to the arithmetic-geometric mean of x and y .

void **arb_chebyshev_t_ui**(*arb_t* a, *ulong* n, const *arb_t* x, *slong* prec)

void **arb_chebyshev_u_ui**(*arb_t* a, *ulong* n, const *arb_t* x, *slong* prec)

Evaluates the Chebyshev polynomial of the first kind $a = T_n(x)$ or the Chebyshev polynomial of the second kind $a = U_n(x)$.

void **arb_chebyshev_t2_ui**(*arb_t* a, *arb_t* b, *ulong* n, const *arb_t* x, *slong* prec)

void **arb_chebyshev_u2_ui**(*arb_t* a, *arb_t* b, *ulong* n, const *arb_t* x, *slong* prec)

Simultaneously evaluates $a = T_n(x), b = T_{n-1}(x)$ or $a = U_n(x), b = U_{n-1}(x)$. Aliasing between a, b and x is not permitted.

void **arb_bell_sum_bsplitt**(*arb_t* res, const *fmpz_t* n, const *fmpz_t* a, const *fmpz_t* b, const *fmpz_t* mmag, *slong* prec)

void **arb_bell_sum_taylor**(*arb_t* res, const *fmpz_t* n, const *fmpz_t* a, const *fmpz_t* b, const *fmpz_t* mmag, *slong* prec)

Helper functions for Bell numbers, evaluating the sum $\sum_{k=a}^{b-1} k^n/k!$. If $mmag$ is non-NULL, it may be used to indicate that the target error tolerance should be $2^{mmag-prec}$.

void **arb_bell_fmpz**(*arb_t* res, const *fmpz_t* n, *slong* prec)

void **arb_bell_ui**(*arb_t* res, *ulong* n, *slong* prec)

Sets res to the Bell number B_n . If the number is too large to fit exactly in $prec$ bits, a numerical approximation is computed efficiently.

The algorithm to compute Bell numbers, including error analysis, is described in detail in [Joh2015].

void **arb_euler_number_fmpz**(*arb_t* res, const *fmpz_t* n, *slong* prec)

void **arb_euler_number_ui**(*arb_t* res, *ulong* n, *slong* prec)

Sets res to the Euler number E_n , which is defined by the exponential generating function $1/\cosh(x)$. The result will be exact if E_n is exactly representable at the requested precision.

void **arb_fmpz_euler_number_ui_multi_mod**(*fmpz_t* res, *ulong* n, double alpha)

void **arb_fmpz_euler_number_ui**(*fmpz_t* res, *ulong* n)

Computes the Euler number E_n as an exact integer. The default algorithm uses a table lookup, the Dirichlet beta function or a hybrid modular algorithm depending on the size of n . The *multi_mod* algorithm accepts a tuning parameter $alpha$ which can be set to a negative value to use defaults.

void **arb_partitions_fmpz**(*arb_t* res, const *fmpz_t* n, *slong* prec)

void **arb_partitions_ui**(*arb_t* res, *ulong* n, *slong* prec)

Sets res to the partition function $p(n)$. When n is large and $\log_2 p(n)$ is more than twice $prec$, the leading term in the Hardy-Ramanujan asymptotic series is used together with an error bound. Otherwise, the exact value is computed and rounded.

void **arb_primorial_nth_ui**(*arb_t* res, *ulong* n, *slong* prec)

Sets res to the n th primorial, defined as the product of the first n prime numbers. The running time is quasilinear in n .

void **arb_primorial_ui**(*arb_t* res, *ulong* n, *slong* prec)

Sets res to the primorial defined as the product of the positive integers up to and including n . The running time is quasilinear in n .

9.9.24 Internals for computing elementary functions

void `_arb_atan_taylor_naive`(*nn_ptr* y, *ulong* *error, *nn_srcptr* x, *slong* xn, *ulong* N, int alternating)

void `_arb_atan_taylor_rs`(*nn_ptr* y, *ulong* *error, *nn_srcptr* x, *slong* xn, *ulong* N, int alternating)

Computes an approximation of $y = \sum_{k=0}^{N-1} x^{2k+1}/(2k+1)$ (if *alternating* is 0) or $y = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)$ (if *alternating* is 1). Used internally for computing arctangents and logarithms. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \leq 255$, $0 \leq x \leq 1/16$, and *xn* positive. The input *x* and output *y* are fixed-point numbers with *xn* fractional limbs. A bound for the ulp error is written to *error*.

void `_arb_exp_taylor_naive`(*nn_ptr* y, *ulong* *error, *nn_srcptr* x, *slong* xn, *ulong* N)

void `_arb_exp_taylor_rs`(*nn_ptr* y, *ulong* *error, *nn_srcptr* x, *slong* xn, *ulong* N)

Computes an approximation of $y = \sum_{k=0}^{N-1} x^k/k!$. Used internally for computing exponentials. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \leq 287$, $0 \leq x \leq 1/16$, and *xn* positive. The input *x* is a fixed-point number with *xn* fractional limbs, and the output *y* is a fixed-point number with *xn* fractional limbs plus one extra limb for the integer part of the result.

A bound for the ulp error is written to *error*.

void `_arb_sin_cos_taylor_naive`(*nn_ptr* ysin, *nn_ptr* ycos, *ulong* *error, *nn_srcptr* x, *slong* xn, *ulong* N)

void `_arb_sin_cos_taylor_rs`(*nn_ptr* ysin, *nn_ptr* ycos, *ulong* *error, *nn_srcptr* x, *slong* xn, *ulong* N, int *sinonly*, int *alternating*)

Computes approximations of $y_s = \sum_{k=0}^{N-1} (-1)^k x^{2k+1}/(2k+1)!$ and $y_c = \sum_{k=0}^{N-1} (-1)^k x^{2k}/(2k)!$. Used internally for computing sines and cosines. The *naive* version uses the forward recurrence, and the *rs* version uses a division-avoiding rectangular splitting scheme.

Requires $N \leq 143$, $0 \leq x \leq 1/16$, and *xn* positive. The input *x* and outputs *ysin*, *ycos* are fixed-point numbers with *xn* fractional limbs. A bound for the ulp error is written to *error*.

If *sinonly* is 1, only the sine is computed; if *sinonly* is 0 both the sine and cosine are computed. To compute sin and cos, *alternating* should be 1. If *alternating* is 0, the hyperbolic sine is computed (this is currently only intended to be used together with *sinonly*).

int `_arb_get_mpn_fixed_mod_log2`(*nn_ptr* w, *fmpz_t* q, *ulong* *error, const *arf_t* x, *slong* wn)

Attempts to write $w = x - q \log(2)$ with $0 \leq w < \log(2)$, where *w* is a fixed-point number with *wn* limbs and ulp error *error*. Returns success.

int `_arb_get_mpn_fixed_mod_pi4`(*nn_ptr* w, *fmpz_t* q, int *octant, *ulong* *error, const *arf_t* x, *slong* wn)

Attempts to write $w = |x| - q\pi/4$ with $0 \leq w < \pi/4$, where *w* is a fixed-point number with *wn* limbs and ulp error *error*. Returns success.

The value of $q \bmod 8$ is written to *octant*. The output variable *q* can be NULL, in which case the full value of *q* is not stored.

slong `_arb_exp_taylor_bound`(*slong* mag, *slong* prec)

Returns *n* such that $|\sum_{k=n}^{\infty} x^k/k!| \leq 2^{-\text{prec}}$, assuming $|x| \leq 2^{\text{mag}} \leq 1/4$.

void `arb_exp_arf_bb`(*arf_t* z, const *arf_t* x, *slong* prec, int m1)

Computes the exponential function using the bit-burst algorithm. If *m1* is nonzero, the exponential function minus one is computed accurately.

Aborts if x is extremely small or large (where another algorithm should be used).

For large x , repeated halving is used. In fact, we always do argument reduction until $|x|$ is smaller than about 2^{-d} where $d \approx 16$ to speed up convergence. If $|x| \approx 2^m$, we thus need about $m + d$ squarings.

Computing $\log(2)$ costs roughly 100-200 multiplications, so is not usually worth the effort at very high precision. However, this function could be improved by using $\log(2)$ based reduction at precision low enough that the value can be assumed to be cached.

```
void _arb_exp_sum_bs_simple(fmpz_t T, fmpz_t Q, flint_bitcnt_t *Qexp, const fmpz_t x,
                          flint_bitcnt_t r, slong N)
```

```
void _arb_exp_sum_bs_powtab(fmpz_t T, fmpz_t Q, flint_bitcnt_t *Qexp, const fmpz_t x,
                          flint_bitcnt_t r, slong N)
```

Computes T , Q and $Qexp$ such that $T/(Q2^{Qexp}) = \sum_{k=1}^N (x/2^r)^k/k!$ using binary splitting. Note that the sum is taken to N inclusive and omits the constant term.

The *powtab* version precomputes a table of powers of x , resulting in slightly higher memory usage but better speed. For best efficiency, N should have many trailing zero bits.

```
void arb_exp_arf_rs_generic(arb_t res, const arf_t x, slong prec, int minus_one)
```

Computes the exponential function using a generic version of the rectangular splitting strategy, intended for intermediate precision.

```
void _arb_atan_sum_bs_simple(fmpz_t T, fmpz_t Q, flint_bitcnt_t *Qexp, const fmpz_t x,
                          flint_bitcnt_t r, slong N)
```

```
void _arb_atan_sum_bs_powtab(fmpz_t T, fmpz_t Q, flint_bitcnt_t *Qexp, const fmpz_t x,
                          flint_bitcnt_t r, slong N)
```

Computes T , Q and $Qexp$ such that $T/(Q2^{Qexp}) = \sum_{k=1}^N (-1)^k (x/2^r)^{2k}/(2k+1)$ using binary splitting. Note that the sum is taken to N inclusive, omits the linear term, and requires a final multiplication by $(x/2^r)$ to give the true series for atan.

The *powtab* version precomputes a table of powers of x , resulting in slightly higher memory usage but better speed. For best efficiency, N should have many trailing zero bits.

```
void arb_atan_arf_bb(arb_t z, const arf_t x, slong prec)
```

Computes the arctangent of x . Initially, the argument-halving formula

$$\operatorname{atan}(x) = 2 \operatorname{atan}\left(\frac{x}{1 + \sqrt{1 + x^2}}\right)$$

is applied up to 8 times to get a small argument. Then a version of the bit-burst algorithm is used. The functional equation

$$\operatorname{atan}(x) = \operatorname{atan}(p/q) + \operatorname{atan}(w), \quad w = \frac{qx - p}{px + q}, \quad p = \lfloor qx \rfloor$$

is applied repeatedly instead of integrating a differential equation for the arctangent, as this appears to be more efficient.

```
void arb_atan_frac_bsplitt(arb_t s, const fmpz_t p, const fmpz_t q, int hyperbolic, slong prec)
```

Computes the arctangent of p/q , optionally the hyperbolic arctangent, using direct series summation with binary splitting.

```
void arb_sin_cos_arf_generic(arb_t s, arb_t c, const arf_t x, slong prec)
```

Computes the sine and cosine of x using a generic strategy. This function gets called internally by the main sin and cos functions when the precision for argument reduction or series evaluation based on lookup tables is exhausted.

This function first performs a cheap test to see if $|x| < \pi/2 - \varepsilon$. If the test fails, it uses π to reduce the argument to the first octant, and then evaluates the sin and cos functions recursively (this call cannot result in infinite recursion).

If no argument reduction is needed, this function uses a generic version of the rectangular splitting algorithm if the precision is not too high, and otherwise invokes the asymptotically fast bit-burst algorithm.

void **arb_sin_cos_arf_bb**(*arb_t* s, *arb_t* c, const *arf_t* x, *slong* prec)

Computes the sine and cosine of x using the bit-burst algorithm. It is required that $|x| < \pi/2$ (this is not checked).

void **arb_sin_cos_wide**(*arb_t* s, *arb_t* c, const *arb_t* x, *slong* prec)

Computes an accurate enclosure (with both endpoints optimal to within about 2^{-30} as afforded by the radius format) of the range of sine and cosine on a given wide interval. The computation is done by evaluating the sine and cosine at the interval endpoints and determining whether peaks of -1 or 1 occur between the endpoints. The interval is then converted back to a ball.

The internal computations are done with doubles, using a simple floating-point algorithm to approximate the sine and cosine. It is easy to see that the cumulative errors in this algorithm add up to less than 2^{-30} , with the dominant source of error being a single approximate reduction by $\pi/2$. This reduction is done safely using doubles up to a magnitude of about 2^{20} . For larger arguments, a slower reduction using *arb_t* arithmetic is done as a preprocessing step.

void **arb_sin_cos_generic**(*arb_t* s, *arb_t* c, const *arb_t* x, *slong* prec)

Computes the sine and cosine of x by taking care of various special cases and computing the propagated error before calling **arb_sin_cos_arf_generic()**. This is used as a fallback inside **arb_sin_cos()** to take care of all cases without a fast path in that function.

void **arb_log_primes_vec_bsplitt**(*arb_ptr* res, *slong* n, *slong* prec)

Sets *res* to a vector containing the natural logarithms of the first n prime numbers, computed using binary splitting applied to simultaneous Machine-type formulas. This function is not optimized for large n or small *prec*.

ARB_LOG_PRIME_CACHE_NUM

Number of logarithms of small prime numbers to cache automatically.

ARB_LOG_REDUCTION_DEFAULT_MAX_PREC

Maximum precision to cache logarithms of small prime numbers automatically.

void **_arb_log_p_ensure_cached**(*slong* prec)

Ensure that the internal cache of logarithms of small prime numbers has entries to at least *prec* bits.

void **arb_exp_arf_log_reduction**(*arb_t* res, const *arf_t* x, *slong* prec, int minus_one)

Computes the exponential function using log reduction.

void **arb_exp_arf_generic**(*arb_t* z, const *arf_t* x, *slong* prec, int minus_one)

Computes the exponential function using an automatic choice between rectangular splitting and the bit-burst algorithm, without precomputation.

void **arb_exp_arf**(*arb_t* z, const *arf_t* x, *slong* prec, int minus_one, *slong* maglim)

Computes the exponential function using an automatic choice between all implemented algorithms.

void **arb_log_newton**(*arb_t* res, const *arb_t* x, *slong* prec)

void **arb_log_arf_newton**(*arb_t* res, const *arf_t* x, *slong* prec)

Computes the logarithm using Newton iteration.

ARB_ATAN_GAUSS_PRIME_CACHE_NUM

Number of primitive arctangents to cache automatically.

void **arb_atan_gauss_primes_vec_bsplitt**(*arb_ptr* res, *slong* n, *slong* prec)

Sets *res* to the primitive angles corresponding to the first n nonreal Gaussian primes (ignoring symmetries), computed using binary splitting applied to simultaneous Machine-type formulas. This function is not optimized for large n or small *prec*.

void `_arb_atan_gauss_p_ensure_cached`(*slong* prec)

void `arb_sin_cos_arf_atan_reduction`(*arb_t* res1, *arb_t* res2, const *arf_t* x, *slong* prec)

Computes sin and/or cos using reduction by primitive angles.

void `arb_atan_newton`(*arb_t* res, const *arb_t* x, *slong* prec)

void `arb_atan_arf_newton`(*arb_t* res, const *arf_t* x, *slong* prec)

Computes the arctangent using Newton iteration.

9.9.25 Vector functions

void `_arb_vec_zero`(*arb_ptr* vec, *slong* n)

Sets all entries in *vec* to zero.

int `_arb_vec_is_zero`(*arb_srcptr* vec, *slong* len)

Returns nonzero iff all entries in *x* are zero.

int `_arb_vec_is_finite`(*arb_srcptr* x, *slong* len)

Returns nonzero iff all entries in *x* certainly are finite.

int `_arb_vec_equal`(*arb_srcptr* vec1, *arb_srcptr* vec2, *slong* len)

Returns nonzero iff *vec1* and *vec2* are equal in the sense of `arb_equal()`, i.e. have both the same midpoint and radius elementwise.

int `_arb_vec_overlaps`(*arb_srcptr* vec1, *arb_srcptr* vec2, *slong* len)

Returns nonzero iff *vec1* overlaps *vec2* elementwise.

int `_arb_vec_contains`(*arb_srcptr* vec1, *arb_srcptr* vec2, *slong* len)

Returns nonzero iff *vec1* contains *vec2* elementwise.

void `_arb_vec_set`(*arb_ptr* res, *arb_srcptr* vec, *slong* len)

Sets *res* to a copy of *vec*.

void `_arb_vec_set_round`(*arb_ptr* res, *arb_srcptr* vec, *slong* len, *slong* prec)

Sets *res* to a copy of *vec*, rounding each entry to *prec* bits.

void `_arb_vec_swap`(*arb_ptr* vec1, *arb_ptr* vec2, *slong* len)

Swaps the entries of *vec1* and *vec2*.

void `_arb_vec_neg`(*arb_ptr* B, *arb_srcptr* A, *slong* n)

void `_arb_vec_sub`(*arb_ptr* C, *arb_srcptr* A, *arb_srcptr* B, *slong* n, *slong* prec)

void `_arb_vec_add`(*arb_ptr* C, *arb_srcptr* A, *arb_srcptr* B, *slong* n, *slong* prec)

void `_arb_vec_scalar_mul`(*arb_ptr* res, *arb_srcptr* vec, *slong* len, const *arb_t* c, *slong* prec)

void `_arb_vec_scalar_div`(*arb_ptr* res, *arb_srcptr* vec, *slong* len, const *arb_t* c, *slong* prec)

void `_arb_vec_scalar_mul_fmpz`(*arb_ptr* res, *arb_srcptr* vec, *slong* len, const *fmpz_t* c, *slong* prec)

void `_arb_vec_scalar_mul_2exp_si`(*arb_ptr* res, *arb_srcptr* src, *slong* len, *slong* c)

void `_arb_vec_scalar_addmul`(*arb_ptr* res, *arb_srcptr* vec, *slong* len, const *arb_t* c, *slong* prec)

Performs the respective scalar operation elementwise.

void `_arb_vec_get_mag`(*mag_t* bound, *arb_srcptr* vec, *slong* len)

Sets *bound* to an upper bound for the entries in *vec*.

slong `_arb_vec_bits`(*arb_srcptr* x, *slong* len)

Returns the maximum of `arb_bits()` for all entries in *vec*.

`void _arb_vec_set_powers(arb_ptr xs, const arb_t x, slong len, slong prec)`
Sets *xs* to the powers $1, x, x^2, \dots, x^{len-1}$.

`void _arb_vec_add_error_arf_vec(arb_ptr res, arf_srcptr err, slong len)`

`void _arb_vec_add_error_mag_vec(arb_ptr res, mag_srcptr err, slong len)`
Adds the magnitude of each entry in *err* to the radius of the corresponding entry in *res*.

`void _arb_vec_indeterminate(arb_ptr vec, slong len)`
Applies *arb_indeterminate()* elementwise.

`void _arb_vec_trim(arb_ptr res, arb_srcptr vec, slong len)`
Applies *arb_trim()* elementwise.

`int _arb_vec_get_unique_fmpz_vec(fmpz *res, arb_srcptr vec, slong len)`
Calls *arb_get_unique_fmpz()* elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in *vec* cannot be rounded uniquely to an integer, returns zero.

`void _arb_vec_printn(arb_srcptr vec, slong len, slong digits, ulong flags)`

`void _arb_vec_printd(arb_srcptr vec, slong len, slong ndigits)`
Prints *vec* in decimal using *arb_printn()* or *arb_printd()* on each entry.

9.10 acb.h – complex numbers

An *acb_t* represents a complex number with error bounds. An *acb_t* consists of a pair of real number balls of type *arb_struct*, representing the real and imaginary part with separate error bounds.

An *acb_t* thus represents a rectangle $[m_1 - r_1, m_1 + r_1] + [m_2 - r_2, m_2 + r_2]i$ in the complex plane. This is used instead of a disk or square representation (consisting of a complex floating-point midpoint with a single radius), since it allows implementing many operations more conveniently by splitting into ball operations on the real and imaginary parts. It also allows tracking when complex numbers have an exact (for example exactly zero) real part and an inexact imaginary part, or vice versa.

The interface for the *acb_t* type is slightly less developed than that for the *arb_t* type. In many cases, the user can easily perform missing operations by directly manipulating the real and imaginary parts.

9.10.1 Types, macros and constants

type **acb_struct**

type **acb_t**

An *acb_struct* consists of a pair of *arb_struct*s. An *acb_t* is defined as an array of length one of type *acb_struct*, permitting an *acb_t* to be passed by reference.

type **acb_ptr**

Alias for *acb_struct* *, used for vectors of numbers.

type **acb_srcptr**

Alias for *const acb_struct* *, used for vectors of numbers when passed as constant input to functions.

acb_realref(x)

Macro returning a pointer to the real part of *x* as an *arb_t*.

acb_imagref(x)

Macro returning a pointer to the imaginary part of *x* as an *arb_t*.

9.10.2 Memory management

void **acb_init**(*acb_t* x)

Initializes the variable *x* for use, and sets its value to zero.

void **acb_clear**(*acb_t* x)

Clears the variable *x*, freeing or recycling its allocated memory.

acb_ptr **_acb_vec_init**(*slong* n)

Returns a pointer to an array of *n* initialized *acb_struct*s.

void **_acb_vec_clear**(*acb_ptr* v, *slong* n)

Clears an array of *n* initialized *acb_struct*s.

slong **acb_allocated_bytes**(const *acb_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add **sizeof(acb_struct)** to get the size of the object as a whole.

slong **_acb_vec_allocated_bytes**(*acb_srcptr* vec, *slong* len)

Returns the total number of bytes allocated for this vector, i.e. the space taken up by the vector itself plus the sum of the internal heap allocation sizes for all its member elements.

double **_acb_vec_estimate_allocated_bytes**(*slong* len, *slong* prec)

Estimates the number of bytes that need to be allocated for a vector of *len* elements with *prec* bits of precision, including the space for internal limb data. See comments for **_arb_vec_estimate_allocated_bytes()**.

9.10.3 Basic manipulation

void **acb_zero**(*acb_t* z)

void **acb_one**(*acb_t* z)

void **acb_onei**(*acb_t* z)

Sets *z* respectively to 0, 1, $i = \sqrt{-1}$.

void **acb_set**(*acb_t* z, const *acb_t* x)

void **acb_set_ui**(*acb_t* z, *ulong* x)

void **acb_set_si**(*acb_t* z, *slong* x)

void **acb_set_d**(*acb_t* z, double x)

void **acb_set_fmpz**(*acb_t* z, const *fmpz_t* x)

void **acb_set_arb**(*acb_t* z, const *arb_t* c)

Sets *z* to the value of *x*.

void **acb_set_si_si**(*acb_t* z, *slong* x, *slong* y)

void **acb_set_d_d**(*acb_t* z, double x, double y)

void **acb_set_fmpz_fmpz**(*acb_t* z, const *fmpz_t* x, const *fmpz_t* y)

void **acb_set_arb_arb**(*acb_t* z, const *arb_t* x, const *arb_t* y)

Sets the real and imaginary part of *z* to the values *x* and *y* respectively

void **acb_set_fmpq**(*acb_t* z, const *fmpq_t* x, *slong* prec)

void **acb_set_round**(*acb_t* z, const *acb_t* x, *slong* prec)

`void acb_set_round_fmpz(acb_t z, const fmpz_t x, long prec)`
`void acb_set_round_arb(acb_t z, const arb_t x, long prec)`
Sets z to x , rounded to $prec$ bits.
`void acb_swap(acb_t z, acb_t x)`
Swaps z and x efficiently.
`void acb_add_error_arf(acb_t x, const arf_t err)`
`void acb_add_error_mag(acb_t x, const mag_t err)`
`void acb_add_error_arb(acb_t x, const arb_t err)`
Adds err to the error bounds of both the real and imaginary parts of x , modifying x in-place.
`void acb_get_mid(acb_t m, const acb_t x)`
Sets m to the midpoint of x .

9.10.4 Input and output

The `acb_print...` functions print to standard output, while `acb_fprint...` functions print to the stream *file*.

`void acb_print(const acb_t x)`
`void acb_fprint(FILE *file, const acb_t x)`
Prints the internal representation of x .
`void acb_printd(const acb_t x, long digits)`
`void acb_fprintd(FILE *file, const acb_t x, long digits)`
Prints x in decimal. The printed value of the radius is not adjusted to compensate for the fact that the binary-to-decimal conversion of both the midpoint and the radius introduces additional error.
`void acb_printn(const acb_t x, long digits, ulong flags)`
`void acb_fprintn(FILE *file, const acb_t x, long digits, ulong flags)`
Prints a nice decimal representation of x , using the format of `arb_get_str()` (or the corresponding `arb_printn()`) for the real and imaginary parts.
By default, the output shows the midpoint of both the real and imaginary parts with a guaranteed error of at most one unit in the last decimal place. In addition, explicit error bounds are printed so that the displayed decimal interval is guaranteed to enclose x .
Any flags understood by `arb_get_str()` can be passed via *flags* to control the format of the real and imaginary parts.

9.10.5 Random number generation

`void acb_randtest(acb_t z, flint_rand_t state, long prec, long mag_bits)`
Generates a random complex number by generating separate random real and imaginary parts.
`void acb_randtest_special(acb_t z, flint_rand_t state, long prec, long mag_bits)`
Generates a random complex number by generating separate random real and imaginary parts. Also generates NaNs and infinities.
`void acb_randtest_precise(acb_t z, flint_rand_t state, long prec, long mag_bits)`
Generates a random complex number with precise real and imaginary parts.

void **acb_randtest_param**(*acb_t* z, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Generates a random complex number, with very high probability of generating integers and half-integers.

void **acb_urandom**(*acb_t* z, *flint_rand_t* state, *slong* prec)

Generates a random complex number with precise real and imaginary parts, uniformly chosen in the unit disk.

9.10.6 Precision and comparisons

int **acb_is_zero**(const *acb_t* z)

Returns nonzero iff z is zero.

int **acb_is_one**(const *acb_t* z)

Returns nonzero iff z is exactly 1.

int **acb_is_finite**(const *acb_t* z)

Returns nonzero iff z certainly is finite.

int **acb_is_exact**(const *acb_t* z)

Returns nonzero iff z is exact.

int **acb_is_int**(const *acb_t* z)

Returns nonzero iff z is an exact integer.

int **acb_is_int_2exp_si**(const *acb_t* x, *slong* e)

Returns nonzero iff z exactly equals $n2^e$ for some integer n .

int **acb_equal**(const *acb_t* x, const *acb_t* y)

Returns nonzero iff x and y are identical as sets, i.e. if the real and imaginary parts are equal as balls.

Note that this is not the same thing as testing whether both x and y certainly represent the same complex number, unless either x or y is exact (and neither contains NaN). To test whether both operands *might* represent the same mathematical quantity, use **acb_overlaps()** or **acb_contains()**, depending on the circumstance.

int **acb_equal_si**(const *acb_t* x, *slong* y)

Returns nonzero iff x is equal to the integer y .

int **acb_eq**(const *acb_t* x, const *acb_t* y)

Returns nonzero iff x and y are certainly equal, as determined by testing that **arb_eq()** holds for both the real and imaginary parts.

int **acb_ne**(const *acb_t* x, const *acb_t* y)

Returns nonzero iff x and y are certainly not equal, as determined by testing that **arb_ne()** holds for either the real or imaginary parts.

int **acb_overlaps**(const *acb_t* x, const *acb_t* y)

Returns nonzero iff x and y have some point in common.

void **acb_union**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

Sets z to a complex interval containing both x and y .

void **acb_get_abs_ubound_arf**(*arf_t* u, const *acb_t* z, *slong* prec)

Sets u to an upper bound for the absolute value of z , computed using a working precision of $prec$ bits.

void **acb_get_abs_lbound_arf**(*arf_t* u, const *acb_t* z, *slong* prec)

Sets u to a lower bound for the absolute value of z , computed using a working precision of $prec$ bits.

void **acb_get_rad_ubound_arf**(*arf_t* u, const *acb_t* z, *slong* prec)

Sets *u* to an upper bound for the error radius of *z* (the value is currently not computed tightly).

void **acb_get_mag**(*mag_t* u, const *acb_t* x)

Sets *u* to an upper bound for the absolute value of *x*.

void **acb_get_mag_lower**(*mag_t* u, const *acb_t* x)

Sets *u* to a lower bound for the absolute value of *x*.

int **acb_contains_fmpq**(const *acb_t* x, const *fmpq_t* y)

int **acb_contains_fmpz**(const *acb_t* x, const *fmpz_t* y)

int **acb_contains**(const *acb_t* x, const *acb_t* y)

Returns nonzero iff *y* is contained in *x*.

int **acb_contains_zero**(const *acb_t* x)

Returns nonzero iff zero is contained in *x*.

int **acb_contains_int**(const *acb_t* x)

Returns nonzero iff the complex interval represented by *x* contains an integer.

int **acb_contains_interior**(const *acb_t* x, const *acb_t* y)

Tests if *y* is contained in the interior of *x*. This predicate always evaluates to false if *x* and *y* are both real-valued, since an imaginary part of 0 is not considered contained in the interior of the point interval 0. More generally, the same problem occurs for intervals with an exact real or imaginary part. Such intervals must be handled specially by the user where a different interpretation is intended.

slong **acb_rel_error_bits**(const *acb_t* x)

Returns the effective relative error of *x* measured in bits. This is computed as if calling **arb_rel_error_bits()** on the real ball whose midpoint is the larger out of the real and imaginary midpoints of *x*, and whose radius is the larger out of the real and imaginary radiuses of *x*.

slong **acb_rel_accuracy_bits**(const *acb_t* x)

Returns the effective relative accuracy of *x* measured in bits, equal to the negative of the return value from **acb_rel_error_bits()**.

slong **acb_rel_one_accuracy_bits**(const *acb_t* x)

Given a ball with midpoint *m* and radius *r*, returns an approximation of the relative accuracy of $[\max(1, |m|) \pm r]$ measured in bits.

slong **acb_bits**(const *acb_t* x)

Returns the maximum of **arb_bits** applied to the real and imaginary parts of *x*, i.e. the minimum precision sufficient to represent *x* exactly.

void **acb_indeterminate**(*acb_t* x)

Sets *x* to $[\text{NaN} \pm \infty] + [\text{NaN} \pm \infty]i$, representing an indeterminate result.

void **acb_trim**(*acb_t* y, const *acb_t* x)

Sets *y* to a copy of *x* with both the real and imaginary parts trimmed (see **arb_trim()**).

int **acb_is_real**(const *acb_t* x)

Returns nonzero iff the imaginary part of *x* is zero. It does not test whether the real part of *x* also is finite.

int **acb_get_unique_fmpz**(*fmpz_t* z, const *acb_t* x)

If *x* contains a unique integer, sets *z* to that value and returns nonzero. Otherwise (if *x* represents no integers or more than one integer), returns zero.

9.10.7 Complex parts

void **acb_get_real**(*arb_t* re, const *acb_t* z)

Sets *re* to the real part of *z*.

void **acb_get_imag**(*arb_t* im, const *acb_t* z)

Sets *im* to the imaginary part of *z*.

void **acb_arg**(*arb_t* r, const *acb_t* z, *slong* prec)

Sets *r* to a real interval containing the complex argument (phase) of *z*. We define the complex argument have a discontinuity on $(-\infty, 0]$, with the special value $\arg(0) = 0$, and $\arg(a + 0i) = \pi$ for $a < 0$. Equivalently, if $z = a + bi$, the argument is given by $\text{atan2}(b, a)$ (see [arb_atan2\(\)](#)).

void **acb_abs**(*arb_t* r, const *acb_t* z, *slong* prec)

Sets *r* to the absolute value of *z*.

void **acb_sgn**(*acb_t* r, const *acb_t* z, *slong* prec)

Sets *r* to the complex sign of *z*, defined as 0 if *z* is exactly zero and the projection onto the unit circle $z/|z| = \exp(i \arg(z))$ otherwise.

void **acb_csgn**(*arb_t* r, const *acb_t* z)

Sets *r* to the extension of the real sign function taking the value 1 for *z* strictly in the right half plane, -1 for *z* strictly in the left half plane, and the sign of the imaginary part when *z* is on the imaginary axis. Equivalently, $\text{csgn}(z) = z/\sqrt{z^2}$ except that the value is 0 when *z* is exactly zero.

9.10.8 Arithmetic

void **acb_neg**(*acb_t* z, const *acb_t* x)

Sets *z* to the negation of *x*.

void **acb_conj**(*acb_t* z, const *acb_t* x)

Sets *z* to the complex conjugate of *x*.

void **acb_add_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* prec)

void **acb_add_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* prec)

void **acb_add_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* prec)

void **acb_add_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

void **acb_add**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

Sets *z* to the sum of *x* and *y*.

void **acb_sub_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* prec)

void **acb_sub_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* prec)

void **acb_sub_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* prec)

void **acb_sub_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

void **acb_sub**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

Sets *z* to the difference of *x* and *y*.

void **acb_mul_onei**(*acb_t* z, const *acb_t* x)

Sets *z* to *x* multiplied by the imaginary unit.

void **acb_div_onei**(*acb_t* z, const *acb_t* x)

Sets z to x divided by the imaginary unit.

void **acb_mul_i_pow_si**(*acb_t* z, const *acb_t* x, *slong* k)

Sets z to x multiplied by i^k , where i denotes the imaginary unit.

void **acb_mul_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* prec)

void **acb_mul_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* prec)

void **acb_mul_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* prec)

void **acb_mul_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

Sets z to the product of x and y .

void **acb_mul**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

Sets z to the product of x and y . If at least one part of x or y is zero, the operations is reduced to two real multiplications. If x and y are the same pointers, they are assumed to represent the same mathematical quantity and the squaring formula is used.

void **acb_mul_2exp_si**(*acb_t* z, const *acb_t* x, *slong* e)

void **acb_mul_2exp_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* e)

Sets z to x multiplied by 2^e , without rounding.

void **acb_sqr**(*acb_t* z, const *acb_t* x, *slong* prec)

Sets z to x squared.

void **acb_cube**(*acb_t* z, const *acb_t* x, *slong* prec)

Sets z to x cubed, computed efficiently using two real squarings, two real multiplications, and scalar operations.

void **acb_addmul**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

void **acb_addmul_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* prec)

void **acb_addmul_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* prec)

void **acb_addmul_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* prec)

void **acb_addmul_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

Sets z to z plus the product of x and y .

void **acb_submul**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

void **acb_submul_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* prec)

void **acb_submul_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* prec)

void **acb_submul_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* prec)

void **acb_submul_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

Sets z to z minus the product of x and y .

void **acb_inv**(*acb_t* z, const *acb_t* x, *slong* prec)

Sets z to the multiplicative inverse of x .

void **acb_div_ui**(*acb_t* z, const *acb_t* x, *ulong* y, *slong* prec)

void **acb_div_si**(*acb_t* z, const *acb_t* x, *slong* y, *slong* prec)

void **acb_div_fmpz**(*acb_t* z, const *acb_t* x, const *fmpz_t* y, *slong* prec)

void **acb_div_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

```
void acb_div(acb_t z, const acb_t x, const acb_t y, slong prec)
```

Sets z to the quotient of x and y .

9.10.9 Dot product

```
void acb_dot_precise(acb_t res, const acb_t s, int subtract, acb_srcptr x, slong xstep, acb_srcptr y,
    slong ystep, slong len, slong prec)
```

```
void acb_dot_simple(acb_t res, const acb_t s, int subtract, acb_srcptr x, slong xstep, acb_srcptr y,
    slong ystep, slong len, slong prec)
```

```
void acb_dot(acb_t res, const acb_t s, int subtract, acb_srcptr x, slong xstep, acb_srcptr y, slong
    ystep, slong len, slong prec)
```

Computes the dot product of the vectors x and y , setting res to $s + (-1)^{\text{subtract}} \sum_{i=0}^{\text{len}-1} x_i y_i$.

The initial term s is optional and can be omitted by passing *NULL* (equivalently, $s = 0$). The parameter *subtract* must be 0 or 1. The length *len* is allowed to be negative, which is equivalent to a length of zero. The parameters *xstep* or *ystep* specify a step length for traversing subsequences of the vectors x and y ; either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between res and s but not between res and the entries of x and y .

The default version determines the optimal precision for each term and performs all internal calculations using mpn arithmetic with minimal overhead. This is the preferred way to compute a dot product; it is generally much faster and more precise than a simple loop.

The *simple* version performs fused multiply-add operations in a simple loop. This can be used for testing purposes and is also used as a fallback by the default version when the exponents are out of range for the optimized code.

The *precise* version computes the dot product exactly up to the final rounding. This can be extremely slow and is only intended for testing.

```
void acb_approx_dot(acb_t res, const acb_t s, int subtract, acb_srcptr x, slong xstep, acb_srcptr y,
    slong ystep, slong len, slong prec)
```

Computes an approximate dot product *without error bounds*. The radii of the inputs are ignored (only the midpoints are read) and only the midpoint of the output is written.

```
void acb_dot_ui(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const ulong *y,
    slong ystep, slong len, slong prec)
```

```
void acb_dot_si(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const slong *y,
    slong ystep, slong len, slong prec)
```

```
void acb_dot_uiui(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const ulong
    *y, slong ystep, slong len, slong prec)
```

```
void acb_dot_siui(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const ulong
    *y, slong ystep, slong len, slong prec)
```

```
void acb_dot_fmpz(acb_t res, const acb_t initial, int subtract, acb_srcptr x, slong xstep, const fmpz
    *y, slong ystep, slong len, slong prec)
```

Equivalent to *acb_dot()*, but with integers in the array y . The *uiui* and *siui* versions take an array of double-limb integers as input; the *siui* version assumes that these represent signed integers in two's complement form.

9.10.10 Mathematical constants

void **acb_const_pi**(*acb_t* y, *slong* prec)

Sets *y* to the constant π .

9.10.11 Powers and roots

void **acb_sqrt**(*acb_t* r, const *acb_t* z, *slong* prec)

Sets *r* to the square root of *z*. If either the real or imaginary part is exactly zero, only a single real square root is needed. Generally, we use the formula $\sqrt{a+bi} = u/2 + ib/u$, $u = \sqrt{2(|a+bi|+a)}$, requiring two real square root extractions.

void **acb_sqrt_analytic**(*acb_t* r, const *acb_t* z, int analytic, *slong* prec)

Computes the square root. If *analytic* is set, gives a NaN-containing result if *z* touches the branch cut.

void **acb_rsqrt**(*acb_t* r, const *acb_t* z, *slong* prec)

Sets *r* to the reciprocal square root of *z*. If either the real or imaginary part is exactly zero, only a single real reciprocal square root is needed. Generally, we use the formula $1/\sqrt{a+bi} = ((a+r) - bi)/v$, $r = |a+bi|$, $v = \sqrt{r|a+bi+r|^2}$, requiring one real square root and one real reciprocal square root.

void **acb_rsqrt_analytic**(*acb_t* r, const *acb_t* z, int analytic, *slong* prec)

Computes the reciprocal square root. If *analytic* is set, gives a NaN-containing result if *z* touches the branch cut.

void **acb_sqrts**(*acb_t* y1, *acb_t* y2, const *acb_t* x, *slong* prec)

Sets *y1* and *y2* to the two square roots of *x*, without any precision loss due to branch cuts. The order in which the square roots appear is not specified.

void **acb_quadratic_roots_fmpz**(*acb_t* r1, *acb_t* r2, const *fmpz_t* a, const *fmpz_t* b, const *fmpz_t* c, *slong* prec)

Sets *r1* and *r2* to the roots of the quadratic polynomial $ax^2 + bx + c$. Requires that *a* is nonzero. This function is implemented so that both roots are computed accurately even when direct use of the quadratic formula would lose accuracy.

void **acb_root_ui**(*acb_t* r, const *acb_t* z, *ulong* k, *slong* prec)

Sets *r* to the principal *k*-th root of *z*.

void **acb_pow_fmpz**(*acb_t* y, const *acb_t* b, const *fmpz_t* e, *slong* prec)

void **acb_pow_ui**(*acb_t* y, const *acb_t* b, *ulong* e, *slong* prec)

void **acb_pow_si**(*acb_t* y, const *acb_t* b, *slong* e, *slong* prec)

Sets $y = b^e$ using binary exponentiation (with an initial division if $e < 0$). Note that these functions can get slow if the exponent is extremely large (in such cases **acb_pow()** may be superior).

void **acb_pow_arb**(*acb_t* z, const *acb_t* x, const *arb_t* y, *slong* prec)

void **acb_pow**(*acb_t* z, const *acb_t* x, const *acb_t* y, *slong* prec)

Sets $z = x^y$, computed using binary exponentiation if *y* is a small exact integer, as $z = (x^{1/2})^{2y}$ if *y* is a small exact half-integer, and generally as $z = \exp(y \log x)$.

void **acb_pow_analytic**(*acb_t* r, const *acb_t* x, const *acb_t* y, int analytic, *slong* prec)

Computes the power x^y . If *analytic* is set, gives a NaN-containing result if *x* touches the branch cut (unless *y* is an integer).

void **acb_unit_root**(*acb_t* res, *ulong* order, *slong* prec)

Sets *res* to $\exp(\frac{2i\pi}{\text{order}})$ to precision *prec*.

9.10.12 Exponentials and logarithms

void **acb_exp**(*acb_t* y, const *acb_t* z, *slong* prec)

Sets *y* to the exponential function of *z*, computed as $\exp(a + bi) = \exp(a) (\cos(b) + \sin(b)i)$.

void **acb_exp_pi_i**(*acb_t* y, const *acb_t* z, *slong* prec)

Sets *y* to $\exp(\pi iz)$.

void **acb_exp_invexp**(*acb_t* s, *acb_t* t, const *acb_t* z, *slong* prec)

Sets *s* = $\exp(z)$ and *t* = $\exp(-z)$.

void **acb_expml**(*acb_t* res, const *acb_t* z, *slong* prec)

Sets *res* to $\exp(z) - 1$, using a more accurate method when $z \approx 0$.

void **acb_log**(*acb_t* y, const *acb_t* z, *slong* prec)

Sets *y* to the principal branch of the natural logarithm of *z*, computed as $\log(a + bi) = \frac{1}{2} \log(a^2 + b^2) + i \arg(a + bi)$.

void **acb_log_analytic**(*acb_t* r, const *acb_t* z, int analytic, *slong* prec)

Computes the natural logarithm. If *analytic* is set, gives a NaN-containing result if *z* touches the branch cut.

void **acb_log1p**(*acb_t* z, const *acb_t* x, *slong* prec)

Sets *z* = $\log(1 + x)$, computed accurately when $x \approx 0$.

9.10.13 Trigonometric functions

void **acb_sin**(*acb_t* s, const *acb_t* z, *slong* prec)

void **acb_cos**(*acb_t* c, const *acb_t* z, *slong* prec)

void **acb_sin_cos**(*acb_t* s, *acb_t* c, const *acb_t* z, *slong* prec)

Sets *s* = $\sin(z)$, *c* = $\cos(z)$, evaluated as $\sin(a + bi) = \sin(a) \cosh(b) + i \cos(a) \sinh(b)$, $\cos(a + bi) = \cos(a) \cosh(b) - i \sin(a) \sinh(b)$.

void **acb_tan**(*acb_t* s, const *acb_t* z, *slong* prec)

Sets *s* = $\tan(z) = \sin(z) / \cos(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void **acb_cot**(*acb_t* s, const *acb_t* z, *slong* prec)

Sets *s* = $\cot(z) = \cos(z) / \sin(z)$. For large imaginary parts, the function is evaluated in a numerically stable way as $\pm i$ plus a decreasing exponential factor.

void **acb_sin_pi**(*acb_t* s, const *acb_t* z, *slong* prec)

void **acb_cos_pi**(*acb_t* s, const *acb_t* z, *slong* prec)

void **acb_sin_cos_pi**(*acb_t* s, *acb_t* c, const *acb_t* z, *slong* prec)

Sets *s* = $\sin(\pi z)$, *c* = $\cos(\pi z)$, evaluating the trigonometric factors of the real and imaginary part accurately via **arb_sin_cos_pi()**.

void **acb_tan_pi**(*acb_t* s, const *acb_t* z, *slong* prec)

Sets *s* = $\tan(\pi z)$. Uses the same algorithm as **acb_tan()**, but evaluates the sine and cosine accurately via **arb_sin_cos_pi()**.

void **acb_cot_pi**(*acb_t* s, const *acb_t* z, *slong* prec)

Sets *s* = $\cot(\pi z)$. Uses the same algorithm as **acb_cot()**, but evaluates the sine and cosine accurately via **arb_sin_cos_pi()**.

void **acb_sec**(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes $\sec(z) = 1/\cos(z)$.

void **acb_csc**(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes $\csc(x) = 1/\sin(z)$.

void **acb_csc_pi**(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes $\csc(\pi x) = 1/\sin(\pi z)$. Evaluates the sine accurately via *acb_sin_pi()*.

void **acb_sinc**(*acb_t* s, const *acb_t* z, *slong* prec)
 Sets $s = \text{sinc}(x) = \sin(z)/z$.

void **acb_sinc_pi**(*acb_t* s, const *acb_t* z, *slong* prec)
 Sets $s = \text{sinc}(\pi x) = \sin(\pi z)/(\pi z)$.

9.10.14 Inverse trigonometric functions

void **acb_asin**(*acb_t* res, const *acb_t* z, *slong* prec)
 Sets *res* to $\text{asin}(z) = -i \log(iz + \sqrt{1 - z^2})$.

void **acb_acos**(*acb_t* res, const *acb_t* z, *slong* prec)
 Sets *res* to $\text{acos}(z) = \frac{1}{2}\pi - \text{asin}(z)$.

void **acb_atan**(*acb_t* res, const *acb_t* z, *slong* prec)
 Sets *res* to $\text{atan}(z) = \frac{1}{2}i(\log(1 - iz) - \log(1 + iz))$.

9.10.15 Hyperbolic functions

void **acb_sinh**(*acb_t* s, const *acb_t* z, *slong* prec)

void **acb_cosh**(*acb_t* c, const *acb_t* z, *slong* prec)

void **acb_sinh_cosh**(*acb_t* s, *acb_t* c, const *acb_t* z, *slong* prec)

void **acb_tanh**(*acb_t* s, const *acb_t* z, *slong* prec)

void **acb_coth**(*acb_t* s, const *acb_t* z, *slong* prec)
 Respectively computes $\sinh(z) = -i \sin(iz)$, $\cosh(z) = \cos(iz)$, $\tanh(z) = -i \tan(iz)$, $\coth(z) = i \cot(iz)$.

void **acb_sech**(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes $\text{sech}(z) = 1/\cosh(z)$.

void **acb_csch**(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes $\text{csch}(z) = 1/\sinh(z)$.

9.10.16 Inverse hyperbolic functions

void **acb_asinh**(*acb_t* res, const *acb_t* z, *slong* prec)
 Sets *res* to $\text{asinh}(z) = -i \text{asin}(iz)$.

void **acb_acosh**(*acb_t* res, const *acb_t* z, *slong* prec)
 Sets *res* to $\text{acosh}(z) = \log(z + \sqrt{z + 1}\sqrt{z - 1})$.

void **acb_atanh**(*acb_t* res, const *acb_t* z, *slong* prec)
 Sets *res* to $\text{atanh}(z) = -i \text{atan}(iz)$.

9.10.17 Lambert W function

void **acb_lambertw_asymp**(*acb_t* res, const *acb_t* z, const *fmpz_t* k, *slong* L, *slong* M, *slong* prec)

Sets *res* to the Lambert W function $W_k(z)$ computed using L and M terms in the bivariate series giving the asymptotic expansion at zero or infinity. This algorithm is valid everywhere, but the error bound is only finite when $|\log(z)|$ is sufficiently large.

int **acb_lambertw_check_branch**(const *acb_t* w, const *fmpz_t* k, *slong* prec)

Tests if w definitely lies in the image of the branch $W_k(z)$. This function is used internally to verify that a computed approximation of the Lambert W function lies on the intended branch. Note that this will necessarily evaluate to false for points exactly on (or overlapping) the branch cuts, where a different algorithm has to be used.

void **acb_lambertw_bound_deriv**(*mag_t* res, const *acb_t* z, const *acb_t* ez1, const *fmpz_t* k)

Sets *res* to an upper bound for $|W'_k(z)|$. The input *ez1* should contain the precomputed value of $ez + 1$.

Along the real line, the directional derivative of $W_k(z)$ is understood to be taken. As a result, the user must handle the branch cut discontinuity separately when using this function to bound perturbations in the value of $W_k(z)$.

void **acb_lambertw**(*acb_t* res, const *acb_t* z, const *fmpz_t* k, int flags, *slong* prec)

Sets *res* to the Lambert W function $W_k(z)$ where the index k selects the branch (with $k = 0$ giving the principal branch). The placement of branch cuts follows [CGHJK1996].

If *flags* is nonzero, nonstandard branch cuts are used.

If *flags* is set to `ACB_LAMBERTW_LEFT`, computes $W_{\text{left}|k}(z)$ which corresponds to $W_k(z)$ in the upper half plane and $W_{k+1}(z)$ in the lower half plane, connected continuously to the left of the branch points. In other words, the branch cut on $(-\infty, 0)$ is rotated counterclockwise to $(0, +\infty)$. (For $k = -1$ and $k = 0$, there is also a branch cut on $(-1/e, 0)$, continuous from below instead of from above to maintain counterclockwise continuity.)

If *flags* is set to `ACB_LAMBERTW_MIDDLE`, computes $W_{\text{middle}}(z)$ which corresponds to $W_{-1}(z)$ in the upper half plane and $W_1(z)$ in the lower half plane, connected continuously through $(-1/e, 0)$ with branch cuts on $(-\infty, -1/e)$ and $(0, +\infty)$. $W_{\text{middle}}(z)$ extends the real analytic function $W_{-1}(x)$ defined on $(-1/e, 0)$ to a complex analytic function, whereas the standard branch $W_{-1}(z)$ has a branch cut along the real segment.

The algorithm used to compute the Lambert W function is described in [Joh2017b].

9.10.18 Rising factorials

void **acb_rising_ui**(*acb_t* z, const *acb_t* x, *ulong* n, *slong* prec)

void **acb_rising**(*acb_t* z, const *acb_t* x, const *acb_t* n, *slong* prec)

Computes the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$. These functions are aliases for `acb_hypgeom_rising_ui()` and `acb_hypgeom_rising()`.

void **acb_rising2_ui**(*acb_t* u, *acb_t* v, const *acb_t* x, *ulong* n, *slong* prec)

Letting $u(x) = x(x+1)(x+2)\cdots(x+n-1)$, simultaneously compute $u(x)$ and $v(x) = u'(x)$. This function is a wrapper of `acb_hypgeom_rising_ui_jet()`.

void **acb_rising_ui_get_mag**(*mag_t* bound, const *acb_t* x, *ulong* n)

Computes an upper bound for the absolute value of the rising factorial $z = x(x+1)(x+2)\cdots(x+n-1)$. Not currently optimized for large n .

9.10.19 Gamma function

void **acb_gamma**(*acb_t* y, const *acb_t* x, *slong* prec)

Computes the gamma function $y = \Gamma(x)$. This is an alias for [`acb\_hypgeom\_gamma\(\)`](#).

void **acb_rgamma**(*acb_t* y, const *acb_t* x, *slong* prec)

Computes the reciprocal gamma function $y = 1/\Gamma(x)$, avoiding division by zero at the poles of the gamma function. This is an alias for [`acb\_hypgeom\_rgamma\(\)`](#).

void **acb_lgamma**(*acb_t* y, const *acb_t* x, *slong* prec)

Computes the logarithmic gamma function $y = \log \Gamma(x)$. This is an alias for [`acb\_hypgeom\_lgamma\(\)`](#).

The branch cut of the logarithmic gamma function is placed on the negative half-axis, which means that $\log \Gamma(z) + \log z = \log \Gamma(z+1)$ holds for all z , whereas $\log \Gamma(z) \neq \log(\Gamma(z))$ in general. In the left half plane, the reflection formula with correct branch structure is evaluated via [`acb\_log\_sin\_pi\(\)`](#).

void **acb_digamma**(*acb_t* y, const *acb_t* x, *slong* prec)

Computes the digamma function $y = \psi(x) = (\log \Gamma(x))' = \Gamma'(x)/\Gamma(x)$.

void **acb_log_sin_pi**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the logarithmic sine function defined by

$$S(z) = \log(\pi) - \log \Gamma(z) + \log \Gamma(1 - z)$$

which is equal to

$$S(z) = \int_{1/2}^z \pi \cot(\pi t) dt$$

where the path of integration goes through the upper half plane if $0 < \arg(z) \leq \pi$ and through the lower half plane if $-\pi < \arg(z) \leq 0$. Equivalently,

$$S(z) = \log(\sin(\pi(z - n))) \mp n\pi i, \quad n = \lfloor \operatorname{re}(z) \rfloor$$

where the negative sign is taken if $0 < \arg(z) \leq \pi$ and the positive sign is taken otherwise (if the interval $\arg(z)$ does not certainly satisfy either condition, the union of both cases is computed). After subtracting n , we have $0 \leq \operatorname{re}(z) < 1$. In this strip, we use $S(z) = \log(\sin(\pi(z)))$ if the imaginary part of z is small. Otherwise, we use $S(z) = i\pi(z - 1/2) + \log((1 + e^{-2i\pi z})/2)$ in the lower half-plane and the conjugated expression in the upper half-plane to avoid exponent overflow.

The function is evaluated at the midpoint and the propagated error is computed from $S'(z)$ to get a continuous change when z is non-real and n spans more than one possible integer value.

void **acb_polygamma**(*acb_t* res, const *acb_t* s, const *acb_t* z, *slong* prec)

Sets *res* to the value of the generalized polygamma function $\psi(s, z)$.

If s is a nonnegative order, this is simply the s -order derivative of the digamma function. If $s = 0$, this function simply calls the digamma function internally. For integers $s \geq 1$, it calls the Hurwitz zeta function. Note that for small integers $s \geq 1$, it can be faster to use [`acb\_poly\_digamma\_series\(\)`](#) and read off the coefficients.

The generalization to other values of s is due to Espinosa and Moll [EM2004]:

$$\psi(s, z) = \frac{\zeta'(s+1, z) + (\gamma + \psi(-s))\zeta(s+1, z)}{\Gamma(-s)}$$

void **acb_barnes_g**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_log_barnes_g**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes Barnes G -function or the logarithmic Barnes G -function, respectively. The logarithmic version has branch cuts on the negative real axis and is continuous elsewhere in the complex plane, in analogy with the logarithmic gamma function. The functional equation

$$\log G(z+1) = \log \Gamma(z) + \log G(z).$$

holds for all z .

For small integers, we directly use the recurrence relation $G(z+1) = \Gamma(z)G(z)$ together with the initial value $G(1) = 1$. For general z , we use the formula

$$\log G(z) = (z-1) \log \Gamma(z) - \zeta'(-1, z) + \zeta'(-1).$$

9.10.20 Zeta function

void **acb_zeta**(*acb_t* z, const *acb_t* s, *slong* prec)

Sets z to the value of the Riemann zeta function $\zeta(s)$. Note: for computing derivatives with respect to s , use **acb_poly_zeta_series()** or related methods.

This is a wrapper of **acb_dirichlet_zeta()**.

void **acb_hurwitz_zeta**(*acb_t* z, const *acb_t* s, const *acb_t* a, *slong* prec)

Sets z to the value of the Hurwitz zeta function $\zeta(s, a)$. Note: for computing derivatives with respect to s , use **acb_poly_zeta_series()** or related methods.

This is a wrapper of **acb_dirichlet_hurwitz()**.

void **acb_bernoulli_poly_ui**(*acb_t* res, *ulong* n, const *acb_t* x, *slong* prec)

Sets res to the value of the Bernoulli polynomial $B_n(x)$.

Warning: this function is only fast if either n or x is a small integer.

This function reads Bernoulli numbers from the global cache if they are already cached, but does not automatically extend the cache by itself.

9.10.21 Polylogarithms

void **acb_polylog**(*acb_t* w, const *acb_t* s, const *acb_t* z, *slong* prec)

void **acb_polylog_si**(*acb_t* w, *slong* s, const *acb_t* z, *slong* prec)

Sets w to the polylogarithm $\text{Li}_s(z)$.

9.10.22 Arithmetic-geometric mean

See *Algorithms for the arithmetic-geometric mean* for implementation details.

void **acb_agm1**(*acb_t* m, const *acb_t* z, *slong* prec)

Sets m to the arithmetic-geometric mean $M(z) = \text{agm}(1, z)$, defined such that the function is continuous in the complex plane except for a branch cut along the negative half axis (where it is continuous from above). This corresponds to always choosing an “optimal” branch for the square root in the arithmetic-geometric mean iteration.

void **acb_agm1_cpx**(*acb_ptr* m, const *acb_t* z, *slong* len, *slong* prec)

Sets the coefficients in the array m to the power series expansion of the arithmetic-geometric mean at the point z truncated to length len , i.e. $M(z+x) \in \mathbb{C}[[x]]$.

void **acb_agm**(*acb_t* m, const *acb_t* x, const *acb_t* y, *slong* prec)

Sets m to the arithmetic-geometric mean of x and y . The square roots in the AGM iteration are chosen so as to form the “optimal” AGM sequence. This gives a well-defined function of x and y except when x/y is a negative real number, in which case there are two optimal AGM sequences. In that case, an arbitrary but consistent choice is made (if a decision cannot be made due to inexact arithmetic, the union of both choices is returned).

9.10.23 Other special functions

void **acb_chebyshev_t_ui**(*acb_t* a, *ulong* n, const *acb_t* x, *slong* prec)

void **acb_chebyshev_u_ui**(*acb_t* a, *ulong* n, const *acb_t* x, *slong* prec)

Evaluates the Chebyshev polynomial of the first kind $a = T_n(x)$ or the Chebyshev polynomial of the second kind $a = U_n(x)$.

void **acb_chebyshev_t2_ui**(*acb_t* a, *acb_t* b, *ulong* n, const *acb_t* x, *slong* prec)

void **acb_chebyshev_u2_ui**(*acb_t* a, *acb_t* b, *ulong* n, const *acb_t* x, *slong* prec)

Simultaneously evaluates $a = T_n(x), b = T_{n-1}(x)$ or $a = U_n(x), b = U_{n-1}(x)$. Aliasing between a , b and x is not permitted.

9.10.24 Piecewise real functions

The following methods extend common piecewise real functions to piecewise complex analytic functions, useful together with the *acb_calc.h* module. If *analytic* is set, evaluation on a discontinuity or non-analytic point gives a NaN result.

void **acb_real_abs**(*acb_t* res, const *acb_t* z, int analytic, *slong* prec)

The absolute value is extended to $+z$ in the right half plane and $-z$ in the left half plane, with a discontinuity on the vertical line $\operatorname{Re}(z) = 0$.

void **acb_real_sgn**(*acb_t* res, const *acb_t* z, int analytic, *slong* prec)

The sign function is extended to $+1$ in the right half plane and -1 in the left half plane, with a discontinuity on the vertical line $\operatorname{Re}(z) = 0$. If *analytic* is not set, this is effectively the same function as *acb_csgn()*.

void **acb_real_heaviside**(*acb_t* res, const *acb_t* z, int analytic, *slong* prec)

The Heaviside step function (or unit step function) is extended to $+1$ in the right half plane and 0 in the left half plane, with a discontinuity on the vertical line $\operatorname{Re}(z) = 0$.

void **acb_real_floor**(*acb_t* res, const *acb_t* z, int analytic, *slong* prec)

The floor function is extended to a piecewise constant function equal to n in the strips with real part $(n, n + 1)$, with discontinuities on the vertical lines $\operatorname{Re}(z) = n$.

void **acb_real_ceil**(*acb_t* res, const *acb_t* z, int analytic, *slong* prec)

The ceiling function is extended to a piecewise constant function equal to $n + 1$ in the strips with real part $(n, n + 1)$, with discontinuities on the vertical lines $\operatorname{Re}(z) = n$.

void **acb_real_max**(*acb_t* res, const *acb_t* x, const *acb_t* y, int analytic, *slong* prec)

The real function $\max(x, y)$ is extended to a piecewise analytic function of two variables by returning x when $\operatorname{Re}(x) \geq \operatorname{Re}(y)$ and returning y when $\operatorname{Re}(x) < \operatorname{Re}(y)$, with discontinuities where $\operatorname{Re}(x) = \operatorname{Re}(y)$.

void **acb_real_min**(*acb_t* res, const *acb_t* x, const *acb_t* y, int analytic, *slong* prec)

The real function $\min(x, y)$ is extended to a piecewise analytic function of two variables by returning x when $\operatorname{Re}(x) \leq \operatorname{Re}(y)$ and returning y when $\operatorname{Re}(x) > \operatorname{Re}(y)$, with discontinuities where $\operatorname{Re}(x) = \operatorname{Re}(y)$.

void **acb_real_sqrtpos**(*acb_t* res, const *acb_t* z, int analytic, *slong* prec)

Extends the real square root function on $[0, +\infty)$ to the usual complex square root on the cut plane. Like *arb_sqrtpos()*, only the nonnegative part of z is considered if z is purely real and *analytic* is not set. This is useful for integrating $\sqrt{f(x)}$ where it is known that $f(x) \geq 0$: unlike *acb_sqrt_analytic()*, no spurious imaginary terms $[\pm\varepsilon]i$ are created when the balls computed for $f(x)$ straddle zero.

9.10.25 Vector functions

void **_acb_vec_zero**(*acb_ptr* A, *slong* n)

Sets all entries in *vec* to zero.

int **_acb_vec_is_zero**(*acb_srcptr* vec, *slong* len)

Returns nonzero iff all entries in x are zero.

int **_acb_vec_is_real**(*acb_srcptr* v, *slong* len)

Returns nonzero iff all entries in x have zero imaginary part.

int **_acb_vec_is_finite**(*acb_srcptr* vec, *slong* len)

Returns nonzero iff all entries in x certainly are finite.

int **_acb_vec_equal**(*acb_srcptr* vec1, *acb_srcptr* vec2, *slong* len)

Returns nonzero iff *vec1* and *vec2* are equal in the sense of *acb_equal()*, i.e. have both the same midpoint and radius elementwise.

int **_acb_vec_overlaps**(*acb_srcptr* vec1, *acb_srcptr* vec2, *slong* len)

Returns true iff *vec1* overlaps *vec2* elementwise.

int **_acb_vec_contains**(*acb_srcptr* vec1, *acb_srcptr* vec2, *slong* len)

Returns true iff *vec1* contains *vec2* elementwise.

void **_acb_vec_set**(*acb_ptr* res, *acb_srcptr* vec, *slong* len)

Sets *res* to a copy of *vec*.

void **_acb_vec_set_round**(*acb_ptr* res, *acb_srcptr* vec, *slong* len, *slong* prec)

Sets *res* to a copy of *vec*, rounding each entry to *prec* bits.

void **_acb_vec_swap**(*acb_ptr* vec1, *acb_ptr* vec2, *slong* len)

Swaps the entries of *vec1* and *vec2*.

void **_acb_vec_get_real**(*arb_ptr* re, *acb_srcptr* vec, *slong* len)

void **_acb_vec_get_imag**(*arb_ptr* im, *acb_srcptr* vec, *slong* len)

Sets each entry of *re* (resp. *im*) to the real (resp. imaginary) part of the corresponding entry of *vec*.

void **_acb_vec_set_real_imag**(*acb_ptr* vec, *arb_srcptr* re, *arb_srcptr* im, *slong* len)

Sets *vec* to the vector with real part *re* and imaginary part *im*.

void **_acb_vec_neg**(*acb_ptr* res, *acb_srcptr* vec, *slong* len)

void **_acb_vec_add**(*acb_ptr* res, *acb_srcptr* vec1, *acb_srcptr* vec2, *slong* len, *slong* prec)

void **_acb_vec_sub**(*acb_ptr* res, *acb_srcptr* vec1, *acb_srcptr* vec2, *slong* len, *slong* prec)

void **_acb_vec_scalar_submul**(*acb_ptr* res, *acb_srcptr* vec, *slong* len, const *acb_t* c, *slong* prec)

void **_acb_vec_scalar_addmul**(*acb_ptr* res, *acb_srcptr* vec, *slong* len, const *acb_t* c, *slong* prec)

void **_acb_vec_scalar_mul**(*acb_ptr* res, *acb_srcptr* vec, *slong* len, const *acb_t* c, *slong* prec)

`void _acb_vec_scalar_mul_ui(acb_ptr res, acb_srcptr vec, slong len, ulong c, slong prec)`

`void _acb_vec_scalar_mul_2exp_si(acb_ptr res, acb_srcptr vec, slong len, slong c)`

`void _acb_vec_scalar_mul_onei(acb_ptr res, acb_srcptr vec, slong len)`

`void _acb_vec_scalar_div_ui(acb_ptr res, acb_srcptr vec, slong len, ulong c, slong prec)`

`void _acb_vec_scalar_div(acb_ptr res, acb_srcptr vec, slong len, const acb_t c, slong prec)`

`void _acb_vec_scalar_mul_arb(acb_ptr res, acb_srcptr vec, slong len, const arb_t c, slong prec)`

`void _acb_vec_scalar_div_arb(acb_ptr res, acb_srcptr vec, slong len, const arb_t c, slong prec)`

`void _acb_vec_scalar_mul_fmpz(acb_ptr res, acb_srcptr vec, slong len, const fmpz_t c, slong prec)`

`void _acb_vec_scalar_div_fmpz(acb_ptr res, acb_srcptr vec, slong len, const fmpz_t c, slong prec)`

Performs the respective scalar operation elementwise.

`void _acb_vec_sqr(acb_ptr res, acb_srcptr vec, slong len, slong prec)`

Sets *res* to the square of *vec* elementwise.

slong `_acb_vec_bits(acb_srcptr vec, slong len)`

Returns the maximum of `arb_bits()` for all entries in *vec*.

`void _acb_vec_set_powers(acb_ptr xs, const acb_t x, slong len, slong prec)`

Sets *xs* to the powers $1, x, x^2, \dots, x^{\text{len}-1}$.

`void _acb_vec_unit_roots(acb_ptr z, slong order, slong len, slong prec)`

Sets *z* to the powers $1, z, z^2, \dots, z^{\text{len}-1}$ where $z = \exp(\frac{2i\pi}{\text{order}})$ to precision *prec*. *order* can be taken negative.

In order to avoid precision loss, this function does not simply compute powers of a primitive root.

`void _acb_vec_add_error_arf_vec(acb_ptr res, arf_srcptr err, slong len)`

`void _acb_vec_add_error_mag_vec(acb_ptr res, mag_srcptr err, slong len)`

Adds the magnitude of each entry in *err* to the radius of the corresponding entry in *res*.

`void _acb_vec_indeterminate(acb_ptr vec, slong len)`

Applies `acb_indeterminate()` elementwise.

`void _acb_vec_trim(acb_ptr res, acb_srcptr vec, slong len)`

Applies `acb_trim()` elementwise.

`int _acb_vec_get_unique_fmpz_vec(fmpz *res, acb_srcptr vec, slong len)`

Calls `acb_get_unique_fmpz()` elementwise and returns nonzero if all entries can be rounded uniquely to integers. If any entry in *vec* cannot be rounded uniquely to an integer, returns zero.

`void _acb_vec_sort_pretty(acb_ptr vec, slong len)`

Sorts the vector of complex numbers based on the real and imaginary parts. This is intended to reveal structure when printing a set of complex numbers, not to apply an order relation in a rigorous way.

`void _acb_vec_printd(acb_srcptr vec, slong len, slong digits)`

`void _acb_vec_printn(acb_srcptr vec, slong len, slong digits, ulong flags)`

Prints *vec* in decimal using `acb_printd()` or `acb_printn()` on each entry.

9.11 arb_poly.h – polynomials over the real numbers

An *arb_poly_t* represents a polynomial over the real numbers, implemented as an array of coefficients of type *arb_struct*.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

9.11.1 Types, macros and constants

type **arb_poly_struct**

type **arb_poly_t**

Contains a pointer to an array of coefficients (*coeffs*), the used length (*length*), and the allocated size of the array (*alloc*).

An *arb_poly_t* is defined as an array of length one of type *arb_poly_struct*, permitting an *arb_poly_t* to be passed by reference.

9.11.2 Memory management

void **arb_poly_init**(*arb_poly_t* poly)

Initializes the polynomial for use, setting it to the zero polynomial.

void **arb_poly_clear**(*arb_poly_t* poly)

Clears the polynomial, deallocating all coefficients and the coefficient array.

void **arb_poly_fit_length**(*arb_poly_t* poly, *slong* len)

Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void **_arb_poly_set_length**(*arb_poly_t* poly, *slong* len)

Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void **_arb_poly_normalise**(*arb_poly_t* poly)

Strips any trailing coefficients which are identical to zero.

slong **arb_poly_allocated_bytes**(const *arb_poly_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(arb_poly_struct)` to get the size of the object as a whole.

9.11.3 Basic manipulation

slong **arb_poly_length**(const *arb_poly_t* poly)

Returns the length of *poly*, i.e. zero if *poly* is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong **arb_poly_degree**(const *arb_poly_t* poly)

Returns the degree of *poly*, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by *poly*, so the return value of this function is effectively an upper bound.

int **arb_poly_is_zero**(const *arb_poly_t* poly)

int **arb_poly_is_one**(const *arb_poly_t* poly)

int **arb_poly_is_x**(const *arb_poly_t* poly)

Returns 1 if *poly* is exactly the polynomial 0, 1 or x respectively. Returns 0 otherwise.

void **arb_poly_zero**(*arb_poly_t* poly)

void **arb_poly_one**(*arb_poly_t* poly)

Sets *poly* to the constant 0 respectively 1.

void **arb_poly_set**(*arb_poly_t* dest, const *arb_poly_t* src)

Sets *dest* to a copy of *src*.

void **arb_poly_set_round**(*arb_poly_t* dest, const *arb_poly_t* src, *slong* prec)

Sets *dest* to a copy of *src*, rounded to *prec* bits.

void **arb_poly_set_trunc**(*arb_poly_t* dest, const *arb_poly_t* src, *slong* n)

void **arb_poly_set_trunc_round**(*arb_poly_t* dest, const *arb_poly_t* src, *slong* n, *slong* prec)

Sets *dest* to a copy of *src*, truncated to length *n* and rounded to *prec* bits.

void **arb_poly_set_coeff_si**(*arb_poly_t* poly, *slong* n, *slong* c)

void **arb_poly_set_coeff_arb**(*arb_poly_t* poly, *slong* n, const *arb_t* c)

Sets the coefficient with index *n* in *poly* to the value *c*. We require that *n* is nonnegative.

void **arb_poly_get_coeff_arb**(*arb_t* v, const *arb_poly_t* poly, *slong* n)

Sets *v* to the value of the coefficient with index *n* in *poly*. We require that *n* is nonnegative.

arb_poly_get_coeff_ptr(poly, n)

Given $n \geq 0$, returns a pointer to coefficient *n* of *poly*, or *NULL* if *n* exceeds the length of *poly*.

void **_arb_poly_shift_right**(*arb_ptr* res, *arb_srcptr* poly, *slong* len, *slong* n)

void **arb_poly_shift_right**(*arb_poly_t* res, const *arb_poly_t* poly, *slong* n)

Sets *res* to *poly* divided by x^n , throwing away the lower coefficients. We require that *n* is nonnegative.

void **_arb_poly_shift_left**(*arb_ptr* res, *arb_srcptr* poly, *slong* len, *slong* n)

void **arb_poly_shift_left**(*arb_poly_t* res, const *arb_poly_t* poly, *slong* n)

Sets *res* to *poly* multiplied by x^n . We require that *n* is nonnegative.

void **arb_poly_truncate**(*arb_poly_t* poly, *slong* n)

Truncates *poly* to have length at most *n*, i.e. degree strictly smaller than *n*. We require that *n* is nonnegative.

slong **arb_poly_valuation**(const *arb_poly_t* poly)

Returns the degree of the lowest term that is not exactly zero in *poly*. Returns -1 if *poly* is the zero polynomial.

9.11.4 Conversions

void **arb_poly_set_fmpz_poly**(*arb_poly_t* poly, const *fmpz_poly_t* src, *slong* prec)

void **arb_poly_set_fmpq_poly**(*arb_poly_t* poly, const *fmpq_poly_t* src, *slong* prec)

void **arb_poly_set_si**(*arb_poly_t* poly, *slong* src)

Sets *poly* to *src*, rounding the coefficients to *prec* bits.

9.11.5 Input and output

void **arb_poly_printd**(const *arb_poly_t* poly, *slong* digits)

Prints the polynomial as an array of coefficients, printing each coefficient using *arb_printd*.

void **arb_poly_fprintd**(FILE *file, const *arb_poly_t* poly, *slong* digits)

Prints the polynomial as an array of coefficients to the stream *file*, printing each coefficient using *arb_fprintd*.

9.11.6 Random generation

void **arb_poly_randtest**(*arb_poly_t* poly, *flint_rand_t* state, *slong* len, *slong* prec, *slong* mag_bits)

Creates a random polynomial with length at most *len*.

9.11.7 Comparisons

int **arb_poly_contains**(const *arb_poly_t* poly1, const *arb_poly_t* poly2)

int **arb_poly_contains_fmpz_poly**(const *arb_poly_t* poly1, const *fmpz_poly_t* poly2)

int **arb_poly_contains_fmpq_poly**(const *arb_poly_t* poly1, const *fmpq_poly_t* poly2)

Returns nonzero iff *poly1* contains *poly2*.

int **arb_poly_equal**(const *arb_poly_t* A, const *arb_poly_t* B)

Returns nonzero iff *A* and *B* are equal as polynomial balls, i.e. all coefficients have equal midpoint and radius.

int **_arb_poly_overlaps**(*arb_srcptr* poly1, *slong* len1, *arb_srcptr* poly2, *slong* len2)

int **arb_poly_overlaps**(const *arb_poly_t* poly1, const *arb_poly_t* poly2)

Returns nonzero iff *poly1* overlaps with *poly2*. The underscore function requires that *len1* is at least as large as *len2*.

int **arb_poly_get_unique_fmpz_poly**(*fmpz_poly_t* z, const *arb_poly_t* x)

If *x* contains a unique integer polynomial, sets *z* to that value and returns nonzero. Otherwise (if *x* represents no integers or more than one integer), returns zero, possibly partially modifying *z*.

9.11.8 Bounds

void **_arb_poly_majorant**(*arb_ptr* res, *arb_srcptr* poly, *slong* len, *slong* prec)

void **arb_poly_majorant**(*arb_poly_t* res, const *arb_poly_t* poly, *slong* prec)

Sets *res* to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in *poly*, rounded to *prec* bits.

9.11.9 Arithmetic

void **_arb_poly_add**(*arb_ptr* C, *arb_srcptr* A, *slong* lenA, *arb_srcptr* B, *slong* lenB, *slong* prec)

Sets $\{C, \max(\text{lenA}, \text{lenB})\}$ to the sum of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$. Allows aliasing of the input and output operands.

void **arb_poly_add**(*arb_poly_t* C, const *arb_poly_t* A, const *arb_poly_t* B, *slong* prec)

void **arb_poly_add_si**(*arb_poly_t* C, const *arb_poly_t* A, *slong* B, *slong* prec)

Sets *C* to the sum of *A* and *B*.

void **_arb_poly_sub**(*arb_ptr* C, *arb_srcptr* A, *slong* lenA, *arb_srcptr* B, *slong* lenB, *slong* prec)
 Sets $\{C, \max(\text{lenA}, \text{lenB})\}$ to the difference of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$. Allows aliasing of the input and output operands.

void **arb_poly_sub**(*arb_poly_t* C, const *arb_poly_t* A, const *arb_poly_t* B, *slong* prec)
 Sets C to the difference of A and B .

void **arb_poly_add_series**(*arb_poly_t* C, const *arb_poly_t* A, const *arb_poly_t* B, *slong* len, *slong* prec)
 Sets C to the sum of A and B , truncated to length len .

void **arb_poly_sub_series**(*arb_poly_t* C, const *arb_poly_t* A, const *arb_poly_t* B, *slong* len, *slong* prec)
 Sets C to the difference of A and B , truncated to length len .

void **arb_poly_neg**(*arb_poly_t* C, const *arb_poly_t* A)
 Sets C to the negation of A .

void **arb_poly_scalar_mul_2exp_si**(*arb_poly_t* C, const *arb_poly_t* A, *slong* c)
 Sets C to A multiplied by 2^c .

void **arb_poly_scalar_mul**(*arb_poly_t* C, const *arb_poly_t* A, const *arb_t* c, *slong* prec)
 Sets C to A multiplied by c .

void **arb_poly_scalar_div**(*arb_poly_t* C, const *arb_poly_t* A, const *arb_t* c, *slong* prec)
 Sets C to A divided by c .

void **_arb_poly_mullow_classical**(*arb_ptr* C, *arb_srcptr* A, *slong* lenA, *arb_srcptr* B, *slong* lenB, *slong* n, *slong* prec)

void **_arb_poly_mullow_block**(*arb_ptr* C, *arb_srcptr* A, *slong* lenA, *arb_srcptr* B, *slong* lenB, *slong* n, *slong* prec)

void **_arb_poly_mullow**(*arb_ptr* C, *arb_srcptr* A, *slong* lenA, *arb_srcptr* B, *slong* lenB, *slong* n, *slong* prec)

Sets $\{C, n\}$ to the product of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$, truncated to length n . The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$, $n > 0$, $\text{lenA} + \text{lenB} - 1 \geq n$.

The *classical* version uses a plain loop. This has good numerical stability but gets slow for large n .

The *block* version decomposes the product into several subproducts which are computed exactly over the integers.

It first attempts to find an integer c such that $A(2^c x)$ and $B(2^c x)$ have slowly varying coefficients, to reduce the number of blocks.

The scaling factor c is chosen in a quick, heuristic way by picking the first and last nonzero terms in each polynomial. If the indices in A are a_2, a_1 and the log-2 magnitudes are e_2, e_1 , and the indices in B are b_2, b_1 with corresponding magnitudes f_2, f_1 , then we compute c as the weighted arithmetic mean of the slopes, rounded to the nearest integer:

$$c = \left\lfloor \frac{(e_2 - e_1) + (f_2 - f_1)}{(a_2 - a_1) + (b_2 - b_1)} + \frac{1}{2} \right\rfloor.$$

This strategy is used because it is simple. It is not optimal in all cases, but will typically give good performance when multiplying two power series with a similar decay rate.

The default algorithm chooses the *classical* algorithm for short polynomials and the *block* algorithm for long polynomials.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

```
void arb_poly_mullassical(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n,
                        slong prec)
```

```
void arb_poly_mullass_block(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, slong
                           prec)
```

```
void arb_poly_mullass(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong n, slong prec)
```

Sets C to the product of A and B , truncated to length n . If the same variable is passed for A and B , sets C to the square of A truncated to length n .

```
void _arb_poly_mul(arb_ptr C, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
```

Sets $\{C, \text{lenA} + \text{lenB} - 1\}$ to the product of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$. This function is implemented as a simple wrapper for `_arb_poly_mullass()`.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

```
void arb_poly_mul(arb_poly_t C, const arb_poly_t A, const arb_poly_t B, slong prec)
```

Sets C to the product of A and B . If the same variable is passed for A and B , sets C to the square of A .

```
void _arb_poly_mulmid_block(arb_ptr z, arb_srcptr x, slong xlen, arb_srcptr y, slong ylen, slong
                           nlo, slong nhi, slong prec)
```

```
void arb_poly_mulmid_block(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, slong
                           nlo, slong nhi, slong prec)
```

```
void _arb_poly_mulmid_classical(arb_ptr z, arb_srcptr x, slong xlen, arb_srcptr y, slong ylen,
                               slong nlo, slong nhi, slong prec)
```

```
void arb_poly_mulmid_classical(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2,
                               slong nlo, slong nhi, slong prec)
```

```
void _arb_poly_mulmid(arb_ptr z, arb_srcptr x, slong xlen, arb_srcptr y, slong ylen, slong nlo, slong
                      nhi, slong prec)
```

```
void arb_poly_mulmid(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, slong nlo,
                     slong nhi, slong prec)
```

Analogous to *mullass* functions, but compute the product truncated at length *nhi* and right-shifted by *nlo*.

```
void _arb_poly_inv_series(arb_ptr Q, arb_srcptr A, slong Alen, slong len, slong prec)
```

Sets $\{Q, \text{len}\}$ to the power series inverse of $\{A, \text{Alen}\}$. Uses Newton iteration.

```
void arb_poly_inv_series(arb_poly_t Q, const arb_poly_t A, slong n, slong prec)
```

Sets Q to the power series inverse of A , truncated to length n .

```
void _arb_poly_div_series(arb_ptr Q, arb_srcptr A, slong Alen, arb_srcptr B, slong Blen, slong n,
                         slong prec)
```

Sets $\{Q, n\}$ to the power series quotient of $\{A, \text{Alen}\}$ by $\{B, \text{Blen}\}$. Uses Newton iteration followed by multiplication.

```
void arb_poly_div_series(arb_poly_t Q, const arb_poly_t A, const arb_poly_t B, slong n, slong
                        prec)
```

Sets Q to the power series quotient A divided by B , truncated to length n .

```
void _arb_poly_div(arb_ptr Q, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
```

```
void _arb_poly_rem(arb_ptr R, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB, slong prec)
```

```
void _arb_poly_divrem(arb_ptr Q, arb_ptr R, arb_srcptr A, slong lenA, arb_srcptr B, slong lenB,
                     slong prec)
```

```
int arb_poly_divrem(arb_poly_t Q, arb_poly_t R, const arb_poly_t A, const arb_poly_t B, slong prec)
```

Performs polynomial division with remainder, computing a quotient Q and a remainder R such that $A = BQ + R$. The implementation reverses the inputs and performs power series division.

If the leading coefficient of B contains zero (or if B is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

```
void _arb_poly_div_root(arb_ptr Q, arb_t R, arb_srcptr A, slong len, const arb_t c, slong prec)
```

Divides A by the polynomial $x - c$, computing the quotient Q as well as the remainder $R = f(c)$.

9.11.10 Composition

```
void _arb_poly_taylor_shift(arb_ptr g, const arb_t c, slong n, slong prec)
```

```
void arb_poly_taylor_shift(arb_poly_t g, const arb_poly_t f, const arb_t c, slong prec)
```

Sets g to the Taylor shift $f(x + c)$. The underscore methods act in-place on $g = f$ which has length n .

```
void _arb_poly_compose(arb_ptr res, arb_srcptr poly1, slong len1, arb_srcptr poly2, slong len2, slong prec)
```

```
void arb_poly_compose(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, slong prec)
```

Sets res to the composition $h(x) = f(g(x))$ where f is given by $poly1$ and g is given by $poly2$. The underscore method does not support aliasing of the output with either input polynomial.

```
void _arb_poly_compose_series(arb_ptr res, arb_srcptr poly1, slong len1, arb_srcptr poly2, slong len2, slong n, slong prec)
```

```
void arb_poly_compose_series(arb_poly_t res, const arb_poly_t poly1, const arb_poly_t poly2, slong n, slong prec)
```

Sets res to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where f is given by $poly1$ and g is given by $poly2$. Wraps `_gr_poly_compose_series()` which chooses automatically between various algorithms.

We require that the constant term in $g(x)$ is exactly zero. The underscore method does not support aliasing of the output with either input polynomial.

```
void _arb_poly_revert_series(arb_ptr h, arb_srcptr f, slong flen, slong n, slong prec)
```

```
void arb_poly_revert_series(arb_poly_t h, const arb_poly_t f, slong n, slong prec)
```

Sets h to the power series reversion of f , i.e. the expansion of the compositional inverse function $f^{-1}(x)$, truncated to order $O(x^n)$. Wraps `_gr_poly_revert_series()` which chooses automatically between various algorithms.

We require that the constant term in f is exactly zero and that the linear term is nonzero. The underscore methods assume that $flen$ is at least 2, and do not support aliasing.

9.11.11 Evaluation

```
void _arb_poly_evaluate_horner(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
```

```
void arb_poly_evaluate_horner(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
```

```
void _arb_poly_evaluate_rectangular(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
```

```
void arb_poly_evaluate_rectangular(arb_t y, const arb_poly_t f, const arb_t x, slong prec)
```

```
void _arb_poly_evaluate(arb_t y, arb_srcptr f, slong len, const arb_t x, slong prec)
```

void **arb_poly_evaluate**(*arb_t* y, const *arb_poly_t* f, const *arb_t* x, *slong* prec)

Sets $y = f(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_arb_poly_evaluate_acb_horner**(*acb_t* y, *arb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **arb_poly_evaluate_acb_horner**(*acb_t* y, const *arb_poly_t* f, const *acb_t* x, *slong* prec)

void **_arb_poly_evaluate_acb_rectangular**(*acb_t* y, *arb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **arb_poly_evaluate_acb_rectangular**(*acb_t* y, const *arb_poly_t* f, const *acb_t* x, *slong* prec)

void **_arb_poly_evaluate_acb**(*acb_t* y, *arb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **arb_poly_evaluate_acb**(*acb_t* y, const *arb_poly_t* f, const *acb_t* x, *slong* prec)

Sets $y = f(x)$ where x is a complex number, evaluating the polynomial respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_arb_poly_evaluate2_horner**(*arb_t* y, *arb_t* z, *arb_srcptr* f, *slong* len, const *arb_t* x, *slong* prec)

void **arb_poly_evaluate2_horner**(*arb_t* y, *arb_t* z, const *arb_poly_t* f, const *arb_t* x, *slong* prec)

void **_arb_poly_evaluate2_rectangular**(*arb_t* y, *arb_t* z, *arb_srcptr* f, *slong* len, const *arb_t* x, *slong* prec)

void **arb_poly_evaluate2_rectangular**(*arb_t* y, *arb_t* z, const *arb_poly_t* f, const *arb_t* x, *slong* prec)

void **_arb_poly_evaluate2**(*arb_t* y, *arb_t* z, *arb_srcptr* f, *slong* len, const *arb_t* x, *slong* prec)

void **arb_poly_evaluate2**(*arb_t* y, *arb_t* z, const *arb_poly_t* f, const *arb_t* x, *slong* prec)

Sets $y = f(x)$, $z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

void **_arb_poly_evaluate2_acb_horner**(*acb_t* y, *acb_t* z, *arb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **arb_poly_evaluate2_acb_horner**(*acb_t* y, *acb_t* z, const *arb_poly_t* f, const *acb_t* x, *slong* prec)

void **_arb_poly_evaluate2_acb_rectangular**(*acb_t* y, *acb_t* z, *arb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **arb_poly_evaluate2_acb_rectangular**(*acb_t* y, *acb_t* z, const *arb_poly_t* f, const *acb_t* x, *slong* prec)

void **_arb_poly_evaluate2_acb**(*acb_t* y, *acb_t* z, *arb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **arb_poly_evaluate2_acb**(*acb_t* y, *acb_t* z, const *arb_poly_t* f, const *acb_t* x, *slong* prec)

Sets $y = f(x)$, $z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

9.11.12 Product trees

void `_arb_poly_product_roots`(*arb_ptr* poly, *arb_srcptr* xs, *slong* n, *slong* prec)

void `arb_poly_product_roots`(*arb_poly_t* poly, *arb_srcptr* xs, *slong* n, *slong* prec)

Generates the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

void `_arb_poly_product_roots_complex`(*arb_ptr* poly, *arb_srcptr* r, *slong* rn, *acb_srcptr* c, *slong* cn, *slong* prec)

void `arb_poly_product_roots_complex`(*arb_poly_t* poly, *arb_srcptr* r, *slong* rn, *acb_srcptr* c, *slong* cn, *slong* prec)

Generates the polynomial

$$\left(\prod_{i=0}^{rn-1} (x - r_i) \right) \left(\prod_{i=0}^{cn-1} (x - c_i)(x - \bar{c}_i) \right)$$

having rn real roots given by the array r and having $2cn$ complex roots in conjugate pairs given by the length- cn array c . Either rn or cn or both may be zero.

Note that only one representative from each complex conjugate pair is supplied (unless a pair is supposed to be repeated with higher multiplicity). To construct a polynomial from complex roots where the conjugate pairs have not been distinguished, use `acb_poly_product_roots()` instead.

arb_ptr *`_arb_poly_tree_alloc`(*slong* len)

Returns an initialized data structured capable of representing a remainder tree (product tree) of len roots.

void `_arb_poly_tree_free`(*arb_ptr* *tree, *slong* len)

Deallocates a tree structure as allocated using `_arb_poly_tree_alloc`.

void `_arb_poly_tree_build`(*arb_ptr* *tree, *arb_srcptr* roots, *slong* len, *slong* prec)

Constructs a product tree from a given array of len roots. The tree structure must be pre-allocated to the specified length using `_arb_poly_tree_alloc()`.

9.11.13 Multipoint evaluation

void `_arb_poly_evaluate_vec_iter`(*arb_ptr* ys, *arb_srcptr* poly, *slong* plen, *arb_srcptr* xs, *slong* n, *slong* prec)

void `arb_poly_evaluate_vec_iter`(*arb_ptr* ys, const *arb_poly_t* poly, *arb_srcptr* xs, *slong* n, *slong* prec)

Evaluates the polynomial simultaneously at n given points, calling `_arb_poly_evaluate()` repeatedly.

void `_arb_poly_evaluate_vec_fast_precomp`(*arb_ptr* vs, *arb_srcptr* poly, *slong* plen, *arb_ptr* *tree, *slong* len, *slong* prec)

void `_arb_poly_evaluate_vec_fast`(*arb_ptr* ys, *arb_srcptr* poly, *slong* plen, *arb_srcptr* xs, *slong* n, *slong* prec)

void `arb_poly_evaluate_vec_fast`(*arb_ptr* ys, const *arb_poly_t* poly, *arb_srcptr* xs, *slong* n, *slong* prec)

Evaluates the polynomial simultaneously at n given points, using fast multipoint evaluation.

9.11.14 Interpolation

void `_arb_poly_interpolate_newton`(*arb_ptr* poly, *arb_srcptr* xs, *arb_srcptr* ys, *slong* n, *slong* prec)

void `arb_poly_interpolate_newton`(*arb_poly_t* poly, *arb_srcptr* xs, *arb_srcptr* ys, *slong* n, *slong* prec)

Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

void `_arb_poly_interpolate_barycentric`(*arb_ptr* poly, *arb_srcptr* xs, *arb_srcptr* ys, *slong* n, *slong* prec)

void `arb_poly_interpolate_barycentric`(*arb_poly_t* poly, *arb_srcptr* xs, *arb_srcptr* ys, *slong* n, *slong* prec)

Recovers the unique polynomial of length at most n that interpolates the given x and y values. This implementation uses the barycentric form of Lagrange interpolation.

void `_arb_poly_interpolation_weights`(*arb_ptr* w, *arb_ptr* *tree, *slong* len, *slong* prec)

void `_arb_poly_interpolate_fast_precomp`(*arb_ptr* poly, *arb_srcptr* ys, *arb_ptr* *tree, *arb_srcptr* weights, *slong* len, *slong* prec)

void `_arb_poly_interpolate_fast`(*arb_ptr* poly, *arb_srcptr* xs, *arb_srcptr* ys, *slong* len, *slong* prec)

void `arb_poly_interpolate_fast`(*arb_poly_t* poly, *arb_srcptr* xs, *arb_srcptr* ys, *slong* n, *slong* prec)

Recovers the unique polynomial of length at most n that interpolates the given x and y values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the x values and a vector of interpolation weights as additional inputs.

9.11.15 Differentiation

void `_arb_poly_derivative`(*arb_ptr* res, *arb_srcptr* poly, *slong* len, *slong* prec)

Sets $\{res, len - 1\}$ to the derivative of $\{poly, len\}$. Allows aliasing of the input and output.

void `arb_poly_derivative`(*arb_poly_t* res, const *arb_poly_t* poly, *slong* prec)

Sets res to the derivative of $poly$.

void `_arb_poly_nth_derivative`(*arb_ptr* res, *arb_srcptr* poly, *ulong* n, *slong* len, *slong* prec)

Sets $\{res, len - n\}$ to the n th derivative of $\{poly, len\}$. Does nothing if $len \leq n$. Allows aliasing of the input and output.

void `arb_poly_nth_derivative`(*arb_poly_t* res, const *arb_poly_t* poly, *ulong* n, *slong* prec)

Sets res to the n th derivative of $poly$.

void `_arb_poly_integral`(*arb_ptr* res, *arb_srcptr* poly, *slong* len, *slong* prec)

Sets $\{res, len\}$ to the integral of $\{poly, len - 1\}$. Allows aliasing of the input and output.

void `arb_poly_integral`(*arb_poly_t* res, const *arb_poly_t* poly, *slong* prec)

Sets res to the integral of $poly$.

9.11.16 Transforms

`void _arb_poly_borel_transform(arb_ptr res, arb_srcptr poly, slong len, slong prec)`

`void arb_poly_borel_transform(arb_poly_t res, const arb_poly_t poly, slong prec)`

Computes the Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k (a_k/k!)x^k$. The underscore method allows aliasing.

`void _arb_poly_inv_borel_transform(arb_ptr res, arb_srcptr poly, slong len, slong prec)`

`void arb_poly_inv_borel_transform(arb_poly_t res, const arb_poly_t poly, slong prec)`

Computes the inverse Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k a_k k! x^k$. The underscore method allows aliasing.

`void _arb_poly_binomial_transform_basecase(arb_ptr b, arb_srcptr a, slong alen, slong len, slong prec)`

`void arb_poly_binomial_transform_basecase(arb_poly_t b, const arb_poly_t a, slong len, slong prec)`

`void _arb_poly_binomial_transform_convolution(arb_ptr b, arb_srcptr a, slong alen, slong len, slong prec)`

`void arb_poly_binomial_transform_convolution(arb_poly_t b, const arb_poly_t a, slong len, slong prec)`

`void _arb_poly_binomial_transform(arb_ptr b, arb_srcptr a, slong alen, slong len, slong prec)`

`void arb_poly_binomial_transform(arb_poly_t b, const arb_poly_t a, slong len, slong prec)`

Computes the binomial transform of the input polynomial, truncating the output to length *len*. The binomial transform maps the coefficients a_k in the input polynomial to the coefficients b_k in the output polynomial via $b_n = \sum_{k=0}^n (-1)^k \binom{n}{k} a_k$. The binomial transform is equivalent to the power series composition $f(x) \rightarrow (1-x)^{-1} f(x/(x-1))$, and is its own inverse.

The *basecase* version evaluates coefficients one by one from the definition, generating the binomial coefficients by a recurrence relation.

The *convolution* version uses the identity $T(f(x)) = B^{-1}(e^x B(f(-x)))$ where T denotes the binomial transform operator and B denotes the Borel transform operator. This only costs a single polynomial multiplication, plus some scalar operations.

The default version automatically chooses an algorithm.

The underscore methods do not support aliasing, and assume that the lengths are nonzero.

`void _arb_poly_graeffe_transform(arb_ptr b, arb_srcptr a, slong len, slong prec)`

`void arb_poly_graeffe_transform(arb_poly_t b, const arb_poly_t a, slong prec)`

Computes the Graeffe transform of input polynomial.

The Graeffe transform G of a polynomial P is defined through the equation $G(x^2) = \pm P(x)P(-x)$. The sign is given by $(-1)^d$, where $d = \deg(P)$. The Graeffe transform has the property that its roots are exactly the squares of the roots of P .

The underscore method assumes that a and b are initialized, a is of length *len*, and b is of length at least *len*. Both methods allow aliasing.

9.11.17 Powers and elementary functions

void `_arb_poly_pow_ui_trunc_binexp`(*arb_ptr* res, *arb_srcptr* f, *slong* flen, *ulong* exp, *slong* len, *slong* prec)

Sets $\{res, len\}$ to $\{f, flen\}$ raised to the power exp , truncated to length len . Requires that len is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that $flen$ and len are positive. Uses binary exponentiation.

void `arb_poly_pow_ui_trunc_binexp`(*arb_poly_t* res, const *arb_poly_t* poly, *ulong* exp, *slong* len, *slong* prec)

Sets res to $poly$ raised to the power exp , truncated to length len . Uses binary exponentiation.

void `_arb_poly_pow_ui`(*arb_ptr* res, *arb_srcptr* f, *slong* flen, *ulong* exp, *slong* prec)

Sets res to $\{f, flen\}$ raised to the power exp . Does not support aliasing of the input and output, and requires that $flen$ is positive.

void `arb_poly_pow_ui`(*arb_poly_t* res, const *arb_poly_t* poly, *ulong* exp, *slong* prec)

Sets res to $poly$ raised to the power exp .

void `_arb_poly_pow_series`(*arb_ptr* h, *arb_srcptr* f, *slong* flen, *arb_srcptr* g, *slong* glen, *slong* len, *slong* prec)

Sets $\{h, len\}$ to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length len . This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that $flen$ and $glen$ do not exceed len .

void `arb_poly_pow_series`(*arb_poly_t* h, const *arb_poly_t* f, const *arb_poly_t* g, *slong* len, *slong* prec)

Sets h to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length len . This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

void `_arb_poly_pow_arb_series`(*arb_ptr* h, *arb_srcptr* f, *slong* flen, const *arb_t* g, *slong* len, *slong* prec)

Sets $\{h, len\}$ to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length len . This function detects special cases such as g being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that $flen$ does not exceed len .

void `arb_poly_pow_arb_series`(*arb_poly_t* h, const *arb_poly_t* f, const *arb_t* g, *slong* len, *slong* prec)

Sets h to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length len .

void `_arb_poly_sqrt_series`(*arb_ptr* g, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void `arb_poly_sqrt_series`(*arb_poly_t* g, const *arb_poly_t* h, *slong* n, *slong* prec)

Sets g to the power series square root of h , truncated to length n . Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that $hlen$ and n are greater than zero.

void `_arb_poly_rsqrts_series`(*arb_ptr* g, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void `arb_poly_rsqrts_series`(*arb_poly_t* g, const *arb_poly_t* h, *slong* n, *slong* prec)

Sets g to the reciprocal power series square root of h , truncated to length n . Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and *n* are greater than zero.

```
void _arb_poly_log_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

```
void arb_poly_log_series(arb_poly_t res, const arb_poly_t f, slong n, slong prec)
```

Sets *res* to the power series logarithm of *f*, truncated to length *n*. Uses the formula $\log(f(x)) = \int f'(x)/f(x)dx$, adding the logarithm of the constant term in *f* as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and *n* are greater than zero.

```
void _arb_poly_log1p_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

```
void arb_poly_log1p_series(arb_poly_t res, const arb_poly_t f, slong n, slong prec)
```

Computes the power series $\log(1 + f)$, with better accuracy when the constant term of *f* is small.

```
void _arb_poly_atan_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

```
void arb_poly_atan_series(arb_poly_t res, const arb_poly_t f, slong n, slong prec)
```

```
void _arb_poly_asin_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

```
void arb_poly_asin_series(arb_poly_t res, const arb_poly_t f, slong n, slong prec)
```

```
void _arb_poly_acos_series(arb_ptr res, arb_srcptr f, slong flen, slong n, slong prec)
```

```
void arb_poly_acos_series(arb_poly_t res, const arb_poly_t f, slong n, slong prec)
```

Sets *res* respectively to the power series inverse tangent, inverse sine and inverse cosine of *f*, truncated to length *n*.

Uses the formulas

$$\begin{aligned}\tan^{-1}(f(x)) &= \int f'(x)/(1 + f(x)^2)dx, \\ \sin^{-1}(f(x)) &= \int f'(x)/(1 - f(x)^2)^{1/2}dx, \\ \cos^{-1}(f(x)) &= - \int f'(x)/(1 - f(x)^2)^{1/2}dx,\end{aligned}$$

adding the inverse function of the constant term in *f* as the constant of integration.

The underscore methods supports aliasing of the input and output arrays. They require that *flen* and *n* are greater than zero.

```
void _arb_poly_exp_series_basecase(arb_ptr f, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_exp_series_basecase(arb_poly_t f, const arb_poly_t h, slong n, slong prec)
```

```
void _arb_poly_exp_series(arb_ptr f, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_exp_series(arb_poly_t f, const arb_poly_t h, slong n, slong prec)
```

Sets *f* to the power series exponential of *h*, truncated to length *n*.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where *m* is the length of *h*.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

```
void _arb_poly_sin_cos_series(arb_ptr s, arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec)
```

void **arb_poly_sin_cos_series**(*arb_poly_t* s, *arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)
 Sets *s* and *c* to the power series sine and cosine of *h*, computed simultaneously. The underscore method supports aliasing and requires the lengths to be nonzero.

void **_arb_poly_sin_series**(*arb_ptr* s, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_sin_series**(*arb_poly_t* s, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_cos_series**(*arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_cos_series**(*arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

Respectively evaluates the power series sine or cosine. These functions simply wrap **_arb_poly_sin_cos_series()**. The underscore methods support aliasing and require the lengths to be nonzero.

void **_arb_poly_tan_series**(*arb_ptr* g, *arb_srcptr* h, *slong* hlen, *slong* len, *slong* prec)

void **arb_poly_tan_series**(*arb_poly_t* g, const *arb_poly_t* h, *slong* n, *slong* prec)

Sets *g* to the power series tangent of *h*.

For small *n* takes the quotient of the sine and cosine as computed using the basecase algorithm. For large *n*, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void **_arb_poly_sin_cos_pi_series**(*arb_ptr* s, *arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_sin_cos_pi_series**(*arb_poly_t* s, *arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_sin_pi_series**(*arb_ptr* s, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_sin_pi_series**(*arb_poly_t* s, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_cos_pi_series**(*arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_cos_pi_series**(*arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_cot_pi_series**(*arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_cot_pi_series**(*arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

Compute the respective trigonometric functions of the input multiplied by π .

void **_arb_poly_sinh_cosh_series_basecase**(*arb_ptr* s, *arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_sinh_cosh_series_basecase**(*arb_poly_t* s, *arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_sinh_cosh_series_exponential**(*arb_ptr* s, *arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_sinh_cosh_series_exponential**(*arb_poly_t* s, *arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_sinh_cosh_series**(*arb_ptr* s, *arb_ptr* c, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_sinh_cosh_series**(*arb_poly_t* s, *arb_poly_t* c, const *arb_poly_t* h, *slong* n, *slong* prec)

void **_arb_poly_sinh_series**(*arb_ptr* s, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

```
void arb_poly_sinh_series(arb_poly_t s, const arb_poly_t h, slong n, slong prec)
```

```
void _arb_poly_cosh_series(arb_ptr c, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_cosh_series(arb_poly_t c, const arb_poly_t h, slong n, slong prec)
```

Sets s and c respectively to the hyperbolic sine and cosine of the power series h , truncated to length n .

The implementations mirror those for sine and cosine, except that the *exponential* version computes both functions using the exponential function instead of the hyperbolic tangent.

```
void _arb_poly_sinc_series(arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_sinc_series(arb_poly_t s, const arb_poly_t h, slong n, slong prec)
```

Sets c to the sinc function of the power series h , truncated to length n .

```
void _arb_poly_sinc_pi_series(arb_ptr s, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_sinc_pi_series(arb_poly_t s, const arb_poly_t h, slong n, slong prec)
```

Compute the sinc function of the input multiplied by π .

9.11.18 Lambert W function

```
void _arb_poly_lambertw_series(arb_ptr res, arb_srcptr z, slong zlen, int flags, slong len, slong prec)
```

```
void arb_poly_lambertw_series(arb_poly_t res, const arb_poly_t z, int flags, slong len, slong prec)
```

Sets res to the Lambert W function of the power series z . If $flags$ is 0, the principal branch is computed; if $flags$ is 1, the second real branch $W_{-1}(z)$ is computed. The underscore method allows aliasing, but assumes that the lengths are nonzero.

9.11.19 Gamma function and factorials

```
void _arb_poly_gamma_series(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_gamma_series(arb_poly_t res, const arb_poly_t h, slong n, slong prec)
```

```
void _arb_poly_rgamma_series(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_rgamma_series(arb_poly_t res, const arb_poly_t h, slong n, slong prec)
```

```
void _arb_poly_lgamma_series(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_lgamma_series(arb_poly_t res, const arb_poly_t h, slong n, slong prec)
```

```
void _arb_poly_digamma_series(arb_ptr res, arb_srcptr h, slong hlen, slong n, slong prec)
```

```
void arb_poly_digamma_series(arb_poly_t res, const arb_poly_t h, slong n, slong prec)
```

Sets res to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log \Gamma(h(x))$, $\psi(h(x))$, truncated to length n .

These functions first generate the Taylor series at the constant term of h , and then call `_arb_poly_compose_series()`. The Taylor coefficients are generated using the Riemann zeta function if the constant term of h is a small integer, and with Stirling's series otherwise.

The underscore methods support aliasing of the input and output arrays, and require that $hlen$ and n are greater than zero.

```
void _arb_poly_rising_ui_series(arb_ptr res, arb_srcptr f, slong flen, ulong r, slong trunc, slong prec)
```

void **arb_poly_rising_ui_series**(*arb_poly_t* res, const *arb_poly_t* f, *ulong* r, *slong* trunc, *slong* prec)

Sets *res* to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length *trunc*. The underscore method assumes that *flen*, *r* and *trunc* are at least 1, and does not support aliasing. Uses binary splitting.

9.11.20 Zeta function

void **arb_poly_zeta_series**(*arb_poly_t* res, const *arb_poly_t* s, const *arb_t* a, int deflate, *slong* n, *slong* prec)

Sets *res* to the Hurwitz zeta function $\zeta(s, a)$ where *s* a power series and *a* is a constant, truncated to length *n*. To evaluate the usual Riemann zeta function, set *a* = 1.

If *deflate* is nonzero, evaluates $\zeta(s, a) + 1/(1-s)$, which is well-defined as a limit when the constant term of *s* is 1. In particular, expanding $\zeta(s, a) + 1/(1-s)$ with $s = 1+x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.$$

If *a* = 1, this implementation uses the reflection formula if the midpoint of the constant term of *s* is negative.

void **_arb_poly_riemann_siegel_theta_series**(*arb_ptr* res, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_riemann_siegel_theta_series**(*arb_poly_t* res, const *arb_poly_t* h, *slong* n, *slong* prec)

Sets *res* to the series expansion of the Riemann-Siegel theta function

$$\theta(h) = \arg \left(\Gamma \left(\frac{2ih+1}{4} \right) \right) - \frac{\log \pi}{2} h$$

where the argument of the gamma function is chosen continuously as the imaginary part of the log gamma function.

The underscore method does not support aliasing of the input and output arrays, and requires that the lengths are greater than zero.

void **_arb_poly_riemann_siegel_z_series**(*arb_ptr* res, *arb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **arb_poly_riemann_siegel_z_series**(*arb_poly_t* res, const *arb_poly_t* h, *slong* n, *slong* prec)

Sets *res* to the series expansion of the Riemann-Siegel Z-function

$$Z(h) = e^{i\theta(h)} \zeta(1/2 + ih).$$

The zeros of the Z-function on the real line precisely correspond to the imaginary parts of the zeros of the Riemann zeta function on the critical line.

The underscore method supports aliasing of the input and output arrays, and requires that the lengths are greater than zero.

9.11.21 Root-finding

void `_arb_poly_root_bound_fujiwara`(*mag_t* bound, *arb_srcptr* poly, *slong* len)

void `arb_poly_root_bound_fujiwara`(*mag_t* bound, *arb_poly_t* poly)

Sets *bound* to an upper bound for the magnitude of all the complex roots of *poly*. Uses Fujiwara's bound

$$2 \max \left\{ \left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{1/2}, \dots, \left| \frac{a_1}{a_n} \right|^{1/(n-1)}, \left| \frac{a_0}{2a_n} \right|^{1/n} \right\}$$

where $a_0, \dots, a_n$ are the coefficients of *poly*.

void `_arb_poly_newton_convergence_factor`(*arf_t* convergence_factor, *arb_srcptr* poly, *slong* len, const *arb_t* convergence_interval, *slong* prec)

Given an interval I specified by *convergence_interval*, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, where f is the polynomial defined by the coefficients $\{poly, len\}$. The bound is obtained by evaluating $f'(I)$ and $f''(I)$ directly. If f has large coefficients, I must be extremely precise in order to get a finite factor.

int `_arb_poly_newton_step`(*arb_t* xnew, *arb_srcptr* poly, *slong* len, const *arb_t* x, const *arb_t* convergence_interval, const *arf_t* convergence_factor, *slong* prec)

Performs a single step with Newton's method.

The input consists of the polynomial f specified by the coefficients $\{poly, len\}$, an interval $x = [m - r, m + r]$ known to contain a single root of f , an interval I (*convergence_interval*) containing x with an associated bound (*convergence_factor*) for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, and a working precision *prec*.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of *prec* bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $m' < m$. If both conditions are satisfied, we set *xnew* to x' and return nonzero. If either condition fails, we set *xnew* to x and return zero, indicating that no progress was made.

void `_arb_poly_newton_refine_root`(*arb_t* r, *arb_srcptr* poly, *slong* len, const *arb_t* start, const *arb_t* convergence_interval, const *arf_t* convergence_factor, *slong* eval_extra_prec, *slong* prec)

Refines a precise estimate of a polynomial root to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for `_arb_poly_newton_step`, except for the precision parameters: *prec* is the target accuracy and *eval_extra_prec* is the estimated number of guard bits that need to be added to evaluate the polynomial accurately close to the root (typically, if the polynomial has large coefficients of alternating signs, this needs to be approximately the bit size of the coefficients).

9.11.22 Other special polynomials

void `_arb_poly_swinnerton_dyer_ui`(*arb_ptr* poly, *ulong* n, *slong* trunc, *slong* prec)

void `arb_poly_swinnerton_dyer_ui`(*arb_poly_t* poly, *ulong* n, *slong* prec)

Computes the Swinnerton-Dyer polynomial S_n , which has degree 2^n and is the rational minimal polynomial of the sum of the square roots of the first n prime numbers.

If *prec* is set to zero, a precision is chosen automatically such that `arb_poly_get_unique_fmpz_poly()` should be successful. Otherwise a working precision of *prec* bits is used.

The underscore version accepts an additional *trunc* parameter. Even when computing a truncated polynomial, the array *poly* must have room for $2^n + 1$ coefficients, used as temporary space.

9.12 `acb_poly.h` – polynomials over the complex numbers

An `acb_poly_t` represents a polynomial over the complex numbers, implemented as an array of coefficients of type `acb_struct`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

9.12.1 Types, macros and constants

type `acb_poly_struct`

type `acb_poly_t`

Contains a pointer to an array of coefficients (`coeffs`), the used length (`length`), and the allocated size of the array (`alloc`).

An `acb_poly_t` is defined as an array of length one of type `acb_poly_struct`, permitting an `acb_poly_t` to be passed by reference.

9.12.2 Memory management

void `acb_poly_init(acb_poly_t poly)`

Initializes the polynomial for use, setting it to the zero polynomial.

void `acb_poly_clear(acb_poly_t poly)`

Clears the polynomial, deallocating all coefficients and the coefficient array.

void `acb_poly_fit_length(acb_poly_t poly, slong len)`

Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void `_acb_poly_set_length(acb_poly_t poly, slong len)`

Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void `_acb_poly_normalise(acb_poly_t poly)`

Strips any trailing coefficients which are identical to zero.

void `acb_poly_swap(acb_poly_t poly1, acb_poly_t poly2)`

Swaps *poly1* and *poly2* efficiently.

slong `acb_poly_allocated_bytes(const acb_poly_t x)`

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(acb_poly_struct)` to get the size of the object as a whole.

9.12.3 Basic properties and manipulation

slong `acb_poly_length(const acb_poly_t poly)`

Returns the length of *poly*, i.e. zero if *poly* is identically zero, and otherwise one more than the index of the highest term that is not identically zero.

slong `acb_poly_degree(const acb_poly_t poly)`

Returns the degree of *poly*, defined as one less than its length. Note that if one or several leading coefficients are balls containing zero, this value can be larger than the true degree of the exact polynomial represented by *poly*, so the return value of this function is effectively an upper bound.

int **acb_poly_is_zero**(const *acb_poly_t* poly)

int **acb_poly_is_one**(const *acb_poly_t* poly)

int **acb_poly_is_x**(const *acb_poly_t* poly)

Returns 1 if *poly* is exactly the polynomial 0, 1 or x respectively. Returns 0 otherwise.

void **acb_poly_zero**(*acb_poly_t* poly)

Sets *poly* to the zero polynomial.

void **acb_poly_one**(*acb_poly_t* poly)

Sets *poly* to the constant polynomial 1.

void **acb_poly_set**(*acb_poly_t* dest, const *acb_poly_t* src)

Sets *dest* to a copy of *src*.

void **acb_poly_set_round**(*acb_poly_t* dest, const *acb_poly_t* src, *slong* prec)

Sets *dest* to a copy of *src*, rounded to *prec* bits.

void **acb_poly_set_trunc**(*acb_poly_t* dest, const *acb_poly_t* src, *slong* n)

void **acb_poly_set_trunc_round**(*acb_poly_t* dest, const *acb_poly_t* src, *slong* n, *slong* prec)

Sets *dest* to a copy of *src*, truncated to length *n* and rounded to *prec* bits.

void **acb_poly_set_coeff_si**(*acb_poly_t* poly, *slong* n, *slong* c)

void **acb_poly_set_coeff_acb**(*acb_poly_t* poly, *slong* n, const *acb_t* c)

Sets the coefficient with index *n* in *poly* to the value *c*. We require that *n* is nonnegative.

void **acb_poly_get_coeff_acb**(*acb_t* v, const *acb_poly_t* poly, *slong* n)

Sets *v* to the value of the coefficient with index *n* in *poly*. We require that *n* is nonnegative.

acb_poly_get_coeff_ptr(poly, n)

Given $n \geq 0$, returns a pointer to coefficient *n* of *poly*, or *NULL* if *n* exceeds the length of *poly*.

void **_acb_poly_shift_right**(*acb_ptr* res, *acb_srcptr* poly, *slong* len, *slong* n)

void **acb_poly_shift_right**(*acb_poly_t* res, const *acb_poly_t* poly, *slong* n)

Sets *res* to *poly* divided by x^n , throwing away the lower coefficients. We require that *n* is nonnegative.

void **_acb_poly_shift_left**(*acb_ptr* res, *acb_srcptr* poly, *slong* len, *slong* n)

void **acb_poly_shift_left**(*acb_poly_t* res, const *acb_poly_t* poly, *slong* n)

Sets *res* to *poly* multiplied by x^n . We require that *n* is nonnegative.

void **acb_poly_truncate**(*acb_poly_t* poly, *slong* n)

Truncates *poly* to have length at most *n*, i.e. degree strictly smaller than *n*. We require that *n* is nonnegative.

slong **acb_poly_valuation**(const *acb_poly_t* poly)

Returns the degree of the lowest term that is not exactly zero in *poly*. Returns -1 if *poly* is the zero polynomial.

9.12.4 Input and output

void **acb_poly_printd**(const *acb_poly_t* poly, *slong* digits)

Prints the polynomial as an array of coefficients, printing each coefficient using *acb_printd*.

void **acb_poly_fprintd**(FILE *file, const *acb_poly_t* poly, *slong* digits)

Prints the polynomial as an array of coefficients to the stream *file*, printing each coefficient using *acb_fprintd*.

9.12.5 Random generation

void **acb_poly_randtest**(*acb_poly_t* poly, *flint_rand_t* state, *slong* len, *slong* prec, *slong* mag_bits)

Creates a random polynomial with length at most *len*.

9.12.6 Comparisons

int **acb_poly_equal**(const *acb_poly_t* A, const *acb_poly_t* B)

Returns nonzero iff *A* and *B* are identical as interval polynomials.

int **acb_poly_contains**(const *acb_poly_t* poly1, const *acb_poly_t* poly2)

int **acb_poly_contains_fmpz_poly**(const *acb_poly_t* poly1, const *fmpz_poly_t* poly2)

int **acb_poly_contains_fmpq_poly**(const *acb_poly_t* poly1, const *fmpq_poly_t* poly2)

Returns nonzero iff *poly2* is contained in *poly1*.

int **_acb_poly_overlaps**(*acb_srcptr* poly1, *slong* len1, *acb_srcptr* poly2, *slong* len2)

int **acb_poly_overlaps**(const *acb_poly_t* poly1, const *acb_poly_t* poly2)

Returns nonzero iff *poly1* overlaps with *poly2*. The underscore function requires that *len1* is at least as large as *len2*.

int **acb_poly_get_unique_fmpz_poly**(*fmpz_poly_t* z, const *acb_poly_t* x)

If *x* contains a unique integer polynomial, sets *z* to that value and returns nonzero. Otherwise (if *x* represents no integers or more than one integer), returns zero, possibly partially modifying *z*.

int **acb_poly_is_real**(const *acb_poly_t* poly)

Returns nonzero iff all coefficients in *poly* have zero imaginary part.

9.12.7 Conversions

void **acb_poly_set_fmpz_poly**(*acb_poly_t* poly, const *fmpz_poly_t* re, *slong* prec)

void **acb_poly_set2_fmpz_poly**(*acb_poly_t* poly, const *fmpz_poly_t* re, const *fmpz_poly_t* im, *slong* prec)

void **acb_poly_set_arb_poly**(*acb_poly_t* poly, const *arb_poly_t* re)

void **acb_poly_set2_arb_poly**(*acb_poly_t* poly, const *arb_poly_t* re, const *arb_poly_t* im)

void **acb_poly_set_fmpq_poly**(*acb_poly_t* poly, const *fmpq_poly_t* re, *slong* prec)

void **acb_poly_set2_fmpq_poly**(*acb_poly_t* poly, const *fmpq_poly_t* re, const *fmpq_poly_t* im, *slong* prec)

Sets *poly* to the given real part *re* plus the imaginary part *im*, both rounded to *prec* bits.

void **acb_poly_set_acb**(*acb_poly_t* poly, const *acb_t* src)

void **acb_poly_set_si**(*acb_poly_t* poly, *slong* src)
Sets *poly* to *src*.

9.12.8 Bounds

void **_acb_poly_majorant**(*arb_ptr* res, *acb_srcptr* poly, *slong* len, *slong* prec)

void **acb_poly_majorant**(*arb_poly_t* res, const *acb_poly_t* poly, *slong* prec)
Sets *res* to an exact real polynomial whose coefficients are upper bounds for the absolute values of the coefficients in *poly*, rounded to *prec* bits.

9.12.9 Arithmetic

void **_acb_poly_add**(*acb_ptr* C, *acb_srcptr* A, *slong* lenA, *acb_srcptr* B, *slong* lenB, *slong* prec)
Sets $\{C, \max(\text{lenA}, \text{lenB})\}$ to the sum of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$. Allows aliasing of the input and output operands.

void **acb_poly_add**(*acb_poly_t* C, const *acb_poly_t* A, const *acb_poly_t* B, *slong* prec)

void **acb_poly_add_si**(*acb_poly_t* C, const *acb_poly_t* A, *slong* B, *slong* prec)
Sets *C* to the sum of *A* and *B*.

void **_acb_poly_sub**(*acb_ptr* C, *acb_srcptr* A, *slong* lenA, *acb_srcptr* B, *slong* lenB, *slong* prec)
Sets $\{C, \max(\text{lenA}, \text{lenB})\}$ to the difference of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$. Allows aliasing of the input and output operands.

void **acb_poly_sub**(*acb_poly_t* C, const *acb_poly_t* A, const *acb_poly_t* B, *slong* prec)
Sets *C* to the difference of *A* and *B*.

void **acb_poly_add_series**(*acb_poly_t* C, const *acb_poly_t* A, const *acb_poly_t* B, *slong* len, *slong* prec)
Sets *C* to the sum of *A* and *B*, truncated to length *len*.

void **acb_poly_sub_series**(*acb_poly_t* C, const *acb_poly_t* A, const *acb_poly_t* B, *slong* len, *slong* prec)
Sets *C* to the difference of *A* and *B*, truncated to length *len*.

void **acb_poly_neg**(*acb_poly_t* C, const *acb_poly_t* A)
Sets *C* to the negation of *A*.

void **acb_poly_scalar_mul_2exp_si**(*acb_poly_t* C, const *acb_poly_t* A, *slong* c)
Sets *C* to *A* multiplied by 2^c .

void **acb_poly_scalar_mul**(*acb_poly_t* C, const *acb_poly_t* A, const *acb_t* c, *slong* prec)
Sets *C* to *A* multiplied by *c*.

void **acb_poly_scalar_div**(*acb_poly_t* C, const *acb_poly_t* A, const *acb_t* c, *slong* prec)
Sets *C* to *A* divided by *c*.

void **_acb_poly_mullov_classical**(*acb_ptr* C, *acb_srcptr* A, *slong* lenA, *acb_srcptr* B, *slong* lenB, *slong* n, *slong* prec)

void **_acb_poly_mullov_transpose**(*acb_ptr* C, *acb_srcptr* A, *slong* lenA, *acb_srcptr* B, *slong* lenB, *slong* n, *slong* prec)

void **_acb_poly_mullov_transpose_gauss**(*acb_ptr* C, *acb_srcptr* A, *slong* lenA, *acb_srcptr* B, *slong* lenB, *slong* n, *slong* prec)

```
void _acb_poly_mullow(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong n,
                    slong prec)
```

Sets $\{C, n\}$ to the product of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$, truncated to length n . The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$, $n > 0$, $\text{lenA} + \text{lenB} - 1 \geq n$.

The *classical* version uses a plain loop.

The *transpose* version evaluates the product using four real polynomial multiplications (via `_arb_poly_mullow()`).

The *transpose_gauss* version evaluates the product using three real polynomial multiplications. This is almost always faster than *transpose*, but has worse numerical stability when the coefficients vary in magnitude.

The default function `_acb_poly_mullow()` automatically switches between *classical* and *transpose* multiplication.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

```
void acb_poly_mullow_classical(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n,
                             slong prec)
```

```
void acb_poly_mullow_transpose(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n,
                              slong prec)
```

```
void acb_poly_mullow_transpose_gauss(acb_poly_t C, const acb_poly_t A, const acb_poly_t B,
                                     slong n, slong prec)
```

```
void acb_poly_mullow(acb_poly_t C, const acb_poly_t A, const acb_poly_t B, slong n, slong prec)
```

Sets C to the product of A and B , truncated to length n . If the same variable is passed for A and B , sets C to the square of A truncated to length n .

```
void _acb_poly_mul(acb_ptr C, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)
```

Sets $\{C, \text{lenA} + \text{lenB} - 1\}$ to the product of $\{A, \text{lenA}\}$ and $\{B, \text{lenB}\}$. The output is not allowed to be aliased with either of the inputs. We require $\text{lenA} \geq \text{lenB} > 0$. This function is implemented as a simple wrapper for `_acb_poly_mullow()`.

If the input pointers are identical (and the lengths are the same), they are assumed to represent the same polynomial, and its square is computed.

```
void acb_poly_mul(acb_poly_t C, const acb_poly_t A1, const acb_poly_t B2, slong prec)
```

Sets C to the product of A and B . If the same variable is passed for A and B , sets C to the square of A .

```
void _acb_poly_mulmid_transpose(acb_ptr z, acb_srcptr x, slong xlen, acb_srcptr y, slong ylen,
                              slong nlo, slong nhi, slong prec)
```

```
void acb_poly_mulmid_transpose(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2,
                              slong nlo, slong nhi, slong prec)
```

```
void _acb_poly_mulmid_classical(acb_ptr z, acb_srcptr x, slong xlen, acb_srcptr y, slong ylen,
                              slong nlo, slong nhi, slong prec)
```

```
void acb_poly_mulmid_classical(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2,
                              slong nlo, slong nhi, slong prec)
```

```
void _acb_poly_mulmid(acb_ptr z, acb_srcptr x, slong xlen, acb_srcptr y, slong ylen, slong nlo, slong
                    nhi, slong prec)
```

```
void acb_poly_mulmid(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, slong nlo,
                    slong nhi, slong prec)
```

Analogous to *mullow* functions, but compute the product truncated at length nhi and right-shifted by nlo .

`void _acb_poly_inv_series(acb_ptr Qinv, acb_srcptr Q, slong Qlen, slong len, slong prec)`
Sets $\{Q_{inv}, len\}$ to the power series inverse of $\{Q, Qlen\}$. Uses Newton iteration.

`void acb_poly_inv_series(acb_poly_t Qinv, const acb_poly_t Q, slong n, slong prec)`
Sets Q_{inv} to the power series inverse of Q .

`void _acb_poly_div_series(acb_ptr Q, acb_srcptr A, slong Alen, acb_srcptr B, slong Blen, slong n, slong prec)`
Sets $\{Q, n\}$ to the power series quotient of $\{A, Alen\}$ by $\{B, Blen\}$. Uses Newton iteration followed by multiplication.

`void acb_poly_div_series(acb_poly_t Q, const acb_poly_t A, const acb_poly_t B, slong n, slong prec)`
Sets Q to the power series quotient A divided by B , truncated to length n .

`void _acb_poly_div(acb_ptr Q, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)`

`void _acb_poly_rem(acb_ptr R, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)`

`void _acb_poly_divrem(acb_ptr Q, acb_ptr R, acb_srcptr A, slong lenA, acb_srcptr B, slong lenB, slong prec)`

`int acb_poly_divrem(acb_poly_t Q, acb_poly_t R, const acb_poly_t A, const acb_poly_t B, slong prec)`
Performs polynomial division with remainder, computing a quotient Q and a remainder R such that $A = BQ + R$. The implementation reverses the inputs and performs power series division.
If the leading coefficient of B contains zero (or if B is identically zero), returns 0 indicating failure without modifying the outputs. Otherwise returns nonzero.

`void _acb_poly_div_root(acb_ptr Q, acb_t R, acb_srcptr A, slong len, const acb_t c, slong prec)`
Divides A by the polynomial $x - c$, computing the quotient Q as well as the remainder $R = f(c)$.

9.12.10 Composition

`void _acb_poly_taylor_shift(acb_ptr g, const acb_t c, slong n, slong prec)`

`void acb_poly_taylor_shift(acb_poly_t g, const acb_poly_t f, const acb_t c, slong prec)`
Sets g to the Taylor shift $f(x + c)$. The underscore method acts in-place on $g = f$ which has length n .

`void _acb_poly_compose(acb_ptr res, acb_srcptr poly1, slong len1, acb_srcptr poly2, slong len2, slong prec)`

`void acb_poly_compose(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, slong prec)`
Sets res to the composition $h(x) = f(g(x))$ where f is given by $poly1$ and g is given by $poly2$. The underscore method does not support aliasing of the output with either input polynomial.

`void _acb_poly_compose_series(acb_ptr res, acb_srcptr poly1, slong len1, acb_srcptr poly2, slong len2, slong n, slong prec)`

`void acb_poly_compose_series(acb_poly_t res, const acb_poly_t poly1, const acb_poly_t poly2, slong n, slong prec)`
Sets res to the power series composition $h(x) = f(g(x))$ truncated to order $O(x^n)$ where f is given by $poly1$ and g is given by $poly2$. Wraps `_gr_poly_compose_series()` which chooses automatically between various algorithms.
We require that the constant term in $g(x)$ is exactly zero. The underscore method does not support aliasing of the output with either input polynomial.

`void _acb_poly_revert_series(acb_ptr h, acb_srcptr f, slong flen, slong n, slong prec)`

void **acb_poly_revert_series**(*acb_poly_t* h, const *acb_poly_t* f, *slong* n, *slong* prec)

Sets h to the power series reversion of f , i.e. the expansion of the compositional inverse function $f^{-1}(x)$, truncated to order $O(x^n)$. Wraps `_gr_poly_revert_series()` which chooses automatically between various algorithms.

We require that the constant term in f is exactly zero and that the linear term is nonzero. The underscore method assumes that $flen$ is at least 2, and do not support aliasing.

9.12.11 Evaluation

void **_acb_poly_evaluate_horner**(*acb_t* y, *acb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **acb_poly_evaluate_horner**(*acb_t* y, const *acb_poly_t* f, const *acb_t* x, *slong* prec)

void **_acb_poly_evaluate_rectangular**(*acb_t* y, *acb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **acb_poly_evaluate_rectangular**(*acb_t* y, const *acb_poly_t* f, const *acb_t* x, *slong* prec)

void **_acb_poly_evaluate**(*acb_t* y, *acb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **acb_poly_evaluate**(*acb_t* y, const *acb_poly_t* f, const *acb_t* x, *slong* prec)

Sets $y = f(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

void **_acb_poly_evaluate2_horner**(*acb_t* y, *acb_t* z, *acb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **acb_poly_evaluate2_horner**(*acb_t* y, *acb_t* z, const *acb_poly_t* f, const *acb_t* x, *slong* prec)

void **_acb_poly_evaluate2_rectangular**(*acb_t* y, *acb_t* z, *acb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **acb_poly_evaluate2_rectangular**(*acb_t* y, *acb_t* z, const *acb_poly_t* f, const *acb_t* x, *slong* prec)

void **_acb_poly_evaluate2**(*acb_t* y, *acb_t* z, *acb_srcptr* f, *slong* len, const *acb_t* x, *slong* prec)

void **acb_poly_evaluate2**(*acb_t* y, *acb_t* z, const *acb_poly_t* f, const *acb_t* x, *slong* prec)

Sets $y = f(x)$, $z = f'(x)$, evaluated respectively using Horner's rule, rectangular splitting, and an automatic algorithm choice.

When Horner's rule is used, the only advantage of evaluating the function and its derivative simultaneously is that one does not have to generate the derivative polynomial explicitly. With the rectangular splitting algorithm, the powers can be reused, making simultaneous evaluation slightly faster.

9.12.12 Product trees

void **_acb_poly_product_roots**(*acb_ptr* poly, *acb_srcptr* xs, *slong* n, *slong* prec)

void **acb_poly_product_roots**(*acb_poly_t* poly, *acb_srcptr* xs, *slong* n, *slong* prec)

Generates the polynomial $(x - x_0)(x - x_1) \cdots (x - x_{n-1})$.

acb_ptr ***_acb_poly_tree_alloc**(*slong* len)

Returns an initialized data structured capable of representing a remainder tree (product tree) of len roots.

void **_acb_poly_tree_free**(*acb_ptr* *tree, *slong* len)

Deallocates a tree structure as allocated using `_acb_poly_tree_alloc`.

```
void _acb_poly_tree_build(acb_ptr *tree, acb_srcptr roots, slong len, slong prec)
```

Constructs a product tree from a given array of *len* roots. The tree structure must be pre-allocated to the specified length using `_acb_poly_tree_alloc()`.

9.12.13 Multipoint evaluation

```
void _acb_poly_evaluate_vec_iter(acb_ptr ys, acb_srcptr poly, slong plen, acb_srcptr xs, slong n, slong prec)
```

```
void acb_poly_evaluate_vec_iter(acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, slong n, slong prec)
```

Evaluates the polynomial simultaneously at *n* given points, calling `_acb_poly_evaluate()` repeatedly.

```
void _acb_poly_evaluate_vec_fast_precomp(acb_ptr vs, acb_srcptr poly, slong plen, acb_ptr *tree, slong len, slong prec)
```

```
void _acb_poly_evaluate_vec_fast(acb_ptr ys, acb_srcptr poly, slong plen, acb_srcptr xs, slong n, slong prec)
```

```
void acb_poly_evaluate_vec_fast(acb_ptr ys, const acb_poly_t poly, acb_srcptr xs, slong n, slong prec)
```

Evaluates the polynomial simultaneously at *n* given points, using fast multipoint evaluation.

9.12.14 Interpolation

```
void _acb_poly_interpolate_newton(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

```
void acb_poly_interpolate_newton(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Recovers the unique polynomial of length at most *n* that interpolates the given *x* and *y* values. This implementation first interpolates in the Newton basis and then converts back to the monomial basis.

```
void _acb_poly_interpolate_barycentric(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

```
void acb_poly_interpolate_barycentric(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Recovers the unique polynomial of length at most *n* that interpolates the given *x* and *y* values. This implementation uses the barycentric form of Lagrange interpolation.

```
void _acb_poly_interpolation_weights(acb_ptr w, acb_ptr *tree, slong len, slong prec)
```

```
void _acb_poly_interpolate_fast_precomp(acb_ptr poly, acb_srcptr ys, acb_ptr *tree, acb_srcptr weights, slong len, slong prec)
```

```
void _acb_poly_interpolate_fast(acb_ptr poly, acb_srcptr xs, acb_srcptr ys, slong len, slong prec)
```

```
void acb_poly_interpolate_fast(acb_poly_t poly, acb_srcptr xs, acb_srcptr ys, slong n, slong prec)
```

Recovers the unique polynomial of length at most *n* that interpolates the given *x* and *y* values, using fast Lagrange interpolation. The precomp function takes a precomputed product tree over the *x* values and a vector of interpolation weights as additional inputs.

9.12.15 Differentiation

`void _acb_poly_derivative(acb_ptr res, acb_srcptr poly, slong len, slong prec)`
 Sets $\{res, len - 1\}$ to the derivative of $\{poly, len\}$. Allows aliasing of the input and output.

`void acb_poly_derivative(acb_poly_t res, const acb_poly_t poly, slong prec)`
 Sets *res* to the derivative of *poly*.

`void _acb_poly_nth_derivative(acb_ptr res, acb_srcptr poly, ulong n, slong len, slong prec)`
 Sets $\{res, len - n\}$ to the *n*th derivative of $\{poly, len\}$. Does nothing if $len \leq n$. Allows aliasing of the input and output.

`void acb_poly_nth_derivative(acb_poly_t res, const acb_poly_t poly, ulong n, slong prec)`
 Sets *res* to the *n*th derivative of *poly*.

`void _acb_poly_integral(acb_ptr res, acb_srcptr poly, slong len, slong prec)`
 Sets $\{res, len\}$ to the integral of $\{poly, len - 1\}$. Allows aliasing of the input and output.

`void acb_poly_integral(acb_poly_t res, const acb_poly_t poly, slong prec)`
 Sets *res* to the integral of *poly*.

9.12.16 Transforms

`void _acb_poly_borel_transform(acb_ptr res, acb_srcptr poly, slong len, slong prec)`

`void acb_poly_borel_transform(acb_poly_t res, const acb_poly_t poly, slong prec)`
 Computes the Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k (a_k/k!)x^k$. The underscore method allows aliasing.

`void _acb_poly_inv_borel_transform(acb_ptr res, acb_srcptr poly, slong len, slong prec)`

`void acb_poly_inv_borel_transform(acb_poly_t res, const acb_poly_t poly, slong prec)`
 Computes the inverse Borel transform of the input polynomial, mapping $\sum_k a_k x^k$ to $\sum_k a_k k! x^k$. The underscore method allows aliasing.

`void _acb_poly_binomial_transform_basecase(acb_ptr b, acb_srcptr a, slong alen, slong len, slong prec)`

`void acb_poly_binomial_transform_basecase(acb_poly_t b, const acb_poly_t a, slong len, slong prec)`

`void _acb_poly_binomial_transform_convolution(acb_ptr b, acb_srcptr a, slong alen, slong len, slong prec)`

`void acb_poly_binomial_transform_convolution(acb_poly_t b, const acb_poly_t a, slong len, slong prec)`

`void _acb_poly_binomial_transform(acb_ptr b, acb_srcptr a, slong alen, slong len, slong prec)`

`void acb_poly_binomial_transform(acb_poly_t b, const acb_poly_t a, slong len, slong prec)`
 Computes the binomial transform of the input polynomial, truncating the output to length *len*. See [arb_poly_binomial_transform\(\)](#) for details.

The underscore methods do not support aliasing, and assume that the lengths are nonzero.

`void _acb_poly_graeffe_transform(acb_ptr b, acb_srcptr a, slong len, slong prec)`

void **acb_poly_graeffe_transform**(*acb_poly_t* b, const *acb_poly_t* a, *slong* prec)

Computes the Graeffe transform of input polynomial, which is of length *len*. See [*arb_poly_graeffe_transform\(\)*](#) for details.

The underscore method assumes that *a* and *b* are initialized, *a* is of length *len*, and *b* is of length at least *len*. Both methods allow aliasing.

9.12.17 Elementary functions

void **_acb_poly_pow_ui_trunc_binexp**(*acb_ptr* res, *acb_srcptr* f, *slong* flen, *ulong* exp, *slong* len, *slong* prec)

Sets $\{res, len\}$ to $\{f, flen\}$ raised to the power *exp*, truncated to length *len*. Requires that *len* is no longer than the length of the power as computed without truncation (i.e. no zero-padding is performed). Does not support aliasing of the input and output, and requires that *flen* and *len* are positive. Uses binary exponentiation.

void **acb_poly_pow_ui_trunc_binexp**(*acb_poly_t* res, const *acb_poly_t* poly, *ulong* exp, *slong* len, *slong* prec)

Sets *res* to *poly* raised to the power *exp*, truncated to length *len*. Uses binary exponentiation.

void **_acb_poly_pow_ui**(*acb_ptr* res, *acb_srcptr* f, *slong* flen, *ulong* exp, *slong* prec)

Sets *res* to $\{f, flen\}$ raised to the power *exp*. Does not support aliasing of the input and output, and requires that *flen* is positive.

void **acb_poly_pow_ui**(*acb_poly_t* res, const *acb_poly_t* poly, *ulong* exp, *slong* prec)

Sets *res* to *poly* raised to the power *exp*.

void **_acb_poly_pow_series**(*acb_ptr* h, *acb_srcptr* f, *slong* flen, *acb_srcptr* g, *slong* glen, *slong* len, *slong* prec)

Sets $\{h, len\}$ to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as *g* being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* and *glen* do not exceed *len*.

void **acb_poly_pow_series**(*acb_poly_t* h, const *acb_poly_t* f, const *acb_poly_t* g, *slong* len, *slong* prec)

Sets *h* to the power series $f(x)^{g(x)} = \exp(g(x) \log f(x))$ truncated to length *len*. This function detects special cases such as *g* being an exact small integer or $\pm 1/2$, and computes such powers more efficiently.

void **_acb_poly_pow_acb_series**(*acb_ptr* h, *acb_srcptr* f, *slong* flen, const *acb_t* g, *slong* len, *slong* prec)

Sets $\{h, len\}$ to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*. This function detects special cases such as *g* being an exact small integer or $\pm 1/2$, and computes such powers more efficiently. This function does not support aliasing of the output with either of the input operands. It requires that all lengths are positive, and assumes that *flen* does not exceed *len*.

void **acb_poly_pow_acb_series**(*acb_poly_t* h, const *acb_poly_t* f, const *acb_t* g, *slong* len, *slong* prec)

Sets *h* to the power series $f(x)^g = \exp(g \log f(x))$ truncated to length *len*.

void **_acb_poly_sqrt_series**(*acb_ptr* g, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sqrt_series**(*acb_poly_t* g, const *acb_poly_t* h, *slong* n, *slong* prec)

Sets *g* to the power series square root of *h*, truncated to length *n*. Uses division-free Newton iteration for the reciprocal square root, followed by a multiplication.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and *n* are greater than zero.

void `_acb_poly_rsqrt_series`(*acb_ptr* *g*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void `acb_poly_rsqrt_series`(*acb_poly_t* *g*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets *g* to the reciprocal power series square root of *h*, truncated to length *n*. Uses division-free Newton iteration.

The underscore method does not support aliasing of the input and output arrays. It requires that *hlen* and *n* are greater than zero.

void `_acb_poly_log_series`(*acb_ptr* *res*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void `acb_poly_log_series`(*acb_poly_t* *res*, const *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)

Sets *res* to the power series logarithm of *f*, truncated to length *n*. Uses the formula $\log(f(x)) = \int f'(x)/f(x)dx$, adding the logarithm of the constant term in *f* as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and *n* are greater than zero.

void `_acb_poly_log1p_series`(*acb_ptr* *res*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void `acb_poly_log1p_series`(*acb_poly_t* *res*, const *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)

Computes the power series $\log(1 + f)$, with better accuracy when the constant term of *f* is small.

void `_acb_poly_atan_series`(*acb_ptr* *res*, *acb_srcptr* *f*, *slong* *flen*, *slong* *n*, *slong* *prec*)

void `acb_poly_atan_series`(*acb_poly_t* *res*, const *acb_poly_t* *f*, *slong* *n*, *slong* *prec*)

Sets *res* the power series inverse tangent of *f*, truncated to length *n*.

Uses the formula

$$\tan^{-1}(f(x)) = \int f'(x)/(1 + f(x)^2)dx,$$

adding the function of the constant term in *f* as the constant of integration.

The underscore method supports aliasing of the input and output arrays. It requires that *flen* and *n* are greater than zero.

void `_acb_poly_exp_series_basecase`(*acb_ptr* *f*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void `acb_poly_exp_series_basecase`(*acb_poly_t* *f*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

void `_acb_poly_exp_series`(*acb_ptr* *f*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void `acb_poly_exp_series`(*acb_poly_t* *f*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets *f* to the power series exponential of *h*, truncated to length *n*.

The basecase version uses a simple recurrence for the coefficients, requiring $O(nm)$ operations where *m* is the length of *h*.

The main implementation uses Newton iteration, starting from a small number of terms given by the basecase algorithm. The complexity is $O(M(n))$. Redundant operations in the Newton iteration are avoided by using the scheme described in [HZ2004].

The underscore methods support aliasing and allow the input to be shorter than the output, but require the lengths to be nonzero.

void `_acb_poly_exp_pi_i_series`(*acb_ptr* *f*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void `acb_poly_exp_pi_i_series`(*acb_poly_t* *f*, const *acb_poly_t* *h*, *slong* *n*, *slong* *prec*)

Sets *f* to the power series $\exp(\pi i h)$ truncated to length *n*. The underscore method supports aliasing and allows the input to be shorter than the output, but requires the lengths to be nonzero.

void `_acb_poly_sin_cos_series`(*acb_ptr* *s*, *acb_ptr* *c*, *acb_srcptr* *h*, *slong* *hlen*, *slong* *n*, *slong* *prec*)

void **acb_poly_sin_cos_series**(*acb_poly_t* s, *acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)
Sets *s* and *c* to the power series sine and cosine of *h*, computed simultaneously. The underscore method supports aliasing and requires the lengths to be nonzero.

void **_acb_poly_sin_series**(*acb_ptr* s, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sin_series**(*acb_poly_t* s, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_cos_series**(*acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_cos_series**(*acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

Respectively evaluates the power series sine or cosine. These functions simply wrap **_acb_poly_sin_cos_series()**. The underscore methods support aliasing and require the lengths to be nonzero.

void **_acb_poly_tan_series**(*acb_ptr* g, *acb_srcptr* h, *slong* hlen, *slong* len, *slong* prec)

void **acb_poly_tan_series**(*acb_poly_t* g, const *acb_poly_t* h, *slong* n, *slong* prec)

Sets *g* to the power series tangent of *h*.

For small *n* takes the quotient of the sine and cosine as computed using the basecase algorithm. For large *n*, uses Newton iteration to invert the inverse tangent series. The complexity is $O(M(n))$.

The underscore version does not support aliasing, and requires the lengths to be nonzero.

void **_acb_poly_sin_cos_pi_series**(*acb_ptr* s, *acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sin_cos_pi_series**(*acb_poly_t* s, *acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_sin_pi_series**(*acb_ptr* s, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sin_pi_series**(*acb_poly_t* s, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_cos_pi_series**(*acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_cos_pi_series**(*acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_cot_pi_series**(*acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_cot_pi_series**(*acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

Compute the respective trigonometric functions of the input multiplied by π .

void **_acb_poly_sinh_cosh_series_basecase**(*acb_ptr* s, *acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sinh_cosh_series_basecase**(*acb_poly_t* s, *acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_sinh_cosh_series_exponential**(*acb_ptr* s, *acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sinh_cosh_series_exponential**(*acb_poly_t* s, *acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_sinh_cosh_series**(*acb_ptr* s, *acb_ptr* c, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)

void **acb_poly_sinh_cosh_series**(*acb_poly_t* s, *acb_poly_t* c, const *acb_poly_t* h, *slong* n, *slong* prec)

void **_acb_poly_sinh_series**(*acb_ptr* s, *acb_srcptr* h, *slong* hlen, *slong* n, *slong* prec)


```
void acb_poly_sinh_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)
```

```
void _acb_poly_cosh_series(acb_ptr c, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_cosh_series(acb_poly_t c, const acb_poly_t h, slong n, slong prec)
```

Sets *s* and *c* respectively to the hyperbolic sine and cosine of the power series *h*, truncated to length *n*.

The implementations mirror those for sine and cosine, except that the *exponential* version computes both functions using the exponential function instead of the hyperbolic tangent.

```
void _acb_poly_sinc_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_sinc_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)
```

Sets *s* to the sinc function of the power series *h*, truncated to length *n*.

```
void _acb_poly_sinc_pi_series(acb_ptr s, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_sinc_pi_series(acb_poly_t s, const acb_poly_t h, slong n, slong prec)
```

Compute the sinc function of the input multiplied by π .

9.12.18 Lambert W function

```
void _acb_poly_lambertw_series(acb_ptr res, acb_srcptr z, slong zlen, const fmpz_t k, int flags,  
                             slong len, slong prec)
```

```
void acb_poly_lambertw_series(acb_poly_t res, const acb_poly_t z, const fmpz_t k, int flags, slong  
                             len, slong prec)
```

Sets *res* to branch *k* of the Lambert W function of the power series *z*. The argument *flags* is reserved for future use. The underscore method allows aliasing, but assumes that the lengths are nonzero.

9.12.19 Gamma function

```
void _acb_poly_gamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_gamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)
```

```
void _acb_poly_rgamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_rgamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)
```

```
void _acb_poly_lgamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_lgamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)
```

```
void _acb_poly_digamma_series(acb_ptr res, acb_srcptr h, slong hlen, slong n, slong prec)
```

```
void acb_poly_digamma_series(acb_poly_t res, const acb_poly_t h, slong n, slong prec)
```

Sets *res* to the series expansion of $\Gamma(h(x))$, $1/\Gamma(h(x))$, or $\log \Gamma(h(x))$, $\psi(h(x))$, truncated to length *n*.

These functions first generate the Taylor series at the constant term of *h*, and then call `_acb_poly_compose_series()`. The Taylor coefficients are generated using Stirling's series.

The underscore methods support aliasing of the input and output arrays, and require that *hlen* and *n* are greater than zero.

```
void _acb_poly_rising_ui_series(acb_ptr res, acb_srcptr f, slong flen, ulong r, slong trunc, slong  
                             prec)
```



```
void acb_poly_rising_ui_series(acb_poly_t res, const acb_poly_t f, ulong r, slong trunc, slong
                             prec)
```

Sets *res* to the rising factorial $(f)(f+1)(f+2)\cdots(f+r-1)$, truncated to length *trunc*. The underscore method assumes that *f**len*, *r* and *trunc* are at least 1, and does not support aliasing. Uses binary splitting.

9.12.20 Power sums

```
void _acb_poly_powsum_series_naive(acb_ptr z, const acb_t s, const acb_t a, const acb_t q, slong
                                   n, slong len, slong prec)
```

```
void _acb_poly_powsum_series_naive_threaded(acb_ptr z, const acb_t s, const acb_t a, const
                                             acb_t q, slong n, slong len, slong prec)
```

Computes

$$z = S(s, a, n) = \sum_{k=0}^{n-1} \frac{q^k}{(k+a)^{s+t}}$$

as a power series in *t* truncated to length *len*. This function evaluates the sum naively term by term. The *threaded* version splits the computation over the number of threads returned by *flint_get_num_threads()*.

```
void _acb_poly_powsum_one_series_sieved(acb_ptr z, const acb_t s, slong n, slong len, slong prec)
```

Computes

$$z = S(s, 1, n) \sum_{k=1}^n \frac{1}{k^{s+t}}$$

as a power series in *t* truncated to length *len*. This function stores a table of powers that have already been calculated, computing $(ij)^r$ as $i^r j^r$ whenever $k = ij$ is composite. As a further optimization, it groups all even *k* and evaluates the sum as a polynomial in $2^{-(s+t)}$. This scheme requires about $n/\log n$ powers, $n/2$ multiplications, and temporary storage of $n/6$ power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when *len* is small.

9.12.21 Zeta function

```
void _acb_poly_zeta_em_choose_param(mag_t bound, ulong *N, ulong *M, const acb_t s, const
                                    acb_t a, slong d, slong target, slong prec)
```

Chooses *N* and *M* for Euler-Maclaurin summation of the Hurwitz zeta function, using a default algorithm.

```
void _acb_poly_zeta_em_bound1(mag_t bound, const acb_t s, const acb_t a, slong N, slong M, slong
                              d, slong wp)
```

```
void _acb_poly_zeta_em_bound(arb_ptr vec, const acb_t s, const acb_t a, ulong N, ulong M, slong d,
                              slong wp)
```

Compute bounds for Euler-Maclaurin evaluation of the Hurwitz zeta function or its power series, using the formulas in [Joh2013].

```
void _acb_poly_zeta_em_tail_naive(acb_ptr z, const acb_t s, const acb_t Na, acb_srcptr Nasx,
                                  slong M, slong len, slong prec)
```

```
void _acb_poly_zeta_em_tail_bsplitt(acb_ptr z, const acb_t s, const acb_t Na, acb_srcptr Nasx,
                                     slong M, slong len, slong prec)
```

Evaluates the tail in the Euler-Maclaurin sum for the Hurwitz zeta function, respectively using the naive recurrence and binary splitting.

```
void _acb_poly_zeta_em_sum(acb_ptr z, const acb_t s, const acb_t a, int deflate, ulong N, ulong M,
                          slong d, slong prec)
```

Evaluates the truncated Euler-Maclaurin sum of order N, M for the length- d truncated Taylor series of the Hurwitz zeta function $\zeta(s, a)$ at s , using a working precision of $prec$ bits. With $a = 1$, this gives the usual Riemann zeta function.

If *deflate* is nonzero, $\zeta(s, a) - 1/(s - 1)$ is evaluated (which permits series expansion at $s = 1$).

```
void _acb_poly_zeta_cpx_series(acb_ptr z, const acb_t s, const acb_t a, int deflate, slong d, slong
                              prec)
```

Computes the series expansion of $\zeta(s + x, a)$ (or $\zeta(s + x, a) - 1/(s + x - 1)$ if *deflate* is nonzero) to order d .

This function wraps `_acb_poly_zeta_em_sum()`, automatically choosing default values for N, M using `_acb_poly_zeta_em_choose_param()` to target an absolute truncation error of 2^{-prec} .

```
void _acb_poly_zeta_series(acb_ptr res, acb_srcptr h, slong hlen, const acb_t a, int deflate, slong
                          len, slong prec)
```

```
void acb_poly_zeta_series(acb_poly_t res, const acb_poly_t f, const acb_t a, int deflate, slong n,
                          slong prec)
```

Sets *res* to the Hurwitz zeta function $\zeta(s, a)$ where s a power series and a is a constant, truncated to length n . To evaluate the usual Riemann zeta function, set $a = 1$.

If *deflate* is nonzero, evaluates $\zeta(s, a) + 1/(1 - s)$, which is well-defined as a limit when the constant term of s is 1. In particular, expanding $\zeta(s, a) + 1/(1 - s)$ with $s = 1 + x$ gives the Stieltjes constants

$$\sum_{k=0}^{n-1} \frac{(-1)^k}{k!} \gamma_k(a) x^k.$$

If $a = 1$, this implementation uses the reflection formula if the midpoint of the constant term of s is negative.

9.12.22 Other special functions

```
void _acb_poly_polylog_cpx_small(acb_ptr w, const acb_t s, const acb_t z, slong len, slong prec)
```

```
void _acb_poly_polylog_cpx_zeta(acb_ptr w, const acb_t s, const acb_t z, slong len, slong prec)
```

```
void _acb_poly_polylog_cpx(acb_ptr w, const acb_t s, const acb_t z, slong len, slong prec)
```

Sets w to the Taylor series with respect to x of the polylogarithm $\text{Li}_{s+x}(z)$, where s and z are given complex constants. The output is computed to length *len* which must be positive. Aliasing between w and s or z is not permitted.

The *small* version uses the standard power series expansion with respect to z , convergent when $|z| < 1$. The *zeta* version evaluates the polylogarithm as a sum of two Hurwitz zeta functions. The default version automatically delegates to the *small* version when z is close to zero, and the *zeta* version otherwise. For further details, see *Algorithms for polylogarithms*.

```
void _acb_poly_polylog_series(acb_ptr w, acb_srcptr s, slong slen, const acb_t z, slong len, slong
                              prec)
```

```
void acb_poly_polylog_series(acb_poly_t w, const acb_poly_t s, const acb_t z, slong len, slong
                              prec)
```

Sets w to the polylogarithm $\text{Li}_s(z)$ where s is a given power series, truncating the output to length *len*. The underscore method requires all lengths to be positive and supports aliasing between all inputs and outputs.

```
void _acb_poly_erf_series(acb_ptr res, acb_srcptr z, slong zlen, slong n, slong prec)
```

void **acb_poly_erf_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* n, *slong* prec)

Sets *res* to the error function of the power series *z*, truncated to length *n*. These methods are provided for backwards compatibility. See *acb_hypgeom_erf_series()*, *acb_hypgeom_erfc_series()*, *acb_hypgeom_erfi_series()*.

void **_acb_poly_agm1_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_poly_agm1_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* n, *slong* prec)

Sets *res* to the arithmetic-geometric mean of 1 and the power series *z*, truncated to length *n*.

See the *acb_elliptic.h* module for power series of elliptic functions. The following wrappers are available for backwards compatibility.

void **_acb_poly_elliptic_k_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_poly_elliptic_k_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* n, *slong* prec)

void **_acb_poly_elliptic_p_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, const *acb_t* tau, *slong* len, *slong* prec)

void **acb_poly_elliptic_p_series**(*acb_poly_t* res, const *acb_poly_t* z, const *acb_t* tau, *slong* n, *slong* prec)

9.12.23 Root-finding

void **_acb_poly_root_bound_fujiwara**(*mag_t* bound, *acb_srcptr* poly, *slong* len)

void **acb_poly_root_bound_fujiwara**(*mag_t* bound, *acb_poly_t* poly)

Sets *bound* to an upper bound for the magnitude of all the complex roots of *poly*. Uses Fujiwara's bound

$$2 \max \left\{ \left| \frac{a_{n-1}}{a_n} \right|, \left| \frac{a_{n-2}}{a_n} \right|^{1/2}, \dots, \left| \frac{a_1}{a_n} \right|^{1/(n-1)}, \left| \frac{a_0}{2a_n} \right|^{1/n} \right\}$$

where $a_0, \dots, a_n$ are the coefficients of *poly*.

void **_acb_poly_root_inclusion**(*acb_t* r, const *acb_t* m, *acb_srcptr* poly, *acb_srcptr* polyder, *slong* len, *slong* prec)

Given any complex number *m*, and a nonconstant polynomial *f* and its derivative *f'*, sets *r* to a complex interval centered on *m* that is guaranteed to contain at least one root of *f*. Such an interval is obtained by taking a ball of radius $|f(m)/f'(m)|n$ where *n* is the degree of *f*. Proof: assume that the distance to the nearest root exceeds $r = |f(m)/f'(m)|n$. Then

$$\left| \frac{f'(m)}{f(m)} \right| = \left| \sum_i \frac{1}{m - \zeta_i} \right| \leq \sum_i \frac{1}{|m - \zeta_i|} < \frac{n}{r} = \left| \frac{f'(m)}{f(m)} \right|$$

which is a contradiction (see [Kob2010]).

slong **_acb_poly_validate_roots**(*acb_ptr* roots, *acb_srcptr* poly, *slong* len, *slong* prec)

Given a list of approximate roots of the input polynomial, this function sets a rigorous bounding interval for each root, and determines which roots are isolated from all the other roots. It then rearranges the list of roots so that the isolated roots are at the front of the list, and returns the count of isolated roots.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the remaining output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial's roots are contained among them.

slong **_acb_poly_find_roots**(*acb_ptr* roots, *acb_srcptr* poly, *acb_srcptr* initial, *slong* len, *slong* maxiter, *slong* prec)

slong **acb_poly_find_roots**(*acb_ptr* roots, const *acb_poly_t* poly, *acb_srcptr* initial, *slong* maxiter, *slong* prec)

Attempts to compute all the roots of the given nonzero polynomial *poly* using a working precision of *prec* bits. If *n* denotes the degree of *poly*, the function writes *n* approximate roots with rigorous error bounds to the preallocated array *roots*, and returns the number of roots that are isolated.

If the return value equals the degree of the polynomial, then all roots have been found. If the return value is smaller, all the output intervals are guaranteed to contain roots, but it is possible that not all of the polynomial's roots are contained among them.

The roots are computed numerically by performing several steps with the Durand-Kerner method and terminating if the estimated accuracy of the roots approaches the working precision or if the number of steps exceeds *maxiter*, which can be set to zero in order to use a default value. Finally, the approximate roots are validated rigorously.

Initial values for the iteration can be provided as the array *initial*. If *initial* is set to *NULL*, default values are chosen using the Newton polygon technique of [Bin1996]. Suppose that the polynomial is $\sum_k a_k x^k$ and let $k_1, \dots, k_q$ denote the index coordinates of the vertices in the convex hull of $(k, \log |a_k|)$. For each $2 \leq i \leq q$, we pick $m_i = k_i - k_{i-1}$ points on the circle with radius $r_i = (|a_{k_{i-1}}|/|a_{k_i}|)^{1/m_i}$.

The polynomial is assumed to be squarefree. If there are repeated roots, the iteration is likely to find them (with low numerical accuracy), but the error bounds will not converge as the precision increases.

double **_acb_poly_find_roots_double**(*acb_ptr* roots, *acb_srcptr* poly, *acb_srcptr* initial, *slong* len, *slong* maxiter, *slong* prec)

Attempts to compute all the complex root approximations of the input polynomial, using double precision floating-point arithmetic. In particular, the radii of the entries in *roots* are set to zero.

The implementation is based on the Durand-Kerner method, with a special attention to overflow issues. The precision *prec* is used only below 53 for early termination if the estimated accuracy is below the working precision.

The function returns the last *correction*, that is the maximal step size computed at the last iteration. In particular, if there was no overflow, and up to rounding errors, the Gershgorin circle theorem ensures that the roots of the input polynomial are included in the union of the disks centered at the entries z_k of *roots* and of radius $(len - 2)|z_k|$ *correction*.

double **cd_poly_find_roots**(double *roots, const double *poly, const double *initial, *slong* len, *slong* maxiter, double reltol)

Attempts to compute all the root approximations as above. The array *roots* should be preallocated with the size of $2len$ *double*. The *poly* array is an array of size $2len$ *double*, where the entries at indices $2k$ and $2k + 1$ represent the real and the imaginary parts of the coefficient of degree k . Similarly, *initial* can be *NULL* or an array of size $2(len - 1)$ *double*, where the real and imaginary parts of the initial points are stored contiguously.

double **cd_poly_refine_roots**(double *z, const double *poly, *slong* len, double stepsize_bound)

double **cd_poly_refine_roots_with_pivot**(double *z, const double *poly, *slong* len, double stepsize_bound)

Updates the array of $len - 1$ complex numbers in *z* with one step of the Durand-Kerner method. The variant with pivot reorders the intermediate operations to reduce the risk of overflow.

int **_acb_poly_validate_real_roots**(*acb_srcptr* roots, *acb_srcptr* poly, *slong* len, *slong* prec)

int **acb_poly_validate_real_roots**(*acb_srcptr* roots, const *acb_poly_t* poly, *slong* prec)

Given a strictly real polynomial *poly* (of length *len*) and isolating intervals for all its complex roots, determines if all the real roots are separated from the non-real roots. If this function returns nonzero, every root enclosure that touches the real axis (as tested by applying *arb_contains_zero()* to the imaginary part) corresponds to a real root (its imaginary part can

be set to zero), and every other root enclosure corresponds to a non-real root (with known sign for the imaginary part).

If this function returns zero, then the signs of the imaginary parts are not known for certain, based on the accuracy of the inputs and the working precision *prec*.

9.13 arb_fmpz_poly.h – extra methods for integer polynomials

This module provides methods for FLINT polynomials with integer and rational coefficients (*fmpz_poly_t*) and (*fmpq_poly_t*) requiring use of Arb real or complex numbers.

Some methods output real or complex numbers while others use real and complex numbers internally to produce an exact result. This module also contains some useful helper functions not specifically related to real and complex numbers.

Note that methods that combine Arb *polynomials* and FLINT polynomials are found in the respective Arb polynomial modules, such as *arb_poly_set_fmpz_poly()* and *arb_poly_get_unique_fmpz_poly()*.

9.13.1 Evaluation

```
void _arb_fmpz_poly_evaluate_arb_horner(arb_t res, const fmpz *poly, slong len, const arb_t x,  
                                        slong prec)
```

```
void arb_fmpz_poly_evaluate_arb_horner(arb_t res, const fmpz_poly_t poly, const arb_t x, slong  
                                       prec)
```

```
void _arb_fmpz_poly_evaluate_arb_rectangular(arb_t res, const fmpz *poly, slong len, const  
                                             arb_t x, slong prec)
```

```
void arb_fmpz_poly_evaluate_arb_rectangular(arb_t res, const fmpz_poly_t poly, const arb_t x,  
                                             slong prec)
```

```
void _arb_fmpz_poly_evaluate_arb(arb_t res, const fmpz *poly, slong len, const arb_t x, slong prec)
```

```
void arb_fmpz_poly_evaluate_arb(arb_t res, const fmpz_poly_t poly, const arb_t x, slong prec)
```

```
void _arb_fmpz_poly_evaluate_acb_horner(acb_t res, const fmpz *poly, slong len, const acb_t x,  
                                        slong prec)
```

```
void arb_fmpz_poly_evaluate_acb_horner(acb_t res, const fmpz_poly_t poly, const acb_t x, slong  
                                       prec)
```

```
void _arb_fmpz_poly_evaluate_acb_rectangular(acb_t res, const fmpz *poly, slong len, const  
                                             acb_t x, slong prec)
```

```
void arb_fmpz_poly_evaluate_acb_rectangular(acb_t res, const fmpz_poly_t poly, const acb_t x,  
                                             slong prec)
```

```
void _arb_fmpz_poly_evaluate_acb(acb_t res, const fmpz *poly, slong len, const acb_t x, slong prec)
```

```
void arb_fmpz_poly_evaluate_acb(acb_t res, const fmpz_poly_t poly, const acb_t x, slong prec)
```

Evaluates *poly* (given by a polynomial object or an array with *len* coefficients) at the given real or complex number, respectively using Horner's rule, rectangular splitting, or a default algorithm choice.

9.13.2 Utility methods

`ulong arb_fmpz_poly_deflation(const fmpz_poly_t poly)`

Finds the maximal exponent by which *poly* can be deflated.

`void arb_fmpz_poly_deflate(fmpz_poly_t res, const fmpz_poly_t poly, ulong deflation)`

Sets *res* to a copy of *poly* deflated by the exponent *deflation*.

9.13.3 Polynomial roots

`void arb_fmpz_poly_complex_roots(acb_ptr roots, const fmpz_poly_t poly, int flags, slong prec)`

Writes to *roots* all the real and complex roots of the polynomial *poly*, computed to at least *prec* accurate bits. The root enclosures are guaranteed to be disjoint, so that all roots are isolated.

The real roots are written first in ascending order (with the imaginary parts set exactly to zero). The following nonreal roots are written in arbitrary order, but with conjugate pairs grouped together (the root in the upper plane leading the root in the lower plane).

The input polynomial *must* be squarefree. For a general polynomial, compute the squarefree part $f/\gcd(f, f')$ or do a full squarefree factorization to obtain the multiplicities of the roots:

```
fmpz_poly_factor_t fac;
fmpz_poly_factor_init(fac);
fmpz_poly_factor_squarefree(fac, poly);

for (i = 0; i < fac->num; i++)
{
    deg = fmpz_poly_degree(fac->p + i);
    flint_printf("%wd roots of multiplicity %wd\n", deg, fac->exp[i]);
    roots = _acb_vec_init(deg);
    arb_fmpz_poly_complex_roots(roots, fac->p + i, 0, prec);
    _acb_vec_clear(roots, deg);
}

fmpz_poly_factor_clear(fac);
```

All roots are refined to a relative accuracy of at least *prec* bits. The output values will generally have higher actual precision, depending on the precision needed for isolation and the precision used internally by the algorithm.

This implementation should be adequate for general use, but it is not currently competitive with state-of-the-art isolation methods for finding real roots alone.

The following *flags* are supported:

- `ARB_FMPZ_POLY_ROOTS_VERBOSE`

9.13.4 Special polynomials

Note: see also the methods available in FLINT (e.g. for cyclotomic polynomials).

`void arb_fmpz_poly_gauss_period_minpoly(fmpz_poly_t res, ulong q, ulong n)`

Sets *res* to the minimal polynomial of the Gaussian periods $\sum_{a \in H} \zeta^a$ where $\zeta = \exp(2\pi i/q)$ and *H* are the cosets of the subgroups of order $d = (q-1)/n$ of $(\mathbb{Z}/q\mathbb{Z})^\times$. The resulting polynomial has degree *n*. When $d = 1$, the result is the cyclotomic polynomial Φ_q .

The implementation assumes that *q* is prime, and that *n* is a divisor of $q-1$ such that *n* is coprime with *d*. If any condition is not met, *res* is set to the zero polynomial.

This method provides a fast (in practice) way to construct finite field extensions of prescribed degree. If q satisfies the conditions stated above and $(q - 1)/f$ additionally is coprime with n , where f is the multiplicative order of p mod q , then the Gaussian period minimal polynomial is irreducible over $\text{GF}(p)$ [CP2005].

9.14 acb_dft.h – Discrete Fourier transform

Warning: the interfaces in this module are experimental and may change without notice.

All functions support aliasing.

Let G be a finite abelian group, and χ a character of G . For any map $f : G \rightarrow \mathbb{C}$, the discrete fourier transform $\hat{f} : \hat{G} \rightarrow \mathbb{C}$ is defined by

$$\hat{f}(\chi) = \sum_{x \in G} \overline{\chi(x)} f(x)$$

Note that by the inversion formula

$$\widehat{\hat{f}}(\chi) = \#G \times f(\chi^{-1})$$

it is straightforward to recover f from its DFT $\hat{f}$.

9.14.1 Main DFT functions

If $G = \mathbb{Z}/n\mathbb{Z}$, we compute the DFT according to the usual convention

$$w_x = \sum_{y \bmod n} v_y e^{-\frac{2i\pi}{n}xy}$$

void **acb_dft**(*acb_ptr* w, *acb_srcptr* v, *slong* len, *slong* prec)

Set w to the DFT of v of length len , using an automatic choice of algorithm.

void **acb_dft_inverse**(*acb_ptr* w, *acb_srcptr* v, *slong* len, *slong* prec)

Compute the inverse DFT of v into w .

If several computations are to be done on the same group, the FFT scheme should be reused.

type **acb_dft_pre_struct**

type **acb_dft_pre_t**

Stores a fast DFT scheme on $\mathbb{Z}/n\mathbb{Z}$ as a recursive decomposition into simpler DFT with some tables of roots of unity.

An *acb_dft_pre_t* is defined as an array of *acb_dft_pre_struct* of length 1, permitting it to be passed by reference.

void **acb_dft_precomp_init**(*acb_dft_pre_t* pre, *slong* len, *slong* prec)

Initializes the fast DFT scheme of length len , using an automatic choice of algorithms depending on the factorization of len .

The length len is stored as *pre->n*.

void **acb_dft_precomp_clear**(*acb_dft_pre_t* pre)

Clears *pre*.

void **acb_dft_precomp**(*acb_ptr* w, *acb_srcptr* v, const *acb_dft_pre_t* pre, *slong* prec)

Computes the DFT of the sequence v into w by applying the precomputed scheme *pre*. Both v and w must have length *pre->n*.

void **acb_dft_inverse_precomp**(*acb_ptr* w, *acb_srcptr* v, const *acb_dft_pre_t* pre, *slong* prec)

Compute the inverse DFT of v into w .

9.14.2 DFT on products

A finite abelian group is isomorphic to a product of cyclic components

$$G = \bigoplus_{i=1}^r \mathbb{Z}/n_i\mathbb{Z}$$

Characters are product of component characters and the DFT reads

$$\hat{f}(x_1, \dots, x_r) = \sum_{y_1, \dots, y_r} f(y_1, \dots, y_r) e^{-2i\pi \sum \frac{x_i y_i}{n_i}}$$

We assume that f is given by a vector of length $\prod n_i$ corresponding to a lexicographic ordering of the values $y_1, \dots, y_r$, and the computation returns the same indexing for values of $\hat{f}$.

void **acb_dirichlet_dft_prod**(*acb_ptr* w, *acb_srcptr* v, *slong* *cyc, *slong* num, *slong* prec)

Computes the DFT on the group product of *num* cyclic components of sizes *cyc*. Assume the entries of *v* are indexed according to lexicographic ordering of the cyclic components.

type **acb_dft_prod_struct**

type **acb_dft_prod_t**

Stores a fast DFT scheme on a product of cyclic groups.

An *acb_dft_prod_t* is defined as an array of *acb_dft_prod_struct* of length 1, permitting it to be passed by reference.

void **acb_dft_prod_init**(*acb_dft_prod_t* t, *slong* *cyc, *slong* num, *slong* prec)

Stores in *t* a DFT scheme for the product of *num* cyclic components whose sizes are given in the array *cyc*.

void **acb_dft_prod_clear**(*acb_dft_prod_t* t)

Clears *t*.

void **acb_dirichlet_dft_prod_precomp**(*acb_ptr* w, *acb_srcptr* v, const *acb_dft_prod_t* prod, *slong* prec)

Sets *w* to the DFT of *v*. Assume the entries are lexicographically ordered according to the product of cyclic groups initialized in *t*.

9.14.3 Convolution

For functions f and g on G we consider the convolution

$$(f \star g)(x) = \sum_{y \in G} f(x - y)g(y)$$

void **acb_dft_convolve_naive**(*acb_ptr* w, *acb_srcptr* f, *acb_srcptr* g, *slong* len, *slong* prec)

void **acb_dft_convolve_rad2**(*acb_ptr* w, *acb_srcptr* f, *acb_srcptr* g, *slong* len, *slong* prec)

void **acb_dft_convolve**(*acb_ptr* w, *acb_srcptr* f, *acb_srcptr* g, *slong* len, *slong* prec)

Sets *w* to the convolution of *f* and *g* of length *len*.

The *naive* version simply uses the definition.

The *rad2* version embeds the sequence into a power of 2 length and uses the formula

$$\widehat{f \star g}(\chi) = \hat{f}(\chi)\hat{g}(\chi)$$

to compute it using three radix 2 FFT.

The default version uses radix 2 FFT unless *len* is a product of small primes where a non padded FFT is faster.

9.14.4 FFT algorithms

Fast Fourier transform techniques allow to compute efficiently all values $\hat{f}(\chi)$ by reusing common computations.

Specifically, if $H \triangleleft G$ is a subgroup of size M and index $[G : H] = m$, then writing $f_x(h) = f(xh)$ the translate of f by representatives x of G/H , one has a decomposition

$$\hat{f}(\chi) = \sum_{x \in G/H} \overline{\chi(x)} \hat{f}_x(\chi_H)$$

so that the DFT on G can be computed using m DFT on H (of appropriate translates of f), then M DFT on G/H , one for each restriction χ_H .

This decomposition can be done recursively.

Naive algorithm

`void acb_dft_naive(acb_ptr w, acb_srcptr v, slong n, slong prec)`

Computes the DFT of v into w , where v and w have size n , using the naive $O(n^2)$ algorithm.

`type acb_dft_naive_struct`

`type acb_dft_naive_t`

`void acb_dft_naive_init(acb_dft_naive_t t, slong len, slong prec)`

`void acb_dft_naive_clear(acb_dft_naive_t t)`

Stores a table of roots of unity in t . The length len is stored as $t->n$.

`void acb_dft_naive_precomp(acb_ptr w, acb_srcptr v, const acb_dft_naive_t t, slong prec)`

Sets w to the DFT of v of size $t->n$, using the naive algorithm data t .

CRT decomposition

`void acb_dft_crt(acb_ptr w, acb_srcptr v, slong n, slong prec)`

Computes the DFT of v into w , where v and w have size len , using CRT to express $\mathbb{Z}/n\mathbb{Z}$ as a product of cyclic groups.

`type acb_dft_crt_struct`

`type acb_dft_crt_t`

`void acb_dft_crt_init(acb_dft_crt_t t, slong len, slong prec)`

`void acb_dft_crt_clear(acb_dft_crt_t t)`

Initialize a CRT decomposition of $\mathbb{Z}/n\mathbb{Z}$ as a direct product of cyclic groups. The length len is stored as $t->n$.

`void acb_dft_crt_precomp(acb_ptr w, acb_srcptr v, const acb_dft_crt_t t, slong prec)`

Sets w to the DFT of v of size $t->n$, using the CRT decomposition scheme t .

Cooley-Tukey decomposition

void **acb_dft_cyc**(*acb_ptr* w, *acb_srcptr* v, *slong* n, *slong* prec)
 Computes the DFT of v into w , where v and w have size n , using each prime factor of m of n to decompose with the subgroup $H = m\mathbb{Z}/n\mathbb{Z}$.

type **acb_dft_cyc_struct**

type **acb_dft_cyc_t**

void **acb_dft_cyc_init**(*acb_dft_cyc_t* t, *slong* len, *slong* prec)
 Initialize a decomposition of $\mathbb{Z}/n\mathbb{Z}$ into cyclic subgroups. The length len is stored as $t->n$.

void **acb_dft_cyc_clear**(*acb_dft_cyc_t* t)
 Initialize a decomposition of $\mathbb{Z}/n\mathbb{Z}$ into cyclic subgroups. The length len is stored as $t->n$.

void **acb_dft_cyc_precomp**(*acb_ptr* w, *acb_srcptr* v, const *acb_dft_cyc_t* t, *slong* prec)
 Sets w to the DFT of v of size $t->n$, using the cyclic decomposition scheme t .

Radix 2 decomposition

void **acb_dft_rad2**(*acb_ptr* w, *acb_srcptr* v, int e, *slong* prec)
 Computes the DFT of v into w , where v and w have size 2^e , using a radix 2 FFT.

void **acb_dft_inverse_rad2**(*acb_ptr* w, *acb_srcptr* v, int e, *slong* prec)
 Computes the inverse DFT of v into w , where v and w have size 2^e , using a radix 2 FFT.

type **acb_dft_rad2_struct**

type **acb_dft_rad2_t**

void **acb_dft_rad2_init**(*acb_dft_rad2_t* t, int e, *slong* prec)
 Initialize and clear a radix 2 FFT of size 2^e , stored as $t->n$.

void **acb_dft_rad2_clear**(*acb_dft_rad2_t* t)
 Initialize and clear a radix 2 FFT of size 2^e , stored as $t->n$.

void **acb_dft_rad2_precomp**(*acb_ptr* w, *acb_srcptr* v, const *acb_dft_rad2_t* t, *slong* prec)
 Sets w to the DFT of v of size $t->n$, using the precomputed radix 2 scheme t .

Bluestein transform

void **acb_dft_bluestein**(*acb_ptr* w, *acb_srcptr* v, *slong* n, *slong* prec)
 Computes the DFT of v into w , where v and w have size n , by conversion to a radix 2 one using Bluestein's convolution trick.

type **acb_dft_bluestein_struct**

type **acb_dft_bluestein_t**
 Stores a Bluestein scheme for some length n : that is a *acb_dft_rad2_t* of size $2^e \geq 2n - 1$ and a size n array of convolution factors.

void **acb_dft_bluestein_init**(*acb_dft_bluestein_t* t, *slong* len, *slong* prec)
 Initialize and clear a Bluestein scheme to compute DFT of size len .

void **acb_dft_bluestein_clear**(*acb_dft_bluestein_t* t)
 Initialize and clear a Bluestein scheme to compute DFT of size len .

void **acb_dft_bluestein_precomp**(*acb_ptr* w, *acb_srcptr* v, const *acb_dft_bluestein_t* t, *slong* prec)
 Sets w to the DFT of v of size $t->n$, using the precomputed Bluestein scheme t .

9.15 `arb_mat.h` – matrices over the real numbers

An `arb_mat_t` represents a dense matrix over the real numbers, implemented as an array of entries of type `arb_struct`. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

Note: Methods prefixed with `arb_mat_approx` treat all input entries as floating-point numbers (ignoring the radii of the balls) and compute floating-point output (balls with zero radius) representing approximate solutions *without error bounds*. All other methods compute rigorous error bounds. The `approx` methods are typically useful for computing initial values or preconditioners for rigorous solvers. Some users may also find `approx` methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

9.15.1 Types, macros and constants

type `arb_mat_struct`

type `arb_mat_t`

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An `arb_mat_t` is defined as an array of length one of type `arb_mat_struct`, permitting an `arb_mat_t` to be passed by reference.

`arb_mat_entry(mat, i, j)`

Macro giving a pointer to the entry at row *i* and column *j*.

`arb_mat_nrows(mat)`

Returns the number of rows of the matrix.

`arb_mat_ncols(mat)`

Returns the number of columns of the matrix.

9.15.2 Memory management

void `arb_mat_init(arb_mat_t mat, slong r, slong c)`

Initializes the matrix, setting it to the zero matrix with *r* rows and *c* columns.

void `arb_mat_clear(arb_mat_t mat)`

Clears the matrix, deallocating all entries.

slong `arb_mat_allocated_bytes(const arb_mat_t x)`

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(arb_mat_struct)` to get the size of the object as a whole.

void `arb_mat_window_init(arb_mat_t window, const arb_mat_t mat, slong r1, slong c1, slong r2, slong c2)`

Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive).

void `arb_mat_window_clear(arb_mat_t window)`

Frees the window matrix.

9.15.3 Conversions

void **arb_mat_set**(*arb_mat_t* dest, const *arb_mat_t* src)

void **arb_mat_set_fmpz_mat**(*arb_mat_t* dest, const *fmpz_mat_t* src)

void **arb_mat_set_round_fmpz_mat**(*arb_mat_t* dest, const *fmpz_mat_t* src, *slong* prec)

void **arb_mat_set_fmpq_mat**(*arb_mat_t* dest, const *fmpq_mat_t* src, *slong* prec)

Sets *dest* to *src*. The operands must have identical dimensions.

9.15.4 Random generation

void **arb_mat_randtest**(*arb_mat_t* mat, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Sets *mat* to a random matrix with up to *prec* bits of precision and with exponents of width up to *mag_bits*.

void **arb_mat_randtest_cho**(*arb_mat_t* mat, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Sets *mat* to a random lower-triangular matrix with precise entries and positive diagonal entries. Requires that *mat* is square.

void **arb_mat_randtest_spd**(*arb_mat_t* mat, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Sets *mat* to a random symmetric positive definite matrix, obtained as a product LL^T where L is a random Cholesky matrix. Requires that *mat* is square.

9.15.5 Input and output

void **arb_mat_printd**(const *arb_mat_t* mat, *slong* digits)

Prints each entry in the matrix with the specified number of decimal digits.

void **arb_mat_fprintd**(FILE *file, const *arb_mat_t* mat, *slong* digits)

Prints each entry in the matrix with the specified number of decimal digits to the stream *file*.

9.15.6 Comparisons

Predicate methods return 1 if the property certainly holds and 0 otherwise.

int **arb_mat_equal**(const *arb_mat_t* mat1, const *arb_mat_t* mat2)

Returns whether the matrices have the same dimensions and identical intervals as entries.

int **arb_mat_overlaps**(const *arb_mat_t* mat1, const *arb_mat_t* mat2)

Returns whether the matrices have the same dimensions and each entry in *mat1* overlaps with the corresponding entry in *mat2*.

int **arb_mat_contains**(const *arb_mat_t* mat1, const *arb_mat_t* mat2)

int **arb_mat_contains_fmpz_mat**(const *arb_mat_t* mat1, const *fmpz_mat_t* mat2)

int **arb_mat_contains_fmpq_mat**(const *arb_mat_t* mat1, const *fmpq_mat_t* mat2)

Returns whether the matrices have the same dimensions and each entry in *mat2* is contained in the corresponding entry in *mat1*.

int **arb_mat_eq**(const *arb_mat_t* mat1, const *arb_mat_t* mat2)

Returns whether *mat1* and *mat2* certainly represent the same matrix.

int **arb_mat_ne**(const *arb_mat_t* mat1, const *arb_mat_t* mat2)

Returns whether *mat1* and *mat2* certainly do not represent the same matrix.

int **arb_mat_is_empty**(const *arb_mat_t* mat)

Returns whether the number of rows or the number of columns in *mat* is zero.

int **arb_mat_is_square**(const *arb_mat_t* mat)

Returns whether the number of rows is equal to the number of columns in *mat*.

int **arb_mat_is_exact**(const *arb_mat_t* mat)

Returns whether all entries in *mat* have zero radius.

int **arb_mat_is_zero**(const *arb_mat_t* mat)

Returns whether all entries in *mat* are exactly zero.

int **arb_mat_is_finite**(const *arb_mat_t* mat)

Returns whether all entries in *mat* are finite.

int **arb_mat_is_triu**(const *arb_mat_t* mat)

Returns whether *mat* is upper triangular; that is, all entries below the main diagonal are exactly zero.

int **arb_mat_is_tril**(const *arb_mat_t* mat)

Returns whether *mat* is lower triangular; that is, all entries above the main diagonal are exactly zero.

int **arb_mat_is_diag**(const *arb_mat_t* mat)

Returns whether *mat* is a diagonal matrix; that is, all entries off the main diagonal are exactly zero.

9.15.7 Special matrices

void **arb_mat_zero**(*arb_mat_t* mat)

Sets all entries in *mat* to zero.

void **arb_mat_one**(*arb_mat_t* mat)

Sets the entries on the main diagonal to ones, and all other entries to zero.

void **arb_mat_ones**(*arb_mat_t* mat)

Sets all entries in the matrix to ones.

void **arb_mat_indeterminate**(*arb_mat_t* mat)

Sets all entries in the matrix to indeterminate (NaN).

void **arb_mat_hilbert**(*arb_mat_t* mat, *slong* prec)

Sets *mat* to the Hilbert matrix, which has entries $A_{j,k} = 1/(j + k + 1)$.

void **arb_mat_pascal**(*arb_mat_t* mat, int triangular, *slong* prec)

Sets *mat* to a Pascal matrix, whose entries are binomial coefficients. If *triangular* is 0, constructs a full symmetric matrix with the rows of Pascal's triangle as successive antidiagonals. If *triangular* is 1, constructs the upper triangular matrix with the rows of Pascal's triangle as columns, and if *triangular* is -1, constructs the lower triangular matrix with the rows of Pascal's triangle as rows.

The entries are computed using recurrence relations. When the dimensions get large, some precision loss is possible; in that case, the user may wish to create the matrix at slightly higher precision and then round it to the final precision.

void **arb_mat_stirling**(*arb_mat_t* mat, int kind, *slong* prec)

Sets *mat* to a Stirling matrix, whose entries are Stirling numbers. If *kind* is 0, the entries are set to the unsigned Stirling numbers of the first kind. If *kind* is 1, the entries are set to the signed Stirling numbers of the first kind. If *kind* is 2, the entries are set to the Stirling numbers of the second kind.

The entries are computed using recurrence relations. When the dimensions get large, some precision loss is possible; in that case, the user may wish to create the matrix at slightly higher precision and then round it to the final precision.

void **arb_mat_dct**(*arb_mat_t* mat, int type, *slong* prec)

Sets *mat* to the DCT (discrete cosine transform) matrix of order n where n is the smallest dimension of *mat* (if *mat* is not square, the matrix is extended periodically along the larger dimension). There are many different conventions for defining DCT matrices; here, we use the normalized “DCT-II” transform matrix

$$A_{j,k} = \sqrt{\frac{2}{n}} \cos\left(\frac{\pi j}{n} \left(k + \frac{1}{2}\right)\right)$$

which satisfies $A^{-1} = A^T$. The *type* parameter is currently ignored and should be set to 0. In the future, it might be used to select a different convention.

9.15.8 Transpose

void **arb_mat_transpose**(*arb_mat_t* dest, const *arb_mat_t* src)

Sets *dest* to the exact transpose *src*. The operands must have compatible dimensions. Aliasing is allowed for square matrices.

9.15.9 Norms

void **arb_mat_bound_inf_norm**(*mag_t* b, const *arb_mat_t* A)

Sets *b* to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of *A*.

void **arb_mat_frobenius_norm**(*arb_t* res, const *arb_mat_t* A, *slong* prec)

Sets *res* to the Frobenius norm (i.e. the square root of the sum of squares of entries) of *A*.

void **arb_mat_bound_frobenius_norm**(*mag_t* res, const *arb_mat_t* A)

Sets *res* to an upper bound for the Frobenius norm of *A*.

9.15.10 Arithmetic

void **arb_mat_neg**(*arb_mat_t* dest, const *arb_mat_t* src)

Sets *dest* to the exact negation of *src*. The operands must have the same dimensions.

void **arb_mat_add**(*arb_mat_t* res, const *arb_mat_t* mat1, const *arb_mat_t* mat2, *slong* prec)

Sets *res* to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **arb_mat_sub**(*arb_mat_t* res, const *arb_mat_t* mat1, const *arb_mat_t* mat2, *slong* prec)

Sets *res* to the difference of *mat1* and *mat2*. The operands must have the same dimensions.

void **arb_mat_mul_classical**(*arb_mat_t* C, const *arb_mat_t* A, const *arb_mat_t* B, *slong* prec)

void **arb_mat_mul_threaded**(*arb_mat_t* C, const *arb_mat_t* A, const *arb_mat_t* B, *slong* prec)

void **arb_mat_mul_block**(*arb_mat_t* C, const *arb_mat_t* A, const *arb_mat_t* B, *slong* prec)

void **arb_mat_mul**(*arb_mat_t* res, const *arb_mat_t* mat1, const *arb_mat_t* mat2, *slong* prec)

Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

The *classical* version performs matrix multiplication in the trivial way.

The *block* version decomposes the input matrices into one or several blocks of uniformly scaled matrices and multiplies large blocks via *fmpz_mat_mul*. It also invokes *_arb_mat_addmul_rad_mag_fast()* for the radius matrix multiplications.

The *threaded* version performs classical multiplication but splits the computation over the number of threads returned by `flint_get_num_threads()`.

The default version chooses an algorithm automatically.

void **arb_mat_mul_entrywise**(*arb_mat_t* C, const *arb_mat_t* A, const *arb_mat_t* B, *slong* prec)

Sets *C* to the entrywise product of *A* and *B*. The operands must have the same dimensions.

void **arb_mat_sqr_classical**(*arb_mat_t* B, const *arb_mat_t* A, *slong* prec)

void **arb_mat_sqr**(*arb_mat_t* res, const *arb_mat_t* mat, *slong* prec)

Sets *res* to the matrix square of *mat*. The operands must both be square with the same dimensions.

void **arb_mat_pow_ui**(*arb_mat_t* res, const *arb_mat_t* mat, *ulong* exp, *slong* prec)

Sets *res* to *mat* raised to the power *exp*. Requires that *mat* is a square matrix.

void **_arb_mat_addmul_rad_mag_fast**(*arb_mat_t* C, mag_srcptr A, mag_srcptr B, *slong* ar, *slong* ac, *slong* bc)

Helper function for matrix multiplication. Adds to the radii of *C* the matrix product of the matrices represented by *A* and *B*, where *A* is a linear array of coefficients in row-major order and *B* is a linear array of coefficients in column-major order. This function assumes that all exponents are small and is unsafe for general use.

void **arb_mat_approx_mul**(*arb_mat_t* res, const *arb_mat_t* mat1, const *arb_mat_t* mat2, *slong* prec)

Approximate matrix multiplication. The input radii are ignored and the output matrix is set to an approximate floating-point result. The radii in the output matrix will *not* necessarily be zeroed.

9.15.11 Scalar arithmetic

void **arb_mat_scalar_mul_2exp_si**(*arb_mat_t* B, const *arb_mat_t* A, *slong* c)

Sets *B* to *A* multiplied by 2^c .

void **arb_mat_scalar_addmul_si**(*arb_mat_t* B, const *arb_mat_t* A, *slong* c, *slong* prec)

void **arb_mat_scalar_addmul_fmpz**(*arb_mat_t* B, const *arb_mat_t* A, const *fmpz_t* c, *slong* prec)

void **arb_mat_scalar_addmul_arb**(*arb_mat_t* B, const *arb_mat_t* A, const *arb_t* c, *slong* prec)

Sets *B* to $B + A \times c$.

void **arb_mat_scalar_mul_si**(*arb_mat_t* B, const *arb_mat_t* A, *slong* c, *slong* prec)

void **arb_mat_scalar_mul_fmpz**(*arb_mat_t* B, const *arb_mat_t* A, const *fmpz_t* c, *slong* prec)

void **arb_mat_scalar_mul_arb**(*arb_mat_t* B, const *arb_mat_t* A, const *arb_t* c, *slong* prec)

Sets *B* to $A \times c$.

void **arb_mat_scalar_div_si**(*arb_mat_t* B, const *arb_mat_t* A, *slong* c, *slong* prec)

void **arb_mat_scalar_div_fmpz**(*arb_mat_t* B, const *arb_mat_t* A, const *fmpz_t* c, *slong* prec)

void **arb_mat_scalar_div_arb**(*arb_mat_t* B, const *arb_mat_t* A, const *arb_t* c, *slong* prec)

Sets *B* to A/c .

9.15.12 Vector arithmetic

```
void _arb_mat_vector_mul_row(arb_ptr res, arb_srcptr v, const arb_mat_t A, slong prec)
```

```
void _arb_mat_vector_mul_col(arb_ptr res, const arb_mat_t A, arb_srcptr v, slong prec)
```

```
void arb_mat_vector_mul_row(arb_ptr res, arb_srcptr v, const arb_mat_t A, slong prec)
```

```
void arb_mat_vector_mul_col(arb_ptr res, const arb_mat_t A, arb_srcptr v, slong prec)
```

Sets *res* to the product vA , (resp. Av), where *res* and *v* are seen as row (resp. column) vectors. The lengths of the vectors must match the dimensions of *A*.

The underscore methods do not allow aliasing between *res* and *v*.

9.15.13 Gaussian elimination and solving

```
int arb_mat_lu_classical(slong *perm, arb_mat_t LU, const arb_mat_t A, slong prec)
```

```
int arb_mat_lu_recursive(slong *perm, arb_mat_t LU, const arb_mat_t A, slong prec)
```

```
int arb_mat_lu(slong *perm, arb_mat_t LU, const arb_mat_t A, slong prec)
```

Given an $n \times n$ matrix *A*, computes an LU decomposition $PLU = A$ using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry *i* in the permutation vector *perm* is set to the row index in the input matrix corresponding to row *i* in the output matrix.

The algorithm succeeds and returns nonzero if it can find *n* invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in *P* and *LU* undefined, if it cannot find *n* invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

The *classical* version uses Gaussian elimination directly while the *recursive* version performs the computation in a block recursive way to benefit from fast matrix multiplication. The default version chooses an algorithm automatically.

```
void arb_mat_solve_tril_classical(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit, slong prec)
```

```
void arb_mat_solve_tril_recursive(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit, slong prec)
```

```
void arb_mat_solve_tril(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit, slong prec)
```

```
void arb_mat_solve_triu_classical(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
```

```
void arb_mat_solve_triu_recursive(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
```

```
void arb_mat_solve_triu(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit, slong prec)
```

Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. If *unit* is set, the main diagonal of *L* or *U* is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

```
void arb_mat_solve_lu_precomp(arb_mat_t X, const slong *perm, const arb_mat_t LU, const
                             arb_mat_t B, slong prec)
```

Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU .

```
int arb_mat_solve(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
```

```
int arb_mat_solve_lu(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
```

```
int arb_mat_solve_precond(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
```

Solves $AX = B$ where A is a nonsingular $n \times n$ matrix and X and B are $n \times m$ matrices.

If $m > 0$ and A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that A is invertible and that the exact solution matrix is contained in the output.

Three algorithms are provided:

- The *lu* version performs LU decomposition directly in ball arithmetic. This is fast, but the bounds typically blow up exponentially with n , even if the system is well-conditioned. This algorithm is usually the best choice at very high precision.
- The *precond* version computes an approximate inverse to precondition the system [HS1967]. This is usually several times slower than direct LU decomposition, but the bounds do not blow up with n if the system is well-conditioned. This algorithm is usually the best choice for large systems at low to moderate precision.
- The default version selects between *lu* and *precomp* automatically.

The automatic choice should be reasonable most of the time, but users may benefit from trying either *lu* or *precond* in specific applications. For example, the *lu* solver often performs better for ill-conditioned systems where use of very high precision is unavoidable.

```
int arb_mat_solve_preapprox(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, const
                             arb_mat_t R, const arb_mat_t T, slong prec)
```

Solves $AX = B$ where A is a nonsingular $n \times n$ matrix and X and B are $n \times m$ matrices, given an approximation R of the matrix inverse of A , and given the approximation T of the solution X .

If $m > 0$ and A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient, or that R is not a close enough approximation of the inverse of A), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that A is invertible and that the exact solution matrix is contained in the output.

```
int arb_mat_inv(arb_mat_t X, const arb_mat_t A, slong prec)
```

Sets $X = A^{-1}$ where A is a square matrix, computed by solving the system $AX = I$.

If A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

```
void arb_mat_det_lu(arb_t det, const arb_mat_t A, slong prec)
```

```
void arb_mat_det_precond(arb_t det, const arb_mat_t A, slong prec)
```

```
void arb_mat_det(arb_t det, const arb_mat_t A, slong prec)
```

Sets *det* to the determinant of the matrix A .

The *lu* version uses Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard's inequality.

The *precond* version computes an approximate LU factorization of A and multiplies by the inverse L and U matrices as preconditioners to obtain a matrix close to the identity matrix [Rum2010].

An enclosure for this determinant is computed using Gershgorin circles. This is about four times slower than direct Gaussian elimination, but much more numerically stable.

The default version automatically selects between the *lu* and *precond* versions and additionally handles small or triangular matrices by direct formulas.

```
void arb_mat_approx_solve_triu(arb_mat_t X, const arb_mat_t U, const arb_mat_t B, int unit,
                              slong prec)
```

```
void arb_mat_approx_solve_tril(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, int unit,
                              slong prec)
```

```
int arb_mat_approx_lu(slong *P, arb_mat_t LU, const arb_mat_t A, slong prec)
```

```
void arb_mat_approx_solve_lu_precomp(arb_mat_t X, const slong *perm, const arb_mat_t A,
                                     const arb_mat_t B, slong prec)
```

```
int arb_mat_approx_solve(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
```

```
int arb_mat_approx_inv(arb_mat_t X, const arb_mat_t A, slong prec)
```

These methods perform approximate solving *without any error control*. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

Approximate solutions are useful for computing preconditioning matrices for certified solutions. Some users may also find these methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

9.15.14 Cholesky decomposition and solving

```
int _arb_mat_cholesky_banachiewicz(arb_mat_t A, slong prec)
```

```
int arb_mat_cho(arb_mat_t L, const arb_mat_t A, slong prec)
```

Computes the Cholesky decomposition of A , returning nonzero iff the symmetric matrix defined by the lower triangular part of A is certainly positive definite.

If a nonzero value is returned, then L is set to the lower triangular matrix such that $A = L * L^T$.

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore method computes L from A in-place, leaving the strict upper triangular region undefined.

```
void arb_mat_solve_cho_precomp(arb_mat_t X, const arb_mat_t L, const arb_mat_t B, slong
                              prec)
```

Solves $AX = B$ given the precomputed Cholesky decomposition $A = LL^T$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with L .

```
int arb_mat_spd_solve(arb_mat_t X, const arb_mat_t A, const arb_mat_t B, slong prec)
```

Solves $AX = B$ where A is a symmetric positive definite matrix and X and B are $n \times m$ matrices, using Cholesky decomposition.

If $m > 0$ and A cannot be factored using Cholesky decomposition (indicating either that A is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of A is invertible and that the exact solution matrix is contained in the output.

void **arb_mat_inv_cho_precomp**(*arb_mat_t* X, const *arb_mat_t* L, *slong* prec)

Sets $X = A^{-1}$ where A is a symmetric positive definite matrix whose Cholesky decomposition L has been computed with `arb_mat_cho()`. The inverse is calculated using the method of [Kri2013] which is more efficient than solving $AX = I$ with `arb_mat_solve_cho_precomp()`.

int **arb_mat_spd_inv**(*arb_mat_t* X, const *arb_mat_t* A, *slong* prec)

Sets $X = A^{-1}$ where A is a symmetric positive definite matrix. It is calculated using the method of [Kri2013] which computes fewer intermediate results than solving $AX = I$ with `arb_mat_spd_solve()`.

If A cannot be factored using Cholesky decomposition (indicating either that A is not symmetric positive definite or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the symmetric matrix defined through the lower triangular part of A is invertible and that the exact inverse is contained in the output.

int **_arb_mat_ldl_inplace**(*arb_mat_t* A, *slong* prec)

int **_arb_mat_ldl_golub_and_van_loan**(*arb_mat_t* A, *slong* prec)

int **arb_mat_ldl**(*arb_mat_t* res, const *arb_mat_t* A, *slong* prec)

Computes the LDL^T decomposition of A , returning nonzero iff the symmetric matrix defined by the lower triangular part of A is certainly positive definite.

If a nonzero value is returned, then *res* is set to a lower triangular matrix that encodes the $L * D * L^T$ decomposition of A . In particular, L is a lower triangular matrix with ones on its diagonal and whose strictly lower triangular region is the same as that of *res*. D is a diagonal matrix with the same diagonal as that of *res*.

If zero is returned, then either the matrix is not symmetric positive definite, the input matrix was computed to insufficient precision, or the decomposition was attempted at insufficient precision.

The underscore methods compute *res* from A in-place, leaving the strict upper triangular region undefined. The default method uses algorithm 4.1.2 from [GVL1996].

void **arb_mat_solve_ldl_precomp**(*arb_mat_t* X, const *arb_mat_t* L, const *arb_mat_t* B, *slong* prec)

Solves $AX = B$ given the precomputed $A = LDL^T$ decomposition encoded by L . The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with L .

void **arb_mat_inv_ldl_precomp**(*arb_mat_t* X, const *arb_mat_t* L, *slong* prec)

Sets $X = A^{-1}$ where A is a symmetric positive definite matrix whose LDL^T decomposition encoded by L has been computed with `arb_mat_ldl()`. The inverse is calculated using the method of [Kri2013] which is more efficient than solving $AX = I$ with `arb_mat_solve_ldl_precomp()`.

9.15.15 Characteristic polynomial and companion matrix

void **_arb_mat_charpoly**(*arb_ptr* poly, const *arb_mat_t* mat, *slong* prec)

void **arb_mat_charpoly**(*arb_poly_t* poly, const *arb_mat_t* mat, *slong* prec)

Sets *poly* to the characteristic polynomial of *mat* which must be a square matrix. If the matrix has n rows, the underscore method requires space for $n + 1$ output coefficients. Employs a division-free algorithm using $O(n^4)$ operations.

void **_arb_mat_companion**(*arb_mat_t* mat, *arb_srcptr* poly, *slong* prec)

void **arb_mat_companion**(*arb_mat_t* mat, const *arb_poly_t* poly, *slong* prec)

Sets the n by n matrix *mat* to the companion matrix of the polynomial *poly* which must have degree n . The underscore method reads $n + 1$ input coefficients.

9.15.16 Special functions

void **arb_mat_exp_taylor_sum**(*arb_mat_t* S, const *arb_mat_t* A, *slong* N, *slong* prec)

Sets *S* to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} A^k/k!$. Uses rectangular splitting to compute the sum using $O(\sqrt{N})$ matrix multiplications. The recurrence relation for factorials is used to get scalars that are small integers instead of full factorials. As in [Joh2014b], all divisions are postponed to the end by computing partial factorials of length $O(\sqrt{N})$. The scalars could be reduced by doing more divisions, but this appears to be slower in most cases.

void **arb_mat_exp**(*arb_mat_t* B, const *arb_mat_t* A, *slong* prec)

Sets *B* to the exponential of the matrix *A*, defined by the Taylor series

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

The function is evaluated as $\exp(A/2^r)^{2^r}$, where *r* is chosen to give rapid convergence.

The elementwise error when truncating the Taylor series after *N* terms is bounded by the error in the infinity norm, for which we have

$$\left\| \exp(2^{-r}A) - \sum_{k=0}^{N-1} \frac{(2^{-r}A)^k}{k!} \right\|_{\infty} = \left\| \sum_{k=N}^{\infty} \frac{(2^{-r}A)^k}{k!} \right\|_{\infty} \leq \sum_{k=N}^{\infty} \frac{(2^{-r}\|A\|_{\infty})^k}{k!}.$$

We bound the sum on the right using *mag_exp_tail()*. Truncation error is not added to entries whose values are determined by the sparsity structure of *A*.

void **arb_mat_trace**(*arb_t* trace, const *arb_mat_t* mat, *slong* prec)

Sets *trace* to the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square.

void **_arb_mat_diag_prod**(*arb_t* res, const *arb_mat_t* mat, *slong* a, *slong* b, *slong* prec)

void **arb_mat_diag_prod**(*arb_t* res, const *arb_mat_t* mat, *slong* prec)

Sets *res* to the product of the entries on the main diagonal of *mat*. The underscore method computes the product of the entries between index *a* inclusive and *b* exclusive (the indices must be in range).

9.15.17 Sparsity structure

slong **arb_mat_count_is_zero**(const *arb_mat_t* mat)

Returns the number of entries of *mat* that are certainly zero according to *arb_is_zero()*.

slong **arb_mat_count_not_is_zero**(const *arb_mat_t* mat)

Returns the number of entries of *mat* that are not certainly zero.

9.15.18 Component and error operations

void **arb_mat_get_mid**(*arb_mat_t* B, const *arb_mat_t* A)

Sets the entries of *B* to the exact midpoints of the entries of *A*.

void **arb_mat_add_error_mag**(*arb_mat_t* mat, const *mag_t* err)

Adds *err* in-place to the radii of the entries of *mat*.

9.15.19 Eigenvalues and eigenvectors

To compute eigenvalues and eigenvectors, one can convert to an `acb_mat_t` and use the functions in `acb_mat.h`: *Eigenvalues and eigenvectors*. In the future dedicated methods for real matrices will be added here.

9.15.20 LLL reduction

`int arb_mat_spd_get_fmpz_mat(fmpz_mat_t B, const arb_mat_t A, long prec)`

Attempts to set B to a symmetric and positive definite matrix obtained by rounding the midpoints of entries of $2^{prec} \cdot A$ to integers. Returns 1 on success. Returns 0 and leaves B undefined if A is not symmetric or the result of rounding is not a positive definite matrix. The warnings of `arf_get_fmpz()` apply.

`void arb_mat_spd_lll_reduce(fmpz_mat_t U, const arb_mat_t A, double delta, double eta, long prec)`

Given a symmetric positive definite matrix A , sets U to an invertible matrix such that $U^T A U$ is close to being LLL-reduced for the given parameters δ, η . If `arb_mat_spd_get_fmpz_mat()` succeeds at the chosen precision, we call `fmpz_lll()`, and otherwise set U to the identity matrix. The warnings of `arf_get_fmpz()` apply.

`int arb_mat_spd_is_lll_reduced(const arb_mat_t A, double delta, double eta, long prec)`

Given a symmetric positive definite matrix A , returns nonzero iff A is certainly LLL-reduced for the given LLL parameters η, δ , meaning that it satisfies the inequalities $|\mu_{j,k}| \leq \eta$ and $\delta \|b_{k-1}^*\|^2 \leq \|b_k^*\|^2 + \mu_{k,k-1}^2 \|b_{k-1}^*\|^2$ (with the usual notation).

9.16 acb_mat.h – matrices over the complex numbers

An `acb_mat_t` represents a dense matrix over the complex numbers, implemented as an array of entries of type `acb_struct`. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

Note: Methods prefixed with `acb_mat_approx` treat all input entries as floating-point numbers (ignoring the radii of the balls) and compute floating-point output (balls with zero radius) representing approximate solutions *without error bounds*. All other methods compute rigorous error bounds. The `approx` methods are typically useful for computing initial values or preconditioners for rigorous solvers. Some users may also find `approx` methods useful for doing ordinary numerical linear algebra in applications where error bounds are not needed.

9.16.1 Types, macros and constants

`type acb_mat_struct`

`type acb_mat_t`

Contains a pointer to a flat array of the entries (entries), an array of pointers to the start of each row (rows), and the number of rows (r) and columns (c).

An `acb_mat_t` is defined as an array of length one of type `acb_mat_struct`, permitting an `acb_mat_t` to be passed by reference.

`acb_mat_entry(mat, i, j)`

Macro giving a pointer to the entry at row i and column j .

acb_mat_nrows(mat)

Returns the number of rows of the matrix.

acb_mat_ncols(mat)

Returns the number of columns of the matrix.

9.16.2 Memory management

void **acb_mat_init**(*acb_mat_t* mat, *slong* r, *slong* c)

Initializes the matrix, setting it to the zero matrix with *r* rows and *c* columns.

void **acb_mat_clear**(*acb_mat_t* mat)

Clears the matrix, deallocating all entries.

slong **acb_mat_allocated_bytes**(const *acb_mat_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(acb_mat_struct)` to get the size of the object as a whole.

void **acb_mat_window_init**(*acb_mat_t* window, const *acb_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2)

Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive).

void **acb_mat_window_clear**(*acb_mat_t* window)

Frees the window matrix.

9.16.3 Conversions

void **acb_mat_set**(*acb_mat_t* dest, const *acb_mat_t* src)

void **acb_mat_set_fmpz_mat**(*acb_mat_t* dest, const *fmpz_mat_t* src)

void **acb_mat_set_round_fmpz_mat**(*acb_mat_t* dest, const *fmpz_mat_t* src, *slong* prec)

void **acb_mat_set_fmpq_mat**(*acb_mat_t* dest, const *fmpq_mat_t* src, *slong* prec)

void **acb_mat_set_arb_mat**(*acb_mat_t* dest, const *arb_mat_t* src)

void **acb_mat_set_round_arb_mat**(*acb_mat_t* dest, const *arb_mat_t* src, *slong* prec)

Sets *dest* to *src*. The operands must have identical dimensions.

void **acb_mat_get_real**(*arb_mat_t* re, const *acb_mat_t* mat)

void **acb_mat_get_imag**(*arb_mat_t* im, const *acb_mat_t* mat)

Sets *re* or *im* to the real or imaginary part of *mat*, respectively. The operands must have identical dimensions.

void **acb_mat_set_real_imag**(*acb_mat_t* mat, const *arb_mat_t* re, const *arb_mat_t* im)

Sets *mat* to the complex matrix with real and imaginary parts *re*, *im*. The operands must have identical dimensions.

9.16.4 Random generation

void **acb_mat_randtest**(*acb_mat_t* mat, *flint_rand_t* state, *slong* prec, *slong* mag_bits)
Sets *mat* to a random matrix with up to *prec* bits of precision and with exponents of width up to *mag_bits*.

void **acb_mat_randtest_eig**(*acb_mat_t* mat, *flint_rand_t* state, *acb_sreptr* E, *slong* prec)
Sets *mat* to a random matrix with the prescribed eigenvalues supplied as the vector *E*. The output matrix is required to be square. We generate a random unitary matrix via a matrix exponential, and then evaluate an inverse Schur decomposition.

9.16.5 Input and output

void **acb_mat_printd**(const *acb_mat_t* mat, *slong* digits)
Prints each entry in the matrix with the specified number of decimal digits.

void **acb_mat_fprintd**(FILE *file, const *acb_mat_t* mat, *slong* digits)
Prints each entry in the matrix with the specified number of decimal digits to the stream *file*.

9.16.6 Comparisons

Predicate methods return 1 if the property certainly holds and 0 otherwise.

int **acb_mat_equal**(const *acb_mat_t* mat1, const *acb_mat_t* mat2)
Returns whether the matrices have the same dimensions and identical intervals as entries.

int **acb_mat_overlaps**(const *acb_mat_t* mat1, const *acb_mat_t* mat2)
Returns whether the matrices have the same dimensions and each entry in *mat1* overlaps with the corresponding entry in *mat2*.

int **acb_mat_contains**(const *acb_mat_t* mat1, const *acb_mat_t* mat2)

int **acb_mat_contains_fmpz_mat**(const *acb_mat_t* mat1, const *fmpz_mat_t* mat2)

int **acb_mat_contains_fmpq_mat**(const *acb_mat_t* mat1, const *fmpq_mat_t* mat2)
Returns whether the matrices have the same dimensions and each entry in *mat2* is contained in the corresponding entry in *mat1*.

int **acb_mat_eq**(const *acb_mat_t* mat1, const *acb_mat_t* mat2)
Returns whether *mat1* and *mat2* certainly represent the same matrix.

int **acb_mat_ne**(const *acb_mat_t* mat1, const *acb_mat_t* mat2)
Returns whether *mat1* and *mat2* certainly do not represent the same matrix.

int **acb_mat_is_real**(const *acb_mat_t* mat)
Returns whether all entries in *mat* have zero imaginary part.

int **acb_mat_is_empty**(const *acb_mat_t* mat)
Returns whether the number of rows or the number of columns in *mat* is zero.

int **acb_mat_is_square**(const *acb_mat_t* mat)
Returns whether the number of rows is equal to the number of columns in *mat*.

int **acb_mat_is_exact**(const *acb_mat_t* mat)
Returns whether all entries in *mat* have zero radius.

int **acb_mat_is_zero**(const *acb_mat_t* mat)
Returns whether all entries in *mat* are exactly zero.

int **acb_mat_is_finite**(const *acb_mat_t* mat)

Returns whether all entries in *mat* are finite.

int **acb_mat_is_triu**(const *acb_mat_t* mat)

Returns whether *mat* is upper triangular; that is, all entries below the main diagonal are exactly zero.

int **acb_mat_is_tril**(const *acb_mat_t* mat)

Returns whether *mat* is lower triangular; that is, all entries above the main diagonal are exactly zero.

int **acb_mat_is_diag**(const *acb_mat_t* mat)

Returns whether *mat* is a diagonal matrix; that is, all entries off the main diagonal are exactly zero.

9.16.7 Special matrices

void **acb_mat_zero**(*acb_mat_t* mat)

Sets all entries in *mat* to zero.

void **acb_mat_one**(*acb_mat_t* mat)

Sets the entries on the main diagonal to ones, and all other entries to zero.

void **acb_mat_ones**(*acb_mat_t* mat)

Sets all entries in the matrix to ones.

void **acb_mat_onei**(*acb_mat_t* mat)

Sets the entries of the main diagonal to $i = \sqrt{-1}$ and all other entries to zero.

void **acb_mat_indeterminate**(*acb_mat_t* mat)

Sets all entries in the matrix to indeterminate (NaN).

void **acb_mat_dft**(*acb_mat_t* mat, int type, *slong* prec)

Sets *mat* to the DFT (discrete Fourier transform) matrix of order n where n is the smallest dimension of *mat* (if *mat* is not square, the matrix is extended periodically along the larger dimension). Here, we use the normalized DFT matrix

$$A_{j,k} = \frac{\omega^{jk}}{\sqrt{n}}, \quad \omega = e^{-2\pi i/n}.$$

The *type* parameter is currently ignored and should be set to 0. In the future, it might be used to select a different convention.

9.16.8 Transpose

void **acb_mat_transpose**(*acb_mat_t* dest, const *acb_mat_t* src)

Sets *dest* to the exact transpose *src*. The operands must have compatible dimensions. Aliasing is allowed for square matrices.

void **acb_mat_conjugate_transpose**(*acb_mat_t* dest, const *acb_mat_t* src)

Sets *dest* to the conjugate transpose of *src*. The operands must have compatible dimensions. Aliasing is allowed for square matrices.

void **acb_mat_conjugate**(*acb_mat_t* dest, const *acb_mat_t* src)

Sets *dest* to the elementwise complex conjugate of *src*.

9.16.9 Norms

void **acb_mat_bound_inf_norm**(*mag_t* b, const *acb_mat_t* A)

Sets *b* to an upper bound for the infinity norm (i.e. the largest absolute value row sum) of *A*.

void **acb_mat_frobenius_norm**(*arb_t* res, const *acb_mat_t* A, *slong* prec)

Sets *res* to the Frobenius norm (i.e. the square root of the sum of squares of entries) of *A*.

void **acb_mat_bound_frobenius_norm**(*mag_t* res, const *acb_mat_t* A)

Sets *res* to an upper bound for the Frobenius norm of *A*.

9.16.10 Arithmetic

void **acb_mat_neg**(*acb_mat_t* dest, const *acb_mat_t* src)

Sets *dest* to the exact negation of *src*. The operands must have the same dimensions.

void **acb_mat_add**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

Sets *res* to the sum of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_sub**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

Sets *res* to the difference of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_mul_classical**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

void **acb_mat_mul_threaded**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

void **acb_mat_mul_reorder**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

void **acb_mat_mul**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

Sets *res* to the matrix product of *mat1* and *mat2*. The operands must have compatible dimensions for matrix multiplication.

The *classical* version performs matrix multiplication in the trivial way.

The *threaded* version performs classical multiplication but splits the computation over the number of threads returned by *flint_get_num_threads()*.

The *reorder* version reorders the data and performs one to four real matrix multiplications via *arb_mat_mul()*.

The default version chooses an algorithm automatically.

void **acb_mat_mul_entrywise**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

Sets *res* to the entrywise product of *mat1* and *mat2*. The operands must have the same dimensions.

void **acb_mat_sqr_classical**(*acb_mat_t* res, const *acb_mat_t* mat, *slong* prec)

void **acb_mat_sqr**(*acb_mat_t* res, const *acb_mat_t* mat, *slong* prec)

Sets *res* to the matrix square of *mat*. The operands must both be square with the same dimensions.

void **acb_mat_pow_ui**(*acb_mat_t* res, const *acb_mat_t* mat, *ulong* exp, *slong* prec)

Sets *res* to *mat* raised to the power *exp*. Requires that *mat* is a square matrix.

void **acb_mat_approx_mul**(*acb_mat_t* res, const *acb_mat_t* mat1, const *acb_mat_t* mat2, *slong* prec)

Approximate matrix multiplication. The input radii are ignored and the output matrix is set to an approximate floating-point result. For performance reasons, the radii in the output matrix will *not* necessarily be written (zeroed), but will remain zero if they are already zeroed in *res* before calling this function.

9.16.11 Scalar arithmetic

void `acb_mat_scalar_mul_2exp_si`(*acb_mat_t* B, const *acb_mat_t* A, *slong* c)

Sets *B* to *A* multiplied by 2^c .

void `acb_mat_scalar_addmul_si`(*acb_mat_t* B, const *acb_mat_t* A, *slong* c, *slong* prec)

void `acb_mat_scalar_addmul_fmpz`(*acb_mat_t* B, const *acb_mat_t* A, const *fmpz_t* c, *slong* prec)

void `acb_mat_scalar_addmul_arb`(*acb_mat_t* B, const *acb_mat_t* A, const *arb_t* c, *slong* prec)

void `acb_mat_scalar_addmul_acb`(*acb_mat_t* B, const *acb_mat_t* A, const *acb_t* c, *slong* prec)

Sets *B* to $B + A \times c$.

void `acb_mat_scalar_mul_si`(*acb_mat_t* B, const *acb_mat_t* A, *slong* c, *slong* prec)

void `acb_mat_scalar_mul_fmpz`(*acb_mat_t* B, const *acb_mat_t* A, const *fmpz_t* c, *slong* prec)

void `acb_mat_scalar_mul_arb`(*acb_mat_t* B, const *acb_mat_t* A, const *arb_t* c, *slong* prec)

void `acb_mat_scalar_mul_acb`(*acb_mat_t* B, const *acb_mat_t* A, const *acb_t* c, *slong* prec)

Sets *B* to $A \times c$.

void `acb_mat_scalar_div_si`(*acb_mat_t* B, const *acb_mat_t* A, *slong* c, *slong* prec)

void `acb_mat_scalar_div_fmpz`(*acb_mat_t* B, const *acb_mat_t* A, const *fmpz_t* c, *slong* prec)

void `acb_mat_scalar_div_arb`(*acb_mat_t* B, const *acb_mat_t* A, const *arb_t* c, *slong* prec)

void `acb_mat_scalar_div_acb`(*acb_mat_t* B, const *acb_mat_t* A, const *acb_t* c, *slong* prec)

Sets *B* to A/c .

9.16.12 Vector arithmetic

void `_acb_mat_vector_mul_row`(*acb_ptr* res, *acb_srcptr* v, const *acb_mat_t* A, *slong* prec)

void `_acb_mat_vector_mul_col`(*acb_ptr* res, const *acb_mat_t* A, *acb_srcptr* v, *slong* prec)

void `acb_mat_vector_mul_row`(*acb_ptr* res, *acb_srcptr* v, const *acb_mat_t* A, *slong* prec)

void `acb_mat_vector_mul_col`(*acb_ptr* res, const *acb_mat_t* A, *acb_srcptr* v, *slong* prec)

Sets *res* to the product vA , (resp. Av), where *res* and *v* are seen as row (resp. column) vectors. The lengths of the vectors must match the dimensions of *A*.

The underscore methods do not allow aliasing between *res* and *v*.

9.16.13 Gaussian elimination and solving

```
int acb_mat_lu_classical(slong *perm, acb_mat_t LU, const acb_mat_t A, slong prec)
```

```
int acb_mat_lu_recursive(slong *perm, acb_mat_t LU, const acb_mat_t A, slong prec)
```

```
int acb_mat_lu(slong *perm, acb_mat_t LU, const acb_mat_t A, slong prec)
```

Given an $n \times n$ matrix A , computes an LU decomposition $PLU = A$ using Gaussian elimination with partial pivoting. The input and output matrices can be the same, performing the decomposition in-place.

Entry i in the permutation vector `perm` is set to the row index in the input matrix corresponding to row i in the output matrix.

The algorithm succeeds and returns nonzero if it can find n invertible (i.e. not containing zero) pivot entries. This guarantees that the matrix is invertible.

The algorithm fails and returns zero, leaving the entries in P and LU undefined, if it cannot find n invertible pivot elements. In this case, either the matrix is singular, the input matrix was computed to insufficient precision, or the LU decomposition was attempted at insufficient precision.

The *classical* version uses Gaussian elimination directly while the *recursive* version performs the computation in a block recursive way to benefit from fast matrix multiplication. The default version chooses an algorithm automatically.

```
void acb_mat_solve_tril_classical(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)
```

```
void acb_mat_solve_tril_recursive(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)
```

```
void acb_mat_solve_tril(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit, slong prec)
```

```
void acb_mat_solve_triu_classical(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
```

```
void acb_mat_solve_triu_recursive(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
```

```
void acb_mat_solve_triu(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit, slong prec)
```

Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. If `unit` is set, the main diagonal of L or U is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

```
void acb_mat_solve_lu_precomp(acb_mat_t X, const slong *perm, const acb_mat_t LU, const acb_mat_t B, slong prec)
```

Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU .

```
int acb_mat_solve(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
```

```
int acb_mat_solve_lu(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
```

```
int acb_mat_solve_precond(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
```

Solves $AX = B$ where A is a nonsingular $n \times n$ matrix and X and B are $n \times m$ matrices.

If $m > 0$ and A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that A is invertible and that the exact solution matrix is contained in the output.

Three algorithms are provided:

- The *lu* version performs LU decomposition directly in ball arithmetic. This is fast, but the bounds typically blow up exponentially with n , even if the system is well-conditioned. This algorithm is usually the best choice at very high precision.
- The *precond* version computes an approximate inverse to precondition the system. This is usually several times slower than direct LU decomposition, but the bounds do not blow up with n if the system is well-conditioned. This algorithm is usually the best choice for large systems at low to moderate precision.
- The default version selects between *lu* and *precomp* automatically.

The automatic choice should be reasonable most of the time, but users may benefit from trying either *lu* or *precond* in specific applications. For example, the *lu* solver often performs better for ill-conditioned systems where use of very high precision is unavoidable.

```
int acb_mat_inv(acb_mat_t X, const acb_mat_t A, slong prec)
```

Sets $X = A^{-1}$ where A is a square matrix, computed by solving the system $AX = I$.

If A cannot be inverted numerically (indicating either that A is singular or that the precision is insufficient), the values in the output matrix are left undefined and zero is returned. A nonzero return value guarantees that the matrix is invertible and that the exact inverse is contained in the output.

```
void acb_mat_det_lu(acb_t det, const acb_mat_t A, slong prec)
```

```
void acb_mat_det_precond(acb_t det, const acb_mat_t A, slong prec)
```

```
void acb_mat_det(acb_t det, const acb_mat_t A, slong prec)
```

Sets *det* to the determinant of the matrix A .

The *lu* version uses Gaussian elimination with partial pivoting. If at some point an invertible pivot element cannot be found, the elimination is stopped and the magnitude of the determinant of the remaining submatrix is bounded using Hadamard's inequality.

The *precond* version computes an approximate LU factorization of A and multiplies by the inverse L and U matrices as preconditioners to obtain a matrix close to the identity matrix [Rum2010]. An enclosure for this determinant is computed using Gershgorin circles. This is about four times slower than direct Gaussian elimination, but much more numerically stable.

The default version automatically selects between the *lu* and *precond* versions and additionally handles small or triangular matrices by direct formulas.

```
void acb_mat_approx_solve_triu(acb_mat_t X, const acb_mat_t U, const acb_mat_t B, int unit,
                               slong prec)
```

```
void acb_mat_approx_solve_tril(acb_mat_t X, const acb_mat_t L, const acb_mat_t B, int unit,
                               slong prec)
```

```
int acb_mat_approx_lu(slong *P, acb_mat_t LU, const acb_mat_t A, slong prec)
```

```
void acb_mat_approx_solve_lu_precomp(acb_mat_t X, const slong *perm, const acb_mat_t A,
                                     const acb_mat_t B, slong prec)
```

```
int acb_mat_approx_solve(acb_mat_t X, const acb_mat_t A, const acb_mat_t B, slong prec)
```


int **acb_mat_approx_inv**(*acb_mat_t* X, const *acb_mat_t* A, *slong* prec)

These methods perform approximate solving *without any error control*. The radii in the input matrices are ignored, the computations are done numerically with floating-point arithmetic (using ordinary Gaussian elimination and triangular solving, accelerated through the use of block recursive strategies for large matrices), and the output matrices are set to the approximate floating-point results with zeroed error bounds.

9.16.14 Characteristic polynomial and companion matrix

void **_acb_mat_charpoly**(*acb_ptr* poly, const *acb_mat_t* mat, *slong* prec)

void **acb_mat_charpoly**(*acb_poly_t* poly, const *acb_mat_t* mat, *slong* prec)

Sets *poly* to the characteristic polynomial of *mat* which must be a square matrix. If the matrix has n rows, the underscore method requires space for $n + 1$ output coefficients. Employs a division-free algorithm using $O(n^4)$ operations.

void **_acb_mat_companion**(*acb_mat_t* mat, *acb_sreptr* poly, *slong* prec)

void **acb_mat_companion**(*acb_mat_t* mat, const *acb_poly_t* poly, *slong* prec)

Sets the n by n matrix *mat* to the companion matrix of the polynomial *poly* which must have degree n . The underscore method reads $n + 1$ input coefficients.

9.16.15 Special functions

void **acb_mat_exp_taylor_sum**(*acb_mat_t* S, const *acb_mat_t* A, *slong* N, *slong* prec)

Sets *S* to the truncated exponential Taylor series $S = \sum_{k=0}^{N-1} A^k/k!$. See [arb_mat_exp_taylor_sum\(\)](#) for implementation notes.

void **acb_mat_exp**(*acb_mat_t* B, const *acb_mat_t* A, *slong* prec)

Sets *B* to the exponential of the matrix *A*, defined by the Taylor series

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

The function is evaluated as $\exp(A/2^r)^{2^r}$, where r is chosen to give rapid convergence of the Taylor series. Error bounds are computed as for [arb_mat_exp\(\)](#).

void **acb_mat_trace**(*acb_t* trace, const *acb_mat_t* mat, *slong* prec)

Sets *trace* to the trace of the matrix, i.e. the sum of entries on the main diagonal of *mat*. The matrix is required to be square.

void **_acb_mat_diag_prod**(*acb_t* res, const *acb_mat_t* mat, *slong* a, *slong* b, *slong* prec)

void **acb_mat_diag_prod**(*acb_t* res, const *acb_mat_t* mat, *slong* prec)

Sets *res* to the product of the entries on the main diagonal of *mat*. The underscore method computes the product of the entries between index a inclusive and b exclusive (the indices must be in range).

9.16.16 Component and error operations

void **acb_mat_get_mid**(*acb_mat_t* B, const *acb_mat_t* A)

Sets the entries of *B* to the exact midpoints of the entries of *A*.

void **acb_mat_add_error_mag**(*acb_mat_t* mat, const *mag_t* err)

Adds *err* in-place to the radii of the entries of *mat*.

9.16.17 Eigenvalues and eigenvectors

The functions in this section are experimental. There are classes of matrices where the algorithms fail to converge even as *prec* is increased, or for which the error bounds are much worse than necessary. In some cases, it can help to manually precondition the matrix *A* by applying a similarity transformation $T^{-1}AT$.

- If *A* is badly scaled, take *T* to be a matrix such that the entries of $T^{-1}AT$ are more uniform (this is known as balancing).
- Simply taking *T* to be a random invertible matrix can help if an algorithm fails to converge despite *A* being well-scaled. (This can be the case when dealing with multiple eigenvalues.)

int **acb_mat_approx_eig_qr**(*acb_ptr* E, *acb_mat_t* L, *acb_mat_t* R, const *acb_mat_t* A, const *mag_t* tol, *slong* maxiter, *slong* prec)

Computes floating-point approximations of all the *n* eigenvalues (and optionally eigenvectors) of the given *n* by *n* matrix *A*. The approximations of the eigenvalues are written to the vector *E*, in no particular order. If *L* is not *NULL*, approximations of the corresponding left eigenvectors are written to the rows of *L*. If *R* is not *NULL*, approximations of the corresponding right eigenvectors are written to the columns of *R*.

The parameters *tol* and *maxiter* can be used to control the target numerical error and the maximum number of iterations allowed before giving up. Passing *NULL* and 0 respectively results in default values being used.

Uses the implicitly shifted QR algorithm with reduction to Hessenberg form. No guarantees are made about the accuracy of the output. A nonzero return value indicates that the QR iteration converged numerically, but this is only a heuristic termination test and does not imply any statement whatsoever about error bounds. The output may also be accurate even if this function returns zero.

void **acb_mat_eig_global_enclosure**(*mag_t* eps, const *acb_mat_t* A, *acb_srcptr* E, const *acb_mat_t* R, *slong* prec)

Given an *n* by *n* matrix *A*, a length-*n* vector *E* containing approximations of the eigenvalues of *A*, and an *n* by *n* matrix *R* containing approximations of the corresponding right eigenvectors, computes a rigorous bound ε such that every eigenvalue λ of *A* satisfies $|\lambda - \hat{\lambda}_k| \leq \varepsilon$ for some $\hat{\lambda}_k$ in *E*. In other words, the union of the balls $B_k = \{z : |z - \hat{\lambda}_k| \leq \varepsilon\}$ is guaranteed to be an enclosure of all eigenvalues of *A*.

Note that there is no guarantee that each ball B_k can be identified with a single eigenvalue: it is possible that some balls contain several eigenvalues while other balls contain no eigenvalues. In other words, this method is not powerful enough to compute isolating balls for the individual eigenvalues (or even for clusters of eigenvalues other than the whole spectrum). Nevertheless, in practice the balls B_k will represent eigenvalues one-to-one with high probability if the given approximations are good.

The output can be used to certify that all eigenvalues of *A* lie in some region of the complex plane (such as a specific half-plane, strip, disk, or annulus) without the need to certify the individual eigenvalues. The output is easily converted into lower or upper bounds for the absolute values or real or imaginary parts of the spectrum, and with high probability these bounds will be tight. Using **acb_add_error_mag()** and **acb_union()**, the output can also be converted to a single

`acb_t` enclosing the whole spectrum of A in a rectangle, but note that to test whether a condition holds for all eigenvalues of A , it is typically better to iterate over the individual balls B_k .

This function implements the fast algorithm in Theorem 1 in [Miy2010] which extends the Bauer-Fike theorem. Approximations E and R can, for instance, be computed using `acb_mat_approx_eig_qr()`. No assumptions are made about the structure of A or the quality of the given approximations.

```
void acb_mat_eig_enclosure_rump(acb_t lambda, acb_mat_t J, acb_mat_t R, const acb_mat_t A,
                               const acb_t lambda_approx, const acb_mat_t R_approx, slong
                               prec)
```

Given an n by n matrix A and an approximate eigenvalue-eigenvector pair $lambda\_approx$ and R_approx (where R_approx is an n by 1 matrix), computes an enclosure $lambda$ guaranteed to contain at least one of the eigenvalues of A , along with an enclosure R for a corresponding right eigenvector.

More generally, this function can handle clustered (or repeated) eigenvalues. If R_approx is an n by k matrix containing approximate eigenvectors for a presumed cluster of k eigenvalues near $lambda\_approx$, this function computes an enclosure $lambda$ guaranteed to contain at least k eigenvalues of A along with a matrix R guaranteed to contain a basis for the k -dimensional invariant subspace associated with these eigenvalues. Note that for multiple eigenvalues, determining the individual eigenvectors is an ill-posed problem; describing an enclosure of the invariant subspace is the best we can hope for.

For $k = 1$, it is guaranteed that $AR - R\lambda$ contains the zero matrix. For $k > 2$, this cannot generally be guaranteed (in particular, A might not be diagonalizable). In this case, we can still compute an approximately diagonal k by k interval matrix $J \approx \lambda I$ such that $AR - RJ$ is guaranteed to contain the zero matrix. This matrix has the property that the Jordan canonical form of (any exact matrix contained in) A has a k by k submatrix equal to the Jordan canonical form of (some exact matrix contained in) J . The output J is optional (the user can pass `NULL` to omit it).

The algorithm follows section 13.4 in [Rum2010], corresponding to the `verifyeig()` routine in INTLAB. The initial approximations can, for instance, be computed using `acb_mat_approx_eig_qr()`. No assumptions are made about the structure of A or the quality of the given approximations.

```
int acb_mat_eig_simple_rump(acb_ptr E, acb_mat_t L, acb_mat_t R, const acb_mat_t A,
                           acb_srcptr E_approx, const acb_mat_t R_approx, slong prec)
```

```
int acb_mat_eig_simple_vdhoeven_mourrain(acb_ptr E, acb_mat_t L, acb_mat_t R, const
                                         acb_mat_t A, acb_srcptr E_approx, const acb_mat_t
                                         R_approx, slong prec)
```

```
int acb_mat_eig_simple(acb_ptr E, acb_mat_t L, acb_mat_t R, const acb_mat_t A, acb_srcptr
                       E_approx, const acb_mat_t R_approx, slong prec)
```

Computes all the eigenvalues (and optionally corresponding eigenvectors) of the given n by n matrix A .

Attempts to prove that A has n simple (isolated) eigenvalues, returning 1 if successful and 0 otherwise. On success, isolating complex intervals for the eigenvalues are written to the vector E , in no particular order. If L is not `NULL`, enclosures of the corresponding left eigenvectors are written to the rows of L . If R is not `NULL`, enclosures of the corresponding right eigenvectors are written to the columns of R .

The left eigenvectors are normalized so that $L = R^{-1}$. This produces a diagonalization $LAR = D$ where D is the diagonal matrix with the entries in E on the diagonal.

The user supplies approximations E_approx and R_approx of the eigenvalues and the right eigenvectors. The initial approximations can, for instance, be computed using `acb_mat_approx_eig_qr()`. No assumptions are made about the structure of A or the quality of the given approximations.

Two algorithms are implemented:

- The *rump* version calls `acb_mat_eig_enclosure_rump()` repeatedly to certify eigenvalue-eigenvector pairs one by one. The iteration is stopped to return non-success if a new eigenvalue overlaps with previously computed one. Finally, L is computed by a matrix inversion. This has complexity $O(n^4)$.
- The *vdhoeven_mourrain* version uses the algorithm in [HM2017] to certify all eigenvalues and eigenvectors in one step. This has complexity $O(n^3)$.

The default version currently uses *vdhoeven_mourrain*.

By design, these functions terminate instead of attempting to compute eigenvalue clusters if some eigenvalues cannot be isolated. To compute all eigenvalues of a matrix allowing for overlap, `acb_mat_eig_multiple_rump()` may be used as a fallback, or `acb_mat_eig_multiple()` may be used in the first place.

```
int acb_mat_eig_multiple_rump(acb_ptr E, const acb_mat_t A, acb_srcptr E_approx, const
                             acb_mat_t R_approx, slong prec)
```

```
int acb_mat_eig_multiple(acb_ptr E, const acb_mat_t A, acb_srcptr E_approx, const acb_mat_t
                         R_approx, slong prec)
```

Computes all the eigenvalues of the given n by n matrix A . On success, the output vector E contains n complex intervals, each representing one eigenvalue of A with the correct multiplicities in case of overlap. The output intervals are either disjoint or identical, and identical intervals are guaranteed to be grouped consecutively. Each complete run of k identical intervals thus represents a cluster of exactly k eigenvalues which could not be separated from each other at the current precision, but which could be isolated from the other $n - k$ eigenvalues of the matrix.

The user supplies approximations E_approx and R_approx of the eigenvalues and the right eigenvectors. The initial approximations can, for instance, be computed using `acb_mat_approx_eig_qr()`. No assumptions are made about the structure of A or the quality of the given approximations.

The *rump* algorithm groups approximate eigenvalues that are close and calls `acb_mat_eig_enclosure_rump()` repeatedly to validate each cluster. The complexity is $O(mn^3)$ for m clusters.

The default version, as currently implemented, first attempts to call `acb_mat_eig_simple_vdhoeven_mourrain()` hoping that the eigenvalues are actually simple. It then uses the *rump* algorithm as a fallback.

9.17 acb_hypgeom.h – hypergeometric functions of complex variables

The generalized hypergeometric function is formally defined by

$${}_pF_q(a_1, \dots, a_p; b_1, \dots, b_q; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \dots (a_p)_k}{(b_1)_k \dots (b_q)_k} \frac{z^k}{k!}.$$

It can be interpreted using analytic continuation or regularization when the sum does not converge. In a looser sense, we understand “hypergeometric functions” to be linear combinations of generalized hypergeometric functions with prefactors that are products of exponentials, powers, and gamma functions.

9.17.1 Rising factorials

```
void acb_hypgeom_rising_ui_forward(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising_ui_bs(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising_ui_rs(acb_t res, const acb_t x, ulong n, ulong m, slong prec)
void acb_hypgeom_rising_ui_rec(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising_ui(acb_t res, const acb_t x, ulong n, slong prec)
void acb_hypgeom_rising(acb_t res, const acb_t x, const acb_t n, slong prec)
```

Computes the rising factorial $(x)_n$.

The *forward* version uses the forward recurrence. The *bs* version uses binary splitting. The *rs* version uses rectangular splitting. It takes an extra tuning parameter m which can be set to zero to choose automatically. The *rec* version chooses an algorithm automatically, avoiding use of the gamma function (so that it can be used in the computation of the gamma function). The default versions (*rising_ui* and *rising_ui*) choose an algorithm automatically and may additionally fall back on the gamma function.

```
void acb_hypgeom_rising_ui_jet_powsum(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)
void acb_hypgeom_rising_ui_jet_bs(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)
void acb_hypgeom_rising_ui_jet_rs(acb_ptr res, const acb_t x, ulong n, ulong m, slong len, slong
    prec)
void acb_hypgeom_rising_ui_jet(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)
```

Computes the jet of the rising factorial $(x)_n$, truncated to length len . In other words, constructs the polynomial $(X + x)_n \in \mathbb{R}[X]$, truncated if $len < n + 1$ (and zero-extended if $len > n + 1$).

The *powsum* version computes the sequence of powers of x and forms integral linear combinations of these. The *bs* version uses binary splitting. The *rs* version uses rectangular splitting. It takes an extra tuning parameter m which can be set to zero to choose automatically. The default version chooses an algorithm automatically.

```
void acb_hypgeom_log_rising_ui(acb_ptr res, const acb_t x, ulong n, slong prec)
```

Computes the log-rising factorial $\log(x)_n = \sum_{k=0}^{n-1} \log(x + k)$.

This first computes the ordinary rising factorial and then determines the branch correction $2\pi im$ with respect to the principal logarithm. The correction is computed using Hare's algorithm in floating-point arithmetic if this is safe; otherwise, a direct computation of $\sum_{k=0}^{n-1} \arg(x + k)$ is used as a fallback.

```
void acb_hypgeom_log_rising_ui_jet(acb_ptr res, const acb_t x, ulong n, slong len, slong prec)
```

Computes the jet of the log-rising factorial $\log(x)_n$, truncated to length len .

9.17.2 Gamma function

```
void acb_hypgeom_gamma_stirling_sum_horner(acb_t s, const acb_t z, slong N, slong prec)
void acb_hypgeom_gamma_stirling_sum_improved(acb_t s, const acb_t z, slong N, slong K, slong
    prec)
```

Sets *res* to the final sum in the Stirling series for the gamma function truncated before the term with index N , i.e. computes $\sum_{n=1}^{N-1} B_{2n}/(2n(2n-1)z^{2n-1})$. The *horner* version uses Horner scheme with gradual precision adjustments. The *improved* version uses rectangular splitting for the low-index terms and reexpands the high-index terms as hypergeometric polynomials, using a splitting parameter K (which can be set to 0 to use a default value).

```
void acb_hypgeom_gamma_stirling(acb_t res, const acb_t x, int reciprocal, slong prec)
```

Sets *res* to the gamma function of x computed using the Stirling series together with argument reduction. If *reciprocal* is set, the reciprocal gamma function is computed instead.

int **acb_hypgeom_gamma_taylor**(*acb_t* res, const *acb_t* x, int reciprocal, *slong* prec)

Attempts to compute the gamma function of x using Taylor series together with argument reduction. This is only supported if x and $prec$ are both small enough. If successful, returns 1; otherwise, does nothing and returns 0. If *reciprocal* is set, the reciprocal gamma function is computed instead.

void **acb_hypgeom_gamma**(*acb_t* res, const *acb_t* x, *slong* prec)

Sets *res* to the gamma function of x computed using a default algorithm choice.

void **acb_hypgeom_rgamma**(*acb_t* res, const *acb_t* x, *slong* prec)

Sets *res* to the reciprocal gamma function of x computed using a default algorithm choice.

void **acb_hypgeom_lgamma**(*acb_t* res, const *acb_t* x, *slong* prec)

Sets *res* to the principal branch of the log-gamma function of x computed using a default algorithm choice.

9.17.3 Convergent series

In this section, we define

$$T(k) = \frac{\prod_{i=0}^{p-1} (a_i)_k}{\prod_{i=0}^{q-1} (b_i)_k} z^k$$

and

$${}_p f_q(a_0, \dots, a_{p-1}; b_0 \dots b_{q-1}; z) = {}_{p+1} F_q(a_0, \dots, a_{p-1}, 1; b_0 \dots b_{q-1}; z) = \sum_{k=0}^{\infty} T(k)$$

For the conventional generalized hypergeometric function ${}_p F_q$, compute ${}_p f_{q+1}$ with the explicit parameter $b_q = 1$, or remove a 1 from the a_i parameters if there is one.

void **acb_hypgeom_pfq_bound_factor**(*mag_t* C, *acb_srcptr* a, *slong* p, *acb_srcptr* b, *slong* q, const *acb_t* z, *ulong* n)

Computes a factor C such that $|\sum_{k=n}^{\infty} T(k)| \leq C|T(n)|$. See *Convergent series*. As currently implemented, the bound becomes infinite when n is too small, even if the series converges.

slong **acb_hypgeom_pfq_choose_n**(*acb_srcptr* a, *slong* p, *acb_srcptr* b, *slong* q, const *acb_t* z, *slong* prec)

Heuristically attempts to choose a number of terms n to sum of a hypergeometric series at a working precision of $prec$ bits.

Uses double precision arithmetic internally. As currently implemented, it can fail to produce a good result if the parameters are extremely large or extremely close to nonpositive integers.

Numerical cancellation is assumed to be significant, so truncation is done when the current term is $prec$ bits smaller than the largest encountered term.

This function will also attempt to pick a reasonable truncation point for divergent series.

void **acb_hypgeom_pfq_sum_forward**(*acb_t* s, *acb_t* t, *acb_srcptr* a, *slong* p, *acb_srcptr* b, *slong* q, const *acb_t* z, *slong* n, *slong* prec)

void **acb_hypgeom_pfq_sum_rs**(*acb_t* s, *acb_t* t, *acb_srcptr* a, *slong* p, *acb_srcptr* b, *slong* q, const *acb_t* z, *slong* n, *slong* prec)

void **acb_hypgeom_pfq_sum_bs**(*acb_t* s, *acb_t* t, *acb_srcptr* a, *slong* p, *acb_srcptr* b, *slong* q, const *acb_t* z, *slong* n, *slong* prec)

void **acb_hypgeom_pfq_sum_fme**(*acb_t* s, *acb_t* t, *acb_srcptr* a, *slong* p, *acb_srcptr* b, *slong* q, const *acb_t* z, *slong* n, *slong* prec)


```
void acb_hypgeom_pfq_sum(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const
                        acb_t z, slong n, slong prec)
```

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$. Does not allow aliasing between input and output variables. We require $n \geq 0$.

The *forward* version computes the sum using forward recurrence.

The *bs* version computes the sum using binary splitting.

The *rs* version computes the sum in reverse order using rectangular splitting. It only computes a magnitude bound for the value of t .

The *fme* version uses fast multipoint evaluation.

The default version automatically chooses an algorithm depending on the inputs.

```
void acb_hypgeom_pfq_sum_bs_invz(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q,
                                const acb_t w, slong n, slong prec)
```

```
void acb_hypgeom_pfq_sum_invz(acb_t s, acb_t t, acb_srcptr a, slong p, acb_srcptr b, slong q, const
                              acb_t z, const acb_t w, slong n, slong prec)
```

Like `acb_hypgeom_pfq_sum()`, but taking advantage of $w = 1/z$ possibly having few bits.

```
void acb_hypgeom_pfq_direct(acb_t res, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z,
                           slong n, slong prec)
```

Computes

$${}_p f_q(z) = \sum_{k=0}^{\infty} T(k) = \sum_{k=0}^{n-1} T(k) + \varepsilon$$

directly from the defining series, including a rigorous bound for the truncation error ε in the output.

If $n < 0$, this function chooses a number of terms automatically using `acb_hypgeom_pfq_choose_n()`.

```
void acb_hypgeom_pfq_series_sum_forward(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a,
                                        slong p, const acb_poly_struct *b, slong q, const
                                        acb_poly_t z, int regularized, slong n, slong len, slong
                                        prec)
```

```
void acb_hypgeom_pfq_series_sum_bs(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong
p, const acb_poly_struct *b, slong q, const acb_poly_t z, int
regularized, slong n, slong len, slong prec)
```

```
void acb_hypgeom_pfq_series_sum_rs(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong
p, const acb_poly_struct *b, slong q, const acb_poly_t z, int
regularized, slong n, slong len, slong prec)
```

```
void acb_hypgeom_pfq_series_sum(acb_poly_t s, acb_poly_t t, const acb_poly_struct *a, slong p,
                                const acb_poly_struct *b, slong q, const acb_poly_t z, int
                                regularized, slong n, slong len, slong prec)
```

Computes $s = \sum_{k=0}^{n-1} T(k)$ and $t = T(n)$ given parameters and argument that are power series. Does not allow aliasing between input and output variables. We require $n \geq 0$ and that *len* is positive.

If *regularized* is set, the regularized sum is computed, avoiding division by zero at the poles of the gamma function.

The *forward*, *bs*, *rs* and default versions use forward recurrence, binary splitting, rectangular splitting, and an automatic algorithm choice.


```
void acb_hypgeom_pfq_series_direct(acb_poly_t res, const acb_poly_struct *a, slong p, const
                                acb_poly_struct *b, slong q, const acb_poly_t z, int
                                regularized, slong n, slong len, slong prec)
```

Computes ${}_pF_q(z)$ directly using the defining series, given parameters and argument that are power series. The result is a power series of length *len*. We require that *len* is positive.

An error bound is computed automatically as a function of the number of terms *n*. If *n* < 0, the number of terms is chosen automatically.

If *regularized* is set, the regularized hypergeometric function is computed instead.

9.17.4 Asymptotic series

$U(a, b, z)$ is the confluent hypergeometric function of the second kind with the principal branch cut, and $U^* = z^a U(a, b, z)$. For details about how error bounds are computed, see [Asymptotic series for the confluent hypergeometric function](#).

```
void acb_hypgeom_u_asymp(acb_t res, const acb_t a, const acb_t b, const acb_t z, slong n, slong
                        prec)
```

Sets *res* to $U^*(a, b, z)$ computed using *n* terms of the asymptotic series, with a rigorous bound for the error included in the output. We require $n \geq 0$.

```
int acb_hypgeom_u_use_asymp(const acb_t z, slong prec)
```

Heuristically determines whether the asymptotic series can be used to evaluate $U(a, b, z)$ to *prec* accurate bits (assuming that *a* and *b* are small).

9.17.5 Generalized hypergeometric function

```
void acb_hypgeom_pfq(acb_t res, acb_srcptr a, slong p, acb_srcptr b, slong q, const acb_t z, int
                    regularized, slong prec)
```

Computes the generalized hypergeometric function ${}_pF_q(z)$, or the regularized version if *regularized* is set.

This function automatically delegates to a specialized implementation when the order (p, q) is one of (0,0), (1,0), (0,1), (1,1), (2,1). Otherwise, it falls back to direct summation.

While this is a top-level function meant to take care of special cases automatically, it does not generally perform the optimization of deleting parameters that appear in both *a* and *b*. This can be done ahead of time by the user in applications where duplicate parameters are likely to occur.

9.17.6 Confluent hypergeometric functions

```
void acb_hypgeom_u_1f1_series(acb_poly_t res, const acb_poly_t a, const acb_poly_t b, const
                             acb_poly_t z, slong len, slong prec)
```

Computes $U(a, b, z)$ as a power series truncated to length *len*, given $a, b, z \in \mathbb{C}[[x]]$. If $b[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $b[1] \neq 0$). As currently implemented, the output is indeterminate if *b* is nonexact and contains an integer.

```
void acb_hypgeom_u_1f1(acb_t res, const acb_t a, const acb_t b, const acb_t z, slong prec)
```

Computes $U(a, b, z)$ as a sum of two convergent hypergeometric series. If $b \in \mathbb{Z}$, it computes the limit value via [acb_hypgeom_u_1f1_series\(\)](#). As currently implemented, the output is indeterminate if *b* is nonexact and contains an integer.

```
void acb_hypgeom_u(acb_t res, const acb_t a, const acb_t b, const acb_t z, slong prec)
```

Computes $U(a, b, z)$ using an automatic algorithm choice. The function [acb_hypgeom_u_asymp\(\)](#) is used if *a* or $a - b + 1$ is a nonpositive integer (in which case the asymptotic series terminates), or if *z* is sufficiently large. Otherwise [acb_hypgeom_u_1f1\(\)](#) is used.

void **acb_hypgeom_m_asymp**(*acb_t* res, const *acb_t* a, const *acb_t* b, const *acb_t* z, int regularized, *slong* prec)

void **acb_hypgeom_m_1f1**(*acb_t* res, const *acb_t* a, const *acb_t* b, const *acb_t* z, int regularized, *slong* prec)

void **acb_hypgeom_m**(*acb_t* res, const *acb_t* a, const *acb_t* b, const *acb_t* z, int regularized, *slong* prec)

Computes the confluent hypergeometric function $M(a, b, z) = {}_1F_1(a, b, z)$, or $\mathbf{M}(a, b, z) = \frac{1}{\Gamma(b)} {}_1F_1(a, b, z)$ if *regularized* is set.

void **acb_hypgeom_1f1**(*acb_t* res, const *acb_t* a, const *acb_t* b, const *acb_t* z, int regularized, *slong* prec)

Alias for **acb_hypgeom_m()**.

void **acb_hypgeom_0f1_asymp**(*acb_t* res, const *acb_t* a, const *acb_t* z, int regularized, *slong* prec)

void **acb_hypgeom_0f1_direct**(*acb_t* res, const *acb_t* a, const *acb_t* z, int regularized, *slong* prec)

void **acb_hypgeom_0f1**(*acb_t* res, const *acb_t* a, const *acb_t* z, int regularized, *slong* prec)

Computes the confluent hypergeometric function ${}_0F_1(a, z)$, or $\frac{1}{\Gamma(a)} {}_0F_1(a, z)$ if *regularized* is set, using asymptotic expansions, direct summation, or an automatic algorithm choice. The *asymp* version uses the asymptotic expansions of Bessel functions, together with the connection formulas

$$\frac{{}_0F_1(a, z)}{\Gamma(a)} = (-z)^{(1-a)/2} J_{a-1}(2\sqrt{-z}) = z^{(1-a)/2} I_{a-1}(2\sqrt{z}).$$

The Bessel- J function is used in the left half-plane and the Bessel- I function is used in the right half-plane, to avoid loss of accuracy due to evaluating the square root on the branch cut.

9.17.7 Error functions and Fresnel integrals

void **acb_hypgeom_erf_propagated_error**(*mag_t* re, *mag_t* im, const *acb_t* z)

Sets *re* and *im* to upper bounds for the error in the real and imaginary part resulting from approximating the error function of z by the error function evaluated at the midpoint of z . Uses the first derivative.

void **acb_hypgeom_erf_1f1a**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_erf_1f1b**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_erf_asymp**(*acb_t* res, const *acb_t* z, int complementary, *slong* prec, *slong* prec2)

Computes the error function respectively using

$$\begin{aligned} \operatorname{erf}(z) &= \frac{2z}{\sqrt{\pi}} {}_1F_1\left(\frac{1}{2}, \frac{3}{2}, -z^2\right) \\ \operatorname{erf}(z) &= \frac{2ze^{-z^2}}{\sqrt{\pi}} {}_1F_1\left(1, \frac{3}{2}, z^2\right) \\ \operatorname{erf}(z) &= \frac{z}{\sqrt{z^2}} \left(1 - \frac{e^{-z^2}}{\sqrt{\pi}} U\left(\frac{1}{2}, \frac{1}{2}, z^2\right)\right) = \frac{z}{\sqrt{z^2}} - \frac{e^{-z^2}}{z\sqrt{\pi}} U^*\left(\frac{1}{2}, \frac{1}{2}, z^2\right). \end{aligned}$$

The *asymp* version takes a second precision to use for the U term. It also takes an extra flag *complementary*, computing the complementary error function if set.

void **acb_hypgeom_erf**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the error function using an automatic algorithm choice. If z is too small to use the asymptotic expansion, a working precision sufficient to circumvent cancellation in the hypergeometric series is determined automatically, and a bound for the propagated error is computed with **acb_hypgeom_erf_propagated_error()**.

void `_acb_hypgeom_erf_series`(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void `acb_hypgeom_erf_series`(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)
 Computes the error function of the power series z, truncated to length len.

void `acb_hypgeom_erfc`(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes the complementary error function $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. This function avoids catastrophic cancellation for large positive z.

void `_acb_hypgeom_erfc_series`(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void `acb_hypgeom_erfc_series`(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)
 Computes the complementary error function of the power series z, truncated to length len.

void `acb_hypgeom_erfi`(*acb_t* res, const *acb_t* z, *slong* prec)
 Computes the imaginary error function $\operatorname{erfi}(z) = -i \operatorname{erf}(iz)$. This is a trivial wrapper of `acb_hypgeom_erf()`.

void `_acb_hypgeom_erfi_series`(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void `acb_hypgeom_erfi_series`(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)
 Computes the imaginary error function of the power series z, truncated to length len.

void `acb_hypgeom_fresnel`(*acb_t* res1, *acb_t* res2, const *acb_t* z, int normalized, *slong* prec)
 Sets *res1* to the Fresnel sine integral $S(z)$ and *res2* to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing `NULL` as the other output variable. The definition $S(z) = \int_0^z \sin(t^2) dt$ is used if *normalized* is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2) dt$ is used if *normalized* is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void `_acb_hypgeom_fresnel_series`(*acb_ptr* res1, *acb_ptr* res2, *acb_srcptr* z, *slong* zlen, int normalized, *slong* len, *slong* prec)

void `acb_hypgeom_fresnel_series`(*acb_poly_t* res1, *acb_poly_t* res2, const *acb_poly_t* z, int normalized, *slong* len, *slong* prec)
 Sets *res1* to the Fresnel sine integral and *res2* to the Fresnel cosine integral of the power series z, truncated to length len. Optionally, just a single function can be computed by passing `NULL` as the other output variable.

9.17.8 Bessel functions

void `acb_hypgeom_bessel_j_deriv_bound`(*mag_t* res, const *acb_t* nu, const *acb_t* z, *ulong* d)
 Sets *res* to a bound, possibly crude, for $|J_\nu^{(d)}(z)|$. Currently only specialized for small integer ν and small d .

void `acb_hypgeom_bessel_j_asymp`(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)
 Computes the Bessel function of the first kind via `acb_hypgeom_u_asymp()`. For all complex ν, z , we have

$$J_\nu(z) = \frac{z^\nu}{2^\nu e^{iz} \Gamma(\nu + 1)} {}_1F_1(\nu + \tfrac{1}{2}, 2\nu + 1, 2iz) = A_+ B_+ + A_- B_-$$

where

$$A_\pm = z^\nu (z^2)^{-\frac{1}{2}-\nu} (\mp iz)^{\frac{1}{2}+\nu} (2\pi)^{-1/2} = (\pm iz)^{-1/2-\nu} z^\nu (2\pi)^{-1/2}$$

$$B_\pm = e^{\pm iz} U^*(\nu + \tfrac{1}{2}, 2\nu + 1, \mp 2iz).$$

Nicer representations of the factors $A_\pm$ can be given depending conditionally on the parameters. If $\nu + \frac{1}{2} = n \in \mathbb{Z}$, we have $A_\pm = (\pm i)^n (2\pi z)^{-1/2}$. And if $\operatorname{Re}(z) > 0$, we have $A_\pm = \exp(\mp i[(2\nu + 1)/4]\pi) (2\pi z)^{-1/2}$.

void **acb_hypgeom_bessel_j_0f1**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

Computes the Bessel function of the first kind from

$$J_\nu(z) = \frac{1}{\Gamma(\nu+1)} \left(\frac{z}{2}\right)^\nu {}_0F_1\left(\nu+1, -\frac{z^2}{4}\right).$$

void **acb_hypgeom_bessel_j**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

Computes the Bessel function of the first kind $J_\nu(z)$ using an automatic algorithm choice.

void **acb_hypgeom_bessel_y**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

Computes the Bessel function of the second kind $Y_\nu(z)$ from the formula

$$Y_\nu(z) = \frac{\cos(\nu\pi)J_\nu(z) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

unless $\nu = n$ is an integer in which case the limit value

$$Y_n(z) = -\frac{2}{\pi} (i^n K_n(iz) + [\log(iz) - \log(z)] J_n(z))$$

is computed. As currently implemented, the output is indeterminate if ν is nonexact and contains an integer.

void **acb_hypgeom_bessel_jy**(*acb_t* res1, *acb_t* res2, const *acb_t* nu, const *acb_t* z, *slong* prec)

Sets *res1* to $J_\nu(z)$ and *res2* to $Y_\nu(z)$, computed simultaneously. From these values, the user can easily construct the Bessel functions of the third kind (Hankel functions) $H_\nu^{(1)}(z)$, $H_\nu^{(2)}(z) = J_\nu(z) \pm iY_\nu(z)$.

9.17.9 Modified Bessel functions

void **acb_hypgeom_bessel_i_asymp**(*acb_t* res, const *acb_t* nu, const *acb_t* z, int scaled, *slong* prec)

void **acb_hypgeom_bessel_i_0f1**(*acb_t* res, const *acb_t* nu, const *acb_t* z, int scaled, *slong* prec)

void **acb_hypgeom_bessel_i**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

void **acb_hypgeom_bessel_i_scaled**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

Computes the modified Bessel function of the first kind $I_\nu(z) = z^\nu(iz)^{-\nu}J_\nu(iz)$ respectively using asymptotic series (see [acb_hypgeom_bessel_j_asymp\(\)](#)), the convergent series

$$I_\nu(z) = \frac{1}{\Gamma(\nu+1)} \left(\frac{z}{2}\right)^\nu {}_0F_1\left(\nu+1, \frac{z^2}{4}\right),$$

or an automatic algorithm choice.

The *scaled* version computes the function $e^{-z}I_\nu(z)$. The *asymp* and *0f1* functions implement both variants and allow choosing with a flag.

void **acb_hypgeom_bessel_k_asymp**(*acb_t* res, const *acb_t* nu, const *acb_t* z, int scaled, *slong* prec)

Computes the modified Bessel function of the second kind via [acb_hypgeom_u_asymp\(\)](#). For all ν and all $z \neq 0$, we have

$$K_\nu(z) = \left(\frac{2z}{\pi}\right)^{-1/2} e^{-z} U^*(\nu + \tfrac{1}{2}, 2\nu + 1, 2z).$$

If *scaled* is set, computes the function $e^z K_\nu(z)$.

void **acb_hypgeom_bessel_k_0f1_series**(*acb_poly_t* res, const *acb_poly_t* nu, const *acb_poly_t* z, int scaled, *slong* len, *slong* prec)

Computes the modified Bessel function of the second kind $K_\nu(z)$ as a power series truncated to length *len*, given $\nu, z \in \mathbb{C}[[x]]$. Uses the formula

$$K_\nu(z) = \frac{1}{2} \frac{\pi}{\sin(\pi\nu)} \left[\left(\frac{z}{2}\right)^{-\nu} {}_0\tilde{F}_1\left(1-\nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^\nu {}_0\tilde{F}_1\left(1+\nu, \frac{z^2}{4}\right) \right].$$

If $\nu[0] \in \mathbb{Z}$, it computes one extra derivative and removes the singularity (it is then assumed that $\nu[1] \neq 0$). As currently implemented, the output is indeterminate if $\nu[0]$ is nonexact and contains an integer.

If *scaled* is set, computes the function $e^z K_\nu(z)$.

void **acb_hypgeom_bessel_k_0f1**(*acb_t* res, const *acb_t* nu, const *acb_t* z, int scaled, *slong* prec)

Computes the modified Bessel function of the second kind from

$$K_\nu(z) = \frac{1}{2} \left[\left(\frac{z}{2}\right)^{-\nu} \Gamma(\nu) {}_0F_1\left(1-\nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^\nu \frac{\pi}{\nu \sin(\pi\nu) \Gamma(\nu)} {}_0F_1\left(\nu+1, \frac{z^2}{4}\right) \right]$$

if $\nu \notin \mathbb{Z}$. If $\nu \in \mathbb{Z}$, it computes the limit value via [acb_hypgeom_bessel_k_0f1_series\(\)](#). As currently implemented, the output is indeterminate if ν is nonexact and contains an integer.

If *scaled* is set, computes the function $e^z K_\nu(z)$.

void **acb_hypgeom_bessel_k**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

Computes the modified Bessel function of the second kind $K_\nu(z)$ using an automatic algorithm choice.

void **acb_hypgeom_bessel_k_scaled**(*acb_t* res, const *acb_t* nu, const *acb_t* z, *slong* prec)

Computes the function $e^z K_\nu(z)$.

9.17.10 Airy functions

The Airy functions are linearly independent solutions of the differential equation $y'' - zy = 0$. All solutions are entire functions. The standard solutions are denoted $\text{Ai}(z), \text{Bi}(z)$. For negative z , both functions are oscillatory. For positive z , the first function decreases exponentially while the second increases exponentially.

The Airy functions can be expressed in terms of Bessel functions of fractional order, but this is inconvenient since such formulas only hold piecewise (due to the Stokes phenomenon). Computation of the Airy functions can also be optimized more than Bessel functions in general. We therefore provide a dedicated interface for evaluating Airy functions.

The following methods optionally compute $(\text{Ai}(z), \text{Ai}'(z), \text{Bi}(z), \text{Bi}'(z))$ simultaneously. Any of the four function values can be omitted by passing *NULL* for the unwanted output variables, speeding up the evaluation.

void **acb_hypgeom_airy_direct**(*acb_t* ai, *acb_t* ai_prime, *acb_t* bi, *acb_t* bi_prime, const *acb_t* z, *slong* n, *slong* prec)

Computes the Airy functions using direct series expansions truncated at n terms. Error bounds are included in the output.

void **acb_hypgeom_airy_asymp**(*acb_t* ai, *acb_t* ai_prime, *acb_t* bi, *acb_t* bi_prime, const *acb_t* z, *slong* n, *slong* prec)

Computes the Airy functions using asymptotic expansions truncated at n terms. Error bounds are included in the output. For details about how the error bounds are computed, see [Asymptotic series for Airy functions](#).

```
void acb_hypgeom_airy_bound(mag_t ai, mag_t ai_prime, mag_t bi, mag_t bi_prime, const acb_t z)
```

Computes bounds for the Airy functions using first-order asymptotic expansions together with error bounds. This function uses some shortcuts to make it slightly faster than calling `acb_hypgeom_airy_asymp()` with $n = 1$.

```
void acb_hypgeom_airy(acb_t ai, acb_t ai_prime, acb_t bi, acb_t bi_prime, const acb_t z, slong prec)
```

Computes Airy functions using an automatic algorithm choice.

We use `acb_hypgeom_airy_asymp()` whenever this gives full accuracy and `acb_hypgeom_airy_direct()` otherwise. In the latter case, we first use hardware double precision arithmetic to determine an accurate estimate of the working precision needed to compute the Airy functions accurately for given z . This estimate is obtained by comparing the leading-order asymptotic estimate of the Airy functions with the magnitude of the largest term in the power series. The estimate is generic in the sense that it does not take into account vanishing near the roots of the functions. We subsequently evaluate the power series at the midpoint of z and bound the propagated error using derivatives. Derivatives are bounded using `acb_hypgeom_airy_bound()`.

```
void acb_hypgeom_airy_jet(acb_ptr ai, acb_ptr bi, const acb_t z, slong len, slong prec)
```

Writes to *ai* and *bi* the respective Taylor expansions of the Airy functions at the point z , truncated to length *len*. Either of the outputs can be `NULL` to avoid computing that function. The variable z is not allowed to be aliased with the outputs. To simplify the implementation, this method does not compute the series expansions of the primed versions directly; these are easily obtained by computing one extra coefficient and differentiating the output with `_acb_poly_derivative()`.

```
void _acb_hypgeom_airy_series(acb_ptr ai, acb_ptr ai_prime, acb_ptr bi, acb_ptr bi_prime, acb_srcptr z, slong zlen, slong len, slong prec)
```

```
void acb_hypgeom_airy_series(acb_poly_t ai, acb_poly_t ai_prime, acb_poly_t bi, acb_poly_t bi_prime, const acb_poly_t z, slong len, slong prec)
```

Computes the Airy functions evaluated at the power series z , truncated to length *len*. As with the other Airy methods, any of the outputs can be `NULL`.

9.17.11 Coulomb wave functions

Coulomb wave functions are solutions of the Coulomb wave equation

$$y'' + \left(1 - \frac{2\eta}{z} - \frac{\ell(\ell+1)}{z^2}\right)y = 0$$

which is the radial Schrödinger equation for a charged particle in a Coulomb potential $1/z$, where ℓ is the orbital angular momentum and η is the Sommerfeld parameter. The standard solutions are named $F_\ell(\eta, z)$ (regular at the origin $z = 0$) and $G_\ell(\eta, z)$ (irregular at the origin). The irregular solutions $H_\ell^\pm(\eta, z) = G_\ell(\eta, z) \pm iF_\ell(\eta, z)$ are also used.

Coulomb wave functions are special cases of confluent hypergeometric functions. The normalization constants and connection formulas are discussed in [DYF1999], [Gas2018], [Mic2007] and chapter 33 in [NIST2012]. In this implementation, we define the analytic continuations of all the functions so that the branch cut with respect to z is placed on the negative real axis. Precise definitions are given in http://fungrim.org/topic/Coulomb_wave_functions/

The following methods optionally compute $F_\ell(\eta, z)$, $G_\ell(\eta, z)$, $H_\ell^+(\eta, z)$, $H_\ell^-(\eta, z)$ simultaneously. Any of the four function values can be omitted by passing `NULL` for the unwanted output variables. The redundant functions $H^\pm$ are provided explicitly since taking the linear combination of F and G suffers from cancellation in parts of the complex plane.

```
void acb_hypgeom_coulomb(acb_t F, acb_t G, acb_t Hpos, acb_t Hneg, const acb_t l, const acb_t
    eta, const acb_t z, slong prec)
```

Writes to F , G , $Hpos$, $Hneg$ the values of the respective Coulomb wave functions. Any of the outputs can be *NULL*.

```
void acb_hypgeom_coulomb_jet(acb_ptr F, acb_ptr G, acb_ptr Hpos, acb_ptr Hneg, const acb_t l,
    const acb_t eta, const acb_t z, slong len, slong prec)
```

Writes to F , G , $Hpos$, $Hneg$ the respective Taylor expansions of the Coulomb wave functions at the point z , truncated to length len . Any of the outputs can be *NULL*.

```
void _acb_hypgeom_coulomb_series(acb_ptr F, acb_ptr G, acb_ptr Hpos, acb_ptr Hneg, const
    acb_t l, const acb_t eta, acb_srcptr z, slong zlen, slong len,
    slong prec)
```

```
void acb_hypgeom_coulomb_series(acb_poly_t F, acb_poly_t G, acb_poly_t Hpos, acb_poly_t
    Hneg, const acb_t l, const acb_t eta, const acb_poly_t z, slong
    len, slong prec)
```

Computes the Coulomb wave functions evaluated at the power series z , truncated to length len . Any of the outputs can be *NULL*.

9.17.12 Incomplete gamma and beta functions

```
void acb_hypgeom_gamma_upper_asymp(acb_t res, const acb_t s, const acb_t z, int regularized, slong
    prec)
```

```
void acb_hypgeom_gamma_upper_1f1a(acb_t res, const acb_t s, const acb_t z, int regularized, slong
    prec)
```

```
void acb_hypgeom_gamma_upper_1f1b(acb_t res, const acb_t s, const acb_t z, int regularized, slong
    prec)
```

```
void acb_hypgeom_gamma_upper_singular(acb_t res, slong s, const acb_t z, int regularized, slong
    prec)
```

```
void acb_hypgeom_gamma_upper(acb_t res, const acb_t s, const acb_t z, int regularized, slong prec)
```

If *regularized* is 0, computes the upper incomplete gamma function $\Gamma(s, z)$.

If *regularized* is 1, computes the regularized upper incomplete gamma function $Q(s, z) = \Gamma(s, z)/\Gamma(s)$.

If *regularized* is 2, computes the generalized exponential integral $z^{-s}\Gamma(s, z) = E_{1-s}(z)$ instead (this option is mainly intended for internal use; *acb_hypgeom_expint()* is the intended interface for computing the exponential integral).

The different methods respectively implement the formulas

$$\Gamma(s, z) = e^{-z}U(1-s, 1-s, z)$$

$$\Gamma(s, z) = \Gamma(s) - \frac{z^s}{s} {}_1F_1(s, s+1, -z)$$

$$\Gamma(s, z) = \Gamma(s) - \frac{z^s e^{-z}}{s} {}_1F_1(1, s+1, z)$$

$$\Gamma(s, z) = \frac{(-1)^n}{n!}(\psi(n+1) - \log(z)) + \frac{(-1)^n}{(n+1)!} z {}_2F_2(1, 1, 2, 2+n, -z) - z^{-n} \sum_{k=0}^{n-1} \frac{(-z)^k}{(k-n)k!}, \quad n = -s \in \mathbb{Z}_{\geq 0}$$

and an automatic algorithm choice. The automatic version also handles other special input such as $z = 0$ and $s = 1, 2, 3$. The *singular* version evaluates the finite sum directly and therefore assumes that s is not too large.


```
void _acb_hypgeom_gamma_upper_series(acb_ptr res, const acb_t s, acb_srcptr z, slong zlen, int
                                     regularized, slong n, slong prec)
```

```
void acb_hypgeom_gamma_upper_series(acb_poly_t res, const acb_t s, const acb_poly_t z, int
                                     regularized, slong n, slong prec)
```

Sets *res* to an upper incomplete gamma function where *s* is a constant and *z* is a power series, truncated to length *n*. The *regularized* argument has the same interpretation as in [acb_hypgeom_gamma_upper\(\)](#).

```
void acb_hypgeom_gamma_lower(acb_t res, const acb_t s, const acb_t z, int regularized, slong prec)
```

If *regularized* is 0, computes the lower incomplete gamma function $\gamma(s, z) = \frac{z^s}{s} {}_1F_1(s, s+1, -z)$.

If *regularized* is 1, computes the regularized lower incomplete gamma function $P(s, z) = \gamma(s, z)/\Gamma(s)$.

If *regularized* is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s, z) = z^{-s}P(s, z)$.

```
void _acb_hypgeom_gamma_lower_series(acb_ptr res, const acb_t s, acb_srcptr z, slong zlen, int
                                     regularized, slong n, slong prec)
```

```
void acb_hypgeom_gamma_lower_series(acb_poly_t res, const acb_t s, const acb_poly_t z, int
                                     regularized, slong n, slong prec)
```

Sets *res* to an lower incomplete gamma function where *s* is a constant and *z* is a power series, truncated to length *n*. The *regularized* argument has the same interpretation as in [acb_hypgeom_gamma_lower\(\)](#).

```
void acb_hypgeom_beta_lower(acb_t res, const acb_t a, const acb_t b, const acb_t z, int regularized,
                             slong prec)
```

Computes the (lower) incomplete beta function, defined by $B(a, b; z) = \int_0^z t^{a-1}(1-t)^{b-1}$, optionally the regularized incomplete beta function $I(a, b; z) = B(a, b; z)/B(a, b; 1)$.

In general, the integral must be interpreted using analytic continuation. The precise definitions for all parameter values are

$$B(a, b; z) = \frac{z^a}{a} {}_2F_1(a, 1-b, a+1, z)$$

$$I(a, b; z) = \frac{\Gamma(a+b)}{\Gamma(b)} z^a {}_2\tilde{F}_1(a, 1-b, a+1, z).$$

Note that both functions with this definition are undefined for nonpositive integer *a*, and *I* is undefined for nonpositive integer *a* + *b*.

```
void _acb_hypgeom_beta_lower_series(acb_ptr res, const acb_t a, const acb_t b, acb_srcptr z,
                                    slong zlen, int regularized, slong n, slong prec)
```

```
void acb_hypgeom_beta_lower_series(acb_poly_t res, const acb_t a, const acb_t b, const
                                    acb_poly_t z, int regularized, slong n, slong prec)
```

Sets *res* to the lower incomplete beta function $B(a, b; z)$ (optionally the regularized version $I(a, b; z)$) where *a* and *b* are constants and *z* is a power series, truncating the result to length *n*. The underscore method requires positive lengths and does not support aliasing.

9.17.13 Exponential and trigonometric integrals

The branch cut conventions of the following functions match Mathematica.

void **acb_hypgeom_expint**(*acb_t* res, const *acb_t* s, const *acb_t* z, *slong* prec)

Computes the generalized exponential integral $E_s(z)$. This is a trivial wrapper of *acb_hypgeom_gamma_upper()*.

void **acb_hypgeom_ei_asymp**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_ei_2f2**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_ei**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the exponential integral $Ei(z)$, respectively using

$$Ei(z) = -e^z U(1, 1, -z) - \log(-z) + \frac{1}{2} \left(\log(z) - \log\left(\frac{1}{z}\right) \right)$$

$$Ei(z) = z {}_2F_2(1, 1; 2, 2; z) + \gamma + \frac{1}{2} \left(\log(z) - \log\left(\frac{1}{z}\right) \right)$$

and an automatic algorithm choice.

void **_acb_hypgeom_ei_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_hypgeom_ei_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)

Computes the exponential integral of the power series z , truncated to length len .

void **acb_hypgeom_si_asymp**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_si_1f2**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_si**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the sine integral $Si(z)$, respectively using

$$Si(z) = \frac{i}{2} [e^{iz} U(1, 1, -iz) - e^{-iz} U(1, 1, iz) + \log(-iz) - \log(iz)]$$

$$Si(z) = z {}_1F_2\left(\frac{1}{2}; \frac{3}{2}, \frac{3}{2}; -\frac{z^2}{4}\right)$$

and an automatic algorithm choice.

void **_acb_hypgeom_si_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_hypgeom_si_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)

Computes the sine integral of the power series z , truncated to length len .

void **acb_hypgeom_ci_asymp**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_ci_2f3**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_ci**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the cosine integral $Ci(z)$, respectively using

$$Ci(z) = \log(z) - \frac{1}{2} [e^{iz} U(1, 1, -iz) + e^{-iz} U(1, 1, iz) + \log(-iz) + \log(iz)]$$

$$Ci(z) = -\frac{z^2}{4} {}_2F_3(1, 1; 2, 2, \frac{3}{2}; -\frac{z^2}{4}) + \log(z) + \gamma$$

and an automatic algorithm choice.

void **_acb_hypgeom_ci_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_hypgeom_ci_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)

Computes the cosine integral of the power series z , truncated to length len .

void **acb_hypgeom_shi**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the hyperbolic sine integral $\text{Shi}(z) = -i\text{Si}(iz)$. This is a trivial wrapper of *acb_hypgeom_si*().

void **_acb_hypgeom_shi_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_hypgeom_shi_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)

Computes the hyperbolic sine integral of the power series z, truncated to length len.

void **acb_hypgeom_chi_asymp**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_chi_2f3**(*acb_t* res, const *acb_t* z, *slong* prec)

void **acb_hypgeom_chi**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the hyperbolic cosine integral $\text{Chi}(z)$, respectively using

$$\text{Chi}(z) = -\frac{1}{2} [e^z U(1, 1, -z) + e^{-z} U(1, 1, z) + \log(-z) - \log(z)]$$

$$\text{Chi}(z) = \frac{z^2}{4} {}_2F_3(1, 1; 2, 2, \frac{3}{2}; \frac{z^2}{4}) + \log(z) + \gamma$$

and an automatic algorithm choice.

void **_acb_hypgeom_chi_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **acb_hypgeom_chi_series**(*acb_poly_t* res, const *acb_poly_t* z, *slong* len, *slong* prec)

Computes the hyperbolic cosine integral of the power series z, truncated to length len.

void **acb_hypgeom_li**(*acb_t* res, const *acb_t* z, int offset, *slong* prec)

If *offset* is zero, computes the logarithmic integral $\text{li}(z) = \text{Ei}(\log(z))$.

If *offset* is nonzero, computes the offset logarithmic integral $\text{Li}(z) = \text{li}(z) - \text{li}(2)$.

void **_acb_hypgeom_li_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, int offset, *slong* len, *slong* prec)

void **acb_hypgeom_li_series**(*acb_poly_t* res, const *acb_poly_t* z, int offset, *slong* len, *slong* prec)

Computes the logarithmic integral (optionally the offset version) of the power series z, truncated to length len.

9.17.14 Gauss hypergeometric function

The following methods compute the Gauss hypergeometric function

$$F(z) = {}_2F_1(a, b, c, z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!}$$

or the regularized version $\mathbf{F}(z) = \mathbf{F}(a, b, c, z) = {}_2F_1(a, b, c, z)/\Gamma(c)$ if the flag *regularized* is set.

void **acb_hypgeom_2f1_continuation**(*acb_t* res0, *acb_t* res1, const *acb_t* a, const *acb_t* b, const *acb_t* c, const *acb_t* z0, const *acb_t* z1, const *acb_t* f0, const *acb_t* f1, *slong* prec)

Given $F(z_0), F'(z_0)$ in *f0, f1*, sets *res0* and *res1* to $F(z_1), F'(z_1)$ by integrating the hypergeometric differential equation along a straight-line path. The evaluation points should be well-isolated from the singular points 0 and 1.

void **acb_hypgeom_2f1_series_direct**(*acb_poly_t* res, const *acb_poly_t* a, const *acb_poly_t* b, const *acb_poly_t* c, const *acb_poly_t* z, int regularized, *slong* len, *slong* prec)

Computes $F(z)$ of the given power series truncated to length len, using direct summation of the hypergeometric series.

```
void acb_hypgeom_2f1_direct(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z,
                           int regularized, slong prec)
```

Computes $F(z)$ using direct summation of the hypergeometric series.

```
void acb_hypgeom_2f1_transform(acb_t res, const acb_t a, const acb_t b, const acb_t c, const
                               acb_t z, int flags, int which, slong prec)
```

```
void acb_hypgeom_2f1_transform_limit(acb_t res, const acb_t a, const acb_t b, const acb_t c,
                                     const acb_t z, int regularized, int which, slong prec)
```

Computes $F(z)$ using an argument transformation determined by the flag *which*. Legal values are 1 for $z/(z-1)$, 2 for $1/z$, 3 for $1/(1-z)$, 4 for $1-z$, and 5 for $1-1/z$.

The *transform_limit* version assumes that *which* is not 1. If *which* is 2 or 3, it assumes that $b-a$ represents an exact integer. If *which* is 4 or 5, it assumes that $c-a-b$ represents an exact integer. In these cases, it computes the correct limit value.

See [acb_hypgeom_2f1\(\)](#) for the meaning of *flags*.

```
void acb_hypgeom_2f1_corner(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z,
                            int regularized, slong prec)
```

Computes $F(z)$ near the corner cases $\exp(\pm\pi i\sqrt{3})$ by analytic continuation.

```
int acb_hypgeom_2f1_choose(const acb_t z)
```

Chooses a method to compute the function based on the location of z in the complex plane. If the return value is 0, direct summation should be used. If the return value is 1 to 5, the transformation with this index in [acb_hypgeom_2f1_transform\(\)](#) should be used. If the return value is 6, the corner case algorithm should be used.

```
void acb_hypgeom_2f1(acb_t res, const acb_t a, const acb_t b, const acb_t c, const acb_t z, int flags,
                     slong prec)
```

Computes $F(z)$ or $\mathbf{F}(z)$ using an automatic algorithm choice.

The following bit fields can be set in *flags*:

- `ACB_HYPGEOM_2F1_REGULARIZED` - computes the regularized hypergeometric function $\mathbf{F}(z)$. Setting *flags* to 1 is the same as just toggling this option.
- `ACB_HYPGEOM_2F1_AB` - $a-b$ is an integer.
- `ACB_HYPGEOM_2F1_ABC` - $a+b-c$ is an integer.
- `ACB_HYPGEOM_2F1_AC` - $a-c$ is an integer.
- `ACB_HYPGEOM_2F1_BC` - $b-c$ is an integer.

The last four flags can be set to indicate that the respective parameter differences are known to represent exact integers, even if the input intervals are inexact. This allows the correct limits to be evaluated when applying transformation formulas. For example, to evaluate ${}_2F_1(\sqrt{2}, 1/2, \sqrt{2} + 3/2, 9/10)$, the *ABC* flag should be set. If not set, the result will be an indeterminate interval due to internally dividing by an interval containing zero. If the parameters are exact floating-point numbers (including exact integers or half-integers), then the limits are computed automatically, and setting these flags is unnecessary.

Currently, only the *AB* and *ABC* flags are used this way; the *AC* and *BC* flags might be used in the future.

9.17.15 Orthogonal polynomials and functions

void **acb_hypgeom_chebyshev_t**(*acb_t* res, const *acb_t* n, const *acb_t* z, *slong* prec)

void **acb_hypgeom_chebyshev_u**(*acb_t* res, const *acb_t* n, const *acb_t* z, *slong* prec)

Computes the Chebyshev polynomial (or Chebyshev function) of first or second kind

$$T_n(z) = {}_2F_1\left(-n, n, \frac{1}{2}, \frac{1-z}{2}\right)$$

$$U_n(z) = (n+1) {}_2F_1\left(-n, n+2, \frac{3}{2}, \frac{1-z}{2}\right).$$

The hypergeometric series definitions are only used for computation near the point 1. In general, trigonometric representations are used. For word-size integer n , **acb_chebyshev_t_ui()** and **acb_chebyshev_u_ui()** are called.

void **acb_hypgeom_jacobi_p**(*acb_t* res, const *acb_t* n, const *acb_t* a, const *acb_t* b, const *acb_t* z, *slong* prec)

Computes the Jacobi polynomial (or Jacobi function)

$$P_n^{(a,b)}(z) = \frac{(a+1)_n}{\Gamma(n+1)} {}_2F_1\left(-n, n+a+b+1, a+1, \frac{1-z}{2}\right).$$

For nonnegative integer n , this is a polynomial in a , b and z , even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void **acb_hypgeom_gegenbauer_c**(*acb_t* res, const *acb_t* n, const *acb_t* m, const *acb_t* z, *slong* prec)

Computes the Gegenbauer polynomial (or Gegenbauer function)

$$C_n^m(z) = \frac{(2m)_n}{\Gamma(n+1)} {}_2F_1\left(-n, 2m+n, m+\frac{1}{2}, \frac{1-z}{2}\right).$$

For nonnegative integer n , this is a polynomial in m and z , even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

void **acb_hypgeom_laguerre_l**(*acb_t* res, const *acb_t* n, const *acb_t* m, const *acb_t* z, *slong* prec)

Computes the Laguerre polynomial (or Laguerre function)

$$L_n^m(z) = \frac{(m+1)_n}{\Gamma(n+1)} {}_1F_1(-n, m+1, z).$$

For nonnegative integer n , this is a polynomial in m and z , even when the parameters are such that the hypergeometric series is undefined. In such cases, the polynomial is evaluated using direct methods.

There are at least two incompatible ways to define the Laguerre function when n is a negative integer. One possibility when $m = 0$ is to define $L_{-n}^0(z) = e^z L_{n-1}^0(-z)$. Another possibility is to cover this case with the recurrence relation $L_{n-1}^m(z) + L_n^{m-1}(z) = L_n^m(z)$. Currently, we leave this case undefined (returning indeterminate).

void **acb_hypgeom_hermite_h**(*acb_t* res, const *acb_t* n, const *acb_t* z, *slong* prec)

Computes the Hermite polynomial (or Hermite function)

$$H_n(z) = 2^n \sqrt{\pi} \left(\frac{1}{\Gamma((1-n)/2)} {}_1F_1\left(-\frac{n}{2}, \frac{1}{2}, z^2\right) - \frac{2z}{\Gamma(-n/2)} {}_1F_1\left(\frac{1-n}{2}, \frac{3}{2}, z^2\right) \right).$$

void **acb_hypgeom_legendre_p**(*acb_t* res, const *acb_t* n, const *acb_t* m, const *acb_t* z, int type, *slong* prec)

Sets *res* to the associated Legendre function of the first kind evaluated for degree *n*, order *m*, and argument *z*. When *m* is zero, this reduces to the Legendre polynomial $P_n(z)$.

Many different branch cut conventions appear in the literature. If *type* is 0, the version

$$P_n^m(z) = \frac{(1+z)^{m/2}}{(1-z)^{m/2}} \mathbf{F} \left(-n, n+1, 1-m, \frac{1-z}{2} \right)$$

is computed, and if *type* is 1, the alternative version

$$\mathcal{P}_n^m(z) = \frac{(z+1)^{m/2}}{(z-1)^{m/2}} \mathbf{F} \left(-n, n+1, 1-m, \frac{1-z}{2} \right).$$

is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in *Mathematica* and *mpmath*.

void **acb_hypgeom_legendre_q**(*acb_t* res, const *acb_t* n, const *acb_t* m, const *acb_t* z, int type, *slong* prec)

Sets *res* to the associated Legendre function of the second kind evaluated for degree *n*, order *m*, and argument *z*. When *m* is zero, this reduces to the Legendre function $Q_n(z)$.

Many different branch cut conventions appear in the literature. If *type* is 0, the version

$$Q_n^m(z) = \frac{\pi}{2 \sin(\pi m)} \left(\cos(\pi m) P_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} P_n^{-m}(z) \right)$$

is computed, and if *type* is 1, the alternative version

$$\mathcal{Q}_n^m(z) = \frac{\pi}{2 \sin(\pi m)} e^{\pi i m} \left(\mathcal{P}_n^m(z) - \frac{\Gamma(1+m+n)}{\Gamma(1-m+n)} \mathcal{P}_n^{-m}(z) \right)$$

is computed. Type 0 and type 1 respectively correspond to type 2 and type 3 in *Mathematica* and *mpmath*.

When *m* is an integer, either expression is interpreted as a limit. We make use of the connection formulas [WQ3a], [WQ3b] and [WQ3c] to allow computing the function even in the limiting case. (The formula [WQ3d] would be useful, but is incorrect in the lower half plane.)

void **acb_hypgeom_legendre_p_uiui_rec**(*acb_t* res, *ulong* n, *ulong* m, const *acb_t* z, *slong* prec)

For nonnegative integer *n* and *m*, uses recurrence relations to evaluate $(1-z^2)^{-m/2} P_n^m(z)$ which is a polynomial in *z*.

void **acb_hypgeom_spherical_y**(*acb_t* res, *slong* n, *slong* m, const *acb_t* theta, const *acb_t* phi, *slong* prec)

Computes the spherical harmonic of degree *n*, order *m*, latitude angle *theta*, and longitude angle *phi*, normalized such that

$$Y_n^m(\theta, \phi) = \sqrt{\frac{2n+1}{4\pi} \frac{(n-m)!}{(n+m)!}} e^{im\phi} P_n^m(\cos(\theta)).$$

The definition is extended to negative *m* and *n* by symmetry. This function is a polynomial in $\cos(\theta)$ and $\sin(\theta)$. We evaluate it using `acb_hypgeom_legendre_p_uiui_rec()`.

9.17.16 Dilogarithm

The dilogarithm function is given by $\text{Li}_2(z) = -\int_0^z \frac{\log(1-t)}{t} dt = {}_3F_2(1, 1, 1, 2, 2, z)$.

void **acb_hypgeom_dilog_bernoulli**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the dilogarithm using a series expansion in $w = \log(z)$, with rate of convergence $|w/(2\pi)|^n$. This provides good convergence near $z = e^{\pm i\pi/3}$, where hypergeometric series expansions fail. Since the coefficients involve Bernoulli numbers, this method should only be used at moderate precision.

void **acb_hypgeom_dilog_zero_taylor**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the dilogarithm for z close to 0 using the hypergeometric series (effective only when $|z| \ll 1$).

void **acb_hypgeom_dilog_zero**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the dilogarithm for z close to 0, using the bit-burst algorithm instead of the hypergeometric series directly at very high precision.

void **acb_hypgeom_dilog_transform**(*acb_t* res, const *acb_t* z, int algorithm, *slong* prec)

Computes the dilogarithm by applying one of the transformations $1/z$, $1 - z$, $z/(z - 1)$, $1/(1 - z)$, indexed by *algorithm* from 1 to 4, and calling **acb_hypgeom_dilog_zero()** with the reduced variable. Alternatively, for *algorithm* between 5 and 7, starts from the respective point $\pm i$, $(1 \pm i)/2$, $(1 \pm i)/2$ (with the sign chosen according to the midpoint of z) and computes the dilogarithm by the bit-burst method.

void **acb_hypgeom_dilog_continuation**(*acb_t* res, const *acb_t* a, const *acb_t* z, *slong* prec)

Computes $\text{Li}_2(z) - \text{Li}_2(a)$ using Taylor expansion at a . Binary splitting is used. Both a and z should be well isolated from the points 0 and 1, except that a may be exactly 0. If the straight line path from a to b crosses the branch cut, this method provides continuous analytic continuation instead of computing the principal branch.

void **acb_hypgeom_dilog_bitburst**(*acb_t* res, *acb_t* z0, const *acb_t* z, *slong* prec)

Sets $z0$ to a point with short bit expansion close to z and sets *res* to $\text{Li}_2(z) - \text{Li}_2(z_0)$, computed using the bit-burst algorithm.

void **acb_hypgeom_dilog**(*acb_t* res, const *acb_t* z, *slong* prec)

Computes the dilogarithm using a default algorithm choice.

9.18 arb_hypgeom.h – hypergeometric functions of real variables

See *acb_hypgeom.h – hypergeometric functions of complex variables* for the general implementation of hypergeometric functions.

For convenience, this module provides versions of the same functions for real variables represented using *arb_t* and *arb_poly_t*. Most methods are simple wrappers around the complex versions, but some of the functions in this module have been further optimized specifically for real variables.

This module also provides certain functions exclusive to real variables, such as functions for computing real roots of common special functions.

9.18.1 Rising factorials

```
void _arb_hypgeom_rising_coeffs_1(ulong *c, ulong k, slong n)
```

```
void _arb_hypgeom_rising_coeffs_2(ulong *c, ulong k, slong n)
```

```
void _arb_hypgeom_rising_coeffs_fmpz(fmpz *c, ulong k, slong n)
```

Sets c to the coefficients of the rising factorial polynomial $(X + k)_n$. The 1 and 2 versions respectively compute single-word and double-word coefficients, without checking for overflow, while the *fmpz* version allows arbitrarily large coefficients. These functions are mostly intended for internal use; the *fmpz* version does not use an asymptotically fast algorithm. The degree n must be at least 2.

```
void arb_hypgeom_rising_ui_forward(arb_t res, const arb_t x, ulong n, slong prec)
```

```
void arb_hypgeom_rising_ui_bs(arb_t res, const arb_t x, ulong n, slong prec)
```

```
void arb_hypgeom_rising_ui_rs(arb_t res, const arb_t x, ulong n, ulong m, slong prec)
```

```
void arb_hypgeom_rising_ui_rec(arb_t res, const arb_t x, ulong n, slong prec)
```

```
void arb_hypgeom_rising_ui(arb_t res, const arb_t x, ulong n, slong prec)
```

```
void arb_hypgeom_rising(arb_t res, const arb_t x, const arb_t n, slong prec)
```

Computes the rising factorial $(x)_n$.

The *forward* version uses the forward recurrence. The *bs* version uses binary splitting. The *rs* version uses rectangular splitting. It takes an extra tuning parameter m which can be set to zero to choose automatically. The *rec* version chooses an algorithm automatically, avoiding use of the gamma function (so that it can be used in the computation of the gamma function). The default versions (*rising_ui* and *rising*) choose an algorithm automatically and may additionally fall back on the gamma function.

```
void arb_hypgeom_rising_ui_jet_powsum(arb_ptr res, const arb_t x, ulong n, slong len, slong prec)
```

```
void arb_hypgeom_rising_ui_jet_bs(arb_ptr res, const arb_t x, ulong n, slong len, slong prec)
```

```
void arb_hypgeom_rising_ui_jet_rs(arb_ptr res, const arb_t x, ulong n, ulong m, slong len, slong prec)
```

```
void arb_hypgeom_rising_ui_jet(arb_ptr res, const arb_t x, ulong n, slong len, slong prec)
```

Computes the jet of the rising factorial $(x)_n$, truncated to length len . In other words, constructs the polynomial $(X + x)_n \in \mathbb{R}[X]$, truncated if $len < n + 1$ (and zero-extended if $len > n + 1$).

The *powsum* version computes the sequence of powers of x and forms integral linear combinations of these. The *bs* version uses binary splitting. The *rs* version uses rectangular splitting. It takes an extra tuning parameter m which can be set to zero to choose automatically. The default version chooses an algorithm automatically.

9.18.2 Gamma function

```
void _arb_hypgeom_gamma_stirling_term_bounds(slong *bound, const mag_t zinv, slong N)
```

For $1 \leq n < N$, sets *bound* to an exponent bounding the n -th term in the Stirling series for the gamma function, given a precomputed upper bound for $|z|^{-1}$. This function is intended for internal use and does not check for underflow or underflow in the exponents.

```
void arb_hypgeom_gamma_stirling_sum_horner(arb_t res, const arb_t z, slong N, slong prec)
```

```
void arb_hypgeom_gamma_stirling_sum_improved(arb_t res, const arb_t z, slong N, slong K, slong prec)
```

Sets *res* to the final sum in the Stirling series for the gamma function truncated before the term with index N , i.e. computes $\sum_{n=1}^{N-1} B_{2n}/(2n(2n-1)z^{2n-1})$. The *horner* version uses Horner scheme with gradual precision adjustments. The *improved* version uses rectangular splitting for the low-index terms and reexpands the high-index terms as hypergeometric polynomials, using a splitting parameter K (which can be set to 0 to use a default value).

void **arb_hypgeom_gamma_stirling**(*arb_t* res, const *arb_t* x, int reciprocal, *slong* prec)

Sets *res* to the gamma function of *x* computed using the Stirling series together with argument reduction. If *reciprocal* is set, the reciprocal gamma function is computed instead.

int **arb_hypgeom_gamma_taylor**(*arb_t* res, const *arb_t* x, int reciprocal, *slong* prec)

Attempts to compute the gamma function of *x* using Taylor series together with argument reduction. This is only supported if *x* and *prec* are both small enough. If successful, returns 1; otherwise, does nothing and returns 0. If *reciprocal* is set, the reciprocal gamma function is computed instead.

void **arb_hypgeom_gamma**(*arb_t* res, const *arb_t* x, *slong* prec)

void **arb_hypgeom_gamma_fmpq**(*arb_t* res, const *fmpq_t* x, *slong* prec)

void **arb_hypgeom_gamma_fmpz**(*arb_t* res, const *fmpz_t* x, *slong* prec)

Sets *res* to the gamma function of *x* computed using a default algorithm choice.

void **arb_hypgeom_rgamma**(*arb_t* res, const *arb_t* x, *slong* prec)

Sets *res* to the reciprocal gamma function of *x* computed using a default algorithm choice.

void **arb_hypgeom_lgamma**(*arb_t* res, const *arb_t* x, *slong* prec)

Sets *res* to the log-gamma function of *x* computed using a default algorithm choice.

9.18.3 Binomial coefficients

void **arb_hypgeom_central_bin_ui**(*arb_t* res, *ulong* n, *slong* prec)

Computes the central binomial coefficient $\binom{2n}{n}$.

9.18.4 Generalized hypergeometric function

void **arb_hypgeom_pfq**(*arb_t* res, *arb_srcptr* a, *slong* p, *arb_srcptr* b, *slong* q, const *arb_t* z, int regularized, *slong* prec)

Computes the generalized hypergeometric function ${}_pF_q(z)$, or the regularized version if *regularized* is set.

9.18.5 Confluent hypergeometric functions

void **arb_hypgeom_0f1**(*arb_t* res, const *arb_t* a, const *arb_t* z, int regularized, *slong* prec)

Computes the confluent hypergeometric limit function ${}_0F_1(a, z)$, or $\frac{1}{\Gamma(a)}{}_0F_1(a, z)$ if *regularized* is set.

void **arb_hypgeom_m**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* z, int regularized, *slong* prec)

Computes the confluent hypergeometric function $M(a, b, z) = {}_1F_1(a, b, z)$, or $\mathbf{M}(a, b, z) = \frac{1}{\Gamma(b)}{}_1F_1(a, b, z)$ if *regularized* is set.

void **arb_hypgeom_1f1**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* z, int regularized, *slong* prec)

Alias for *arb_hypgeom_m()*.

void **arb_hypgeom_1f1_integration**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* z, int regularized, *slong* prec)

Computes the confluent hypergeometric function using numerical integration of the representation

$${}_1F_1(a, b, z) = \frac{\Gamma(b)}{\Gamma(a)\Gamma(b-a)} \int_0^1 e^{zt} t^{a-1} (1-t)^{b-a-1} dt.$$

This algorithm can be useful if the parameters are large. This will currently only return a finite enclosure if $a \geq 1$ and $b - a \geq 1$.

void **arb_hypgeom_u**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* z, *slong* prec)

Computes the confluent hypergeometric function $U(a, b, z)$.

void **arb_hypgeom_u_integration**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* z, *slong* prec)

Computes the confluent hypergeometric function $U(a, b, z)$ using numerical integration of the representation

$$U(a, b, z) = \frac{1}{\Gamma(a)} \int_0^\infty e^{-zt} t^{a-1} (1+t)^{b-a-1} dt.$$

This algorithm can be useful if the parameters are large. This will currently only return a finite enclosure if $a \geq 1$ and $z > 0$.

9.18.6 Gauss hypergeometric function

void **arb_hypgeom_2f1**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* c, const *arb_t* z, int regularized, *slong* prec)

Computes the Gauss hypergeometric function ${}_2F_1(a, b, c, z)$, or $\mathbf{F}(a, b, c, z) = \frac{1}{\Gamma(c)} {}_2F_1(a, b, c, z)$ if *regularized* is set.

Additional evaluation flags can be passed via the *regularized* argument; see [acb_hypgeom_2f1\(\)](#) for documentation.

void **arb_hypgeom_2f1_integration**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* c, const *arb_t* z, int regularized, *slong* prec)

Computes the Gauss hypergeometric function using numerical integration of the representation

$${}_2F_1(a, b, c, z) = \frac{\Gamma(a)}{\Gamma(b)\Gamma(c-b)} \int_0^1 t^{b-1} (1-t)^{c-b-1} (1-zt)^{-a} dt.$$

This algorithm can be useful if the parameters are large. This will currently only return a finite enclosure if $b \geq 1$ and $c-b \geq 1$ and $z < 1$, possibly with a and b exchanged.

9.18.7 Error functions and Fresnel integrals

void **arb_hypgeom_erf**(*arb_t* res, const *arb_t* z, *slong* prec)

Computes the error function $\operatorname{erf}(z)$.

void **_arb_hypgeom_erf_series**(*arb_ptr* res, *arb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **arb_hypgeom_erf_series**(*arb_poly_t* res, const *arb_poly_t* z, *slong* len, *slong* prec)

Computes the error function of the power series z , truncated to length len .

void **arb_hypgeom_erfc**(*arb_t* res, const *arb_t* z, *slong* prec)

Computes the complementary error function $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. This function avoids catastrophic cancellation for large positive z .

void **_arb_hypgeom_erfc_series**(*arb_ptr* res, *arb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **arb_hypgeom_erfc_series**(*arb_poly_t* res, const *arb_poly_t* z, *slong* len, *slong* prec)

Computes the complementary error function of the power series z , truncated to length len .

void **arb_hypgeom_erfi**(*arb_t* res, const *arb_t* z, *slong* prec)

Computes the imaginary error function $\operatorname{erfi}(z) = -i \operatorname{erf}(iz)$.

void **_arb_hypgeom_erfi_series**(*arb_ptr* res, *arb_srcptr* z, *slong* zlen, *slong* len, *slong* prec)

void **arb_hypgeom_erfi_series**(*arb_poly_t* res, const *arb_poly_t* z, *slong* len, *slong* prec)

Computes the imaginary error function of the power series z , truncated to length len .

void **arb_hypgeom_erfinv**(*arb_t* res, const *arb_t* z, *slong* prec)

void **arb_hypgeom_erfcinv**(*arb_t* res, const *arb_t* z, *slong* prec)

Computes the inverse error function $\operatorname{erf}^{-1}(z)$ or inverse complementary error function $\operatorname{erfc}^{-1}(z)$.

void **arb_hypgeom_fresnel**(*arb_t* res1, *arb_t* res2, const *arb_t* z, int normalized, *slong* prec)

Sets *res1* to the Fresnel sine integral $S(z)$ and *res2* to the Fresnel cosine integral $C(z)$. Optionally, just a single function can be computed by passing *NULL* as the other output variable. The definition $S(z) = \int_0^z \sin(t^2)dt$ is used if *normalized* is 0, and $S(z) = \int_0^z \sin(\frac{1}{2}\pi t^2)dt$ is used if *normalized* is 1 (the latter is the Abramowitz & Stegun convention). $C(z)$ is defined analogously.

void **_arb_hypgeom_fresnel_series**(*arb_ptr* res1, *arb_ptr* res2, *arb_srcptr* z, *slong* zlen, int normalized, *slong* len, *slong* prec)

void **arb_hypgeom_fresnel_series**(*arb_poly_t* res1, *arb_poly_t* res2, const *arb_poly_t* z, int normalized, *slong* len, *slong* prec)

Sets *res1* to the Fresnel sine integral and *res2* to the Fresnel cosine integral of the power series *z*, truncated to length *len*. Optionally, just a single function can be computed by passing *NULL* as the other output variable.

9.18.8 Incomplete gamma and beta functions

void **arb_hypgeom_gamma_upper**(*arb_t* res, const *arb_t* s, const *arb_t* z, int regularized, *slong* prec)

If *regularized* is 0, computes the upper incomplete gamma function $\Gamma(s, z)$.

If *regularized* is 1, computes the regularized upper incomplete gamma function $Q(s, z) = \Gamma(s, z)/\Gamma(s)$.

If *regularized* is 2, computes the generalized exponential integral $z^{-s}\Gamma(s, z) = E_{1-s}(z)$ instead (this option is mainly intended for internal use; **arb_hypgeom_expint()** is the intended interface for computing the exponential integral).

void **arb_hypgeom_gamma_upper_integration**(*arb_t* res, const *arb_t* s, const *arb_t* z, int regularized, *slong* prec)

Computes the upper incomplete gamma function using numerical integration.

void **_arb_hypgeom_gamma_upper_series**(*arb_ptr* res, const *arb_t* s, *arb_srcptr* z, *slong* zlen, int regularized, *slong* n, *slong* prec)

void **arb_hypgeom_gamma_upper_series**(*arb_poly_t* res, const *arb_t* s, const *arb_poly_t* z, int regularized, *slong* n, *slong* prec)

Sets *res* to an upper incomplete gamma function where *s* is a constant and *z* is a power series, truncated to length *n*. The *regularized* argument has the same interpretation as in **arb_hypgeom_gamma_upper()**.

void **arb_hypgeom_gamma_lower**(*arb_t* res, const *arb_t* s, const *arb_t* z, int regularized, *slong* prec)

If *regularized* is 0, computes the lower incomplete gamma function $\gamma(s, z) = \frac{z^s}{s} {}_1F_1(s, s+1, -z)$.

If *regularized* is 1, computes the regularized lower incomplete gamma function $P(s, z) = \gamma(s, z)/\Gamma(s)$.

If *regularized* is 2, computes a further regularized lower incomplete gamma function $\gamma^*(s, z) = z^{-s}P(s, z)$.

void **_arb_hypgeom_gamma_lower_series**(*arb_ptr* res, const *arb_t* s, *arb_srcptr* z, *slong* zlen, int regularized, *slong* n, *slong* prec)

void **arb_hypgeom_gamma_lower_series**(*arb_poly_t* res, const *arb_t* s, const *arb_poly_t* z, int regularized, *slong* n, *slong* prec)

Sets *res* to an lower incomplete gamma function where *s* is a constant and *z* is a power series, truncated to length *n*. The *regularized* argument has the same interpretation as in **arb_hypgeom_gamma_lower()**.

void **arb_hypgeom_beta_lower**(*arb_t* res, const *arb_t* a, const *arb_t* b, const *arb_t* z, int regularized, *slong* prec)

Computes the (lower) incomplete beta function, defined by $B(a, b; z) = \int_0^z t^{a-1}(1-t)^{b-1}$, optionally the regularized incomplete beta function $I(a, b; z) = B(a, b; z)/B(a, b; 1)$.

void **_arb_hypgeom_beta_lower_series**(*arb_ptr* res, const *arb_t* a, const *arb_t* b, *arb_srcptr* z, *slong* zlen, int regularized, *slong* n, *slong* prec)

void **arb_hypgeom_beta_lower_series**(*arb_poly_t* res, const *arb_t* a, const *arb_t* b, const *arb_poly_t* z, int regularized, *slong* n, *slong* prec)

Sets *res* to the lower incomplete beta function $B(a, b; z)$ (optionally the regularized version $I(a, b; z)$) where a and b are constants and z is a power series, truncating the result to length n . The underscore method requires positive lengths and does not support aliasing.

Internal evaluation functions

void **_arb_hypgeom_gamma_lower_sum_rs_1**(*arb_t* res, *ulong* p, *ulong* q, const *arb_t* z, *slong* N, *slong* prec)

Computes $\sum_{k=0}^{N-1} z^k / (a)_k$ where $a = p/q$ using rectangular splitting. It is assumed that $p + qN$ fits in a limb.

void **_arb_hypgeom_gamma_upper_sum_rs_1**(*arb_t* res, *ulong* p, *ulong* q, const *arb_t* z, *slong* N, *slong* prec)

Computes $\sum_{k=0}^{N-1} (a)_k / z^k$ where $a = p/q$ using rectangular splitting. It is assumed that $p + qN$ fits in a limb.

slong **_arb_hypgeom_gamma_upper_fmpq_inf_choose_N**(*mag_t* err, const *fmpq_t* a, const *arb_t* z, const *mag_t* abs_tol)

Returns number of terms N and sets *err* to the truncation error for evaluating $\Gamma(a, z)$ using the asymptotic series at infinity, targeting an absolute tolerance of *abs_tol*. The error may be set to *err* if the tolerance cannot be achieved. Assumes that z is positive.

void **_arb_hypgeom_gamma_upper_fmpq_inf_bsplitt**(*arb_t* res, const *fmpq_t* a, const *arb_t* z, *slong* N, *slong* prec)

Sets *res* to the approximation of $\Gamma(a, z)$ obtained by truncating the asymptotic series at infinity before term N . The truncation error bound has to be added separately.

slong **_arb_hypgeom_gamma_lower_fmpq_0_choose_N**(*mag_t* err, const *fmpq_t* a, const *arb_t* z, const *mag_t* abs_tol)

Returns number of terms N and sets *err* to the truncation error for evaluating $\gamma(a, z)$ using the Taylor series at zero, targeting an absolute tolerance of *abs_tol*. Assumes that z is positive.

void **_arb_hypgeom_gamma_lower_fmpq_0_bsplitt**(*arb_t* res, const *fmpq_t* a, const *arb_t* z, *slong* N, *slong* prec)

Sets *res* to the approximation of $\gamma(a, z)$ obtained by truncating the Taylor series at zero before term N . The truncation error bound has to be added separately.

slong **_arb_hypgeom_gamma_upper_singular_si_choose_N**(*mag_t* err, *slong* n, const *arb_t* z, const *mag_t* abs_tol)

Returns number of terms N and sets *err* to the truncation error for evaluating $\Gamma(-n, z)$ using the Taylor series at zero, targeting an absolute tolerance of *abs_tol*.

void **_arb_hypgeom_gamma_upper_singular_si_bsplitt**(*arb_t* res, *slong* n, const *arb_t* z, *slong* N, *slong* prec)

Sets *res* to the approximation of $\Gamma(-n, z)$ obtained by truncating the Taylor series at zero before term N . The truncation error bound has to be added separately.

```
void _arb_gamma_upper_fmpq_step_bsplit(arb_t Gz1, const fmpq_t a, const arb_t z0, const arb_t
                                         z1, const arb_t Gz0, const arb_t expmz0, const mag_t
                                         abs_tol, slong prec)
```

Given $Gz0$ and $expmz0$ representing the values $\Gamma(a, z_0)$ and $\exp(-z_0)$, computes $\Gamma(a, z_1)$ using the Taylor series at z_0 evaluated using binary splitting, targeting an absolute error of abs_tol . Assumes that z_0 and z_1 are positive.

9.18.9 Exponential and trigonometric integrals

```
void arb_hypgeom_expint(arb_t res, const arb_t s, const arb_t z, slong prec)
```

Computes the generalized exponential integral $E_s(z)$.

```
void arb_hypgeom_ei(arb_t res, const arb_t z, slong prec)
```

Computes the exponential integral $Ei(z)$.

```
void _arb_hypgeom_ei_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

```
void arb_hypgeom_ei_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the exponential integral of the power series z , truncated to length len .

```
void _arb_hypgeom_si_asymp(arb_t res, const arb_t z, slong N, slong prec)
```

```
void _arb_hypgeom_si_1f2(arb_t res, const arb_t z, slong N, slong wp, slong prec)
```

```
void arb_hypgeom_si(arb_t res, const arb_t z, slong prec)
```

Computes the sine integral $Si(z)$.

```
void _arb_hypgeom_si_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

```
void arb_hypgeom_si_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the sine integral of the power series z , truncated to length len .

```
void _arb_hypgeom_ci_asymp(arb_t res, const arb_t z, slong N, slong prec)
```

```
void _arb_hypgeom_ci_2f3(arb_t res, const arb_t z, slong N, slong wp, slong prec)
```

```
void arb_hypgeom_ci(arb_t res, const arb_t z, slong prec)
```

Computes the cosine integral $Ci(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

```
void _arb_hypgeom_ci_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

```
void arb_hypgeom_ci_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the cosine integral of the power series z , truncated to length len .

```
void arb_hypgeom_shi(arb_t res, const arb_t z, slong prec)
```

Computes the hyperbolic sine integral $Shi(z) = -i Si(iz)$.

```
void _arb_hypgeom_shi_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

```
void arb_hypgeom_shi_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the hyperbolic sine integral of the power series z , truncated to length len .

```
void arb_hypgeom_chi(arb_t res, const arb_t z, slong prec)
```

Computes the hyperbolic cosine integral $Chi(z)$. The result is indeterminate if $z < 0$ since the value of the function would be complex.

```
void _arb_hypgeom_chi_series(arb_ptr res, arb_srcptr z, slong zlen, slong len, slong prec)
```

```
void arb_hypgeom_chi_series(arb_poly_t res, const arb_poly_t z, slong len, slong prec)
```

Computes the hyperbolic cosine integral of the power series z , truncated to length len .

```
void arb_hypgeom_li(arb_t res, const arb_t z, int offset, slong prec)
```

If *offset* is zero, computes the logarithmic integral $li(z) = Ei(\log(z))$.

If *offset* is nonzero, computes the offset logarithmic integral $Li(z) = li(z) - li(2)$.

The result is indeterminate if $z < 0$ since the value of the function would be complex.


```
void _arb_hypgeom_li_series(arb_ptr res, arb_srcptr z, slong zlen, int offset, slong len, slong prec)
void arb_hypgeom_li_series(arb_poly_t res, const arb_poly_t z, int offset, slong len, slong prec)
    Computes the logarithmic integral (optionally the offset version) of the power series  $z$ , truncated
    to length  $len$ .
```

9.18.10 Bessel functions

```
void arb_hypgeom_bessel_j(arb_t res, const arb_t nu, const arb_t z, slong prec)
    Computes the Bessel function of the first kind  $J_\nu(z)$ .
void arb_hypgeom_bessel_y(arb_t res, const arb_t nu, const arb_t z, slong prec)
    Computes the Bessel function of the second kind  $Y_\nu(z)$ .
void arb_hypgeom_bessel_jy(arb_t res1, arb_t res2, const arb_t nu, const arb_t z, slong prec)
    Sets  $res1$  to  $J_\nu(z)$  and  $res2$  to  $Y_\nu(z)$ , computed simultaneously.
void arb_hypgeom_bessel_i(arb_t res, const arb_t nu, const arb_t z, slong prec)
    Computes the modified Bessel function of the first kind  $I_\nu(z) = z^\nu (iz)^{-\nu} J_\nu(iz)$ .
void arb_hypgeom_bessel_i_scaled(arb_t res, const arb_t nu, const arb_t z, slong prec)
    Computes the function  $e^{-z} I_\nu(z)$ .
void arb_hypgeom_bessel_k(arb_t res, const arb_t nu, const arb_t z, slong prec)
    Computes the modified Bessel function of the second kind  $K_\nu(z)$ .
void arb_hypgeom_bessel_k_scaled(arb_t res, const arb_t nu, const arb_t z, slong prec)
    Computes the function  $e^z K_\nu(z)$ .
void arb_hypgeom_bessel_i_integration(arb_t res, const arb_t nu, const arb_t z, int scaled, slong
    prec)
void arb_hypgeom_bessel_k_integration(arb_t res, const arb_t nu, const arb_t z, int scaled, slong
    prec)
    Computes the modified Bessel functions using numerical integration.
```

9.18.11 Airy functions

```
void arb_hypgeom_airy(arb_t ai, arb_t ai_prime, arb_t bi, arb_t bi_prime, const arb_t z, slong
    prec)
    Computes the Airy functions  $(\text{Ai}(z), \text{Ai}'(z), \text{Bi}(z), \text{Bi}'(z))$  simultaneously. Any of the four func-
    tion values can be omitted by passing NULL for the unwanted output variables, speeding up the
    evaluation.
void arb_hypgeom_airy_jet(arb_ptr ai, arb_ptr bi, const arb_t z, slong len, slong prec)
    Writes to  $ai$  and  $bi$  the respective Taylor expansions of the Airy functions at the point  $z$ , truncated
    to length  $len$ . Either of the outputs can be NULL to avoid computing that function. The variable
     $z$  is not allowed to be aliased with the outputs. To simplify the implementation, this method does
    not compute the series expansions of the primed versions directly; these are easily obtained by
    computing one extra coefficient and differentiating the output with _arb_poly_derivative().
void _arb_hypgeom_airy_series(arb_ptr ai, arb_ptr ai_prime, arb_ptr bi, arb_ptr bi_prime,
    arb_srcptr z, slong zlen, slong len, slong prec)
void arb_hypgeom_airy_series(arb_poly_t ai, arb_poly_t ai_prime, arb_poly_t bi, arb_poly_t
    bi_prime, const arb_poly_t z, slong len, slong prec)
    Computes the Airy functions evaluated at the power series  $z$ , truncated to length  $len$ . As with the
    other Airy methods, any of the outputs can be NULL.
```



```
void arb_hypgeom_airy_zero(arb_t a, arb_t a_prime, arb_t b, arb_t b_prime, const fmpz_t n,
                          slong prec)
```

Computes the n -th real zero a_n , a'_n , b_n , or b'_n for the respective Airy function or Airy function derivative. Any combination of the four output variables can be *NULL*. The zeros are indexed by increasing magnitude, starting with $n = 1$ to follow the convention in the literature. An index n that is not positive is invalid input. The implementation uses asymptotic expansions for the zeros [PS1991] together with the interval Newton method for refinement.

9.18.12 Coulomb wave functions

```
void arb_hypgeom_coulomb(arb_t F, arb_t G, const arb_t l, const arb_t eta, const arb_t z, slong
                        prec)
```

Writes to F , G the values of the respective Coulomb wave functions $F_\ell(\eta, z)$ and $G_\ell(\eta, z)$. Either of the outputs can be *NULL*.

```
void arb_hypgeom_coulomb_jet(arb_ptr F, arb_ptr G, const arb_t l, const arb_t eta, const arb_t z,
                             slong len, slong prec)
```

Writes to F , G the respective Taylor expansions of the Coulomb wave functions at the point z , truncated to length len . Either of the outputs can be *NULL*.

```
void _arb_hypgeom_coulomb_series(arb_ptr F, arb_ptr G, const arb_t l, const arb_t eta,
                                arb_sreptr z, slong zlen, slong len, slong prec)
```

```
void arb_hypgeom_coulomb_series(arb_poly_t F, arb_poly_t G, const arb_t l, const arb_t eta,
                                const arb_poly_t z, slong len, slong prec)
```

Computes the Coulomb wave functions evaluated at the power series z , truncated to length len . Either of the outputs can be *NULL*.

9.18.13 Orthogonal polynomials and functions

```
void arb_hypgeom_chebyshev_t(arb_t res, const arb_t nu, const arb_t z, slong prec)
```

```
void arb_hypgeom_chebyshev_u(arb_t res, const arb_t nu, const arb_t z, slong prec)
```

```
void arb_hypgeom_jacobi_p(arb_t res, const arb_t n, const arb_t a, const arb_t b, const arb_t z,
                          slong prec)
```

```
void arb_hypgeom_gegenbauer_c(arb_t res, const arb_t n, const arb_t m, const arb_t z, slong prec)
```

```
void arb_hypgeom_laguerre_l(arb_t res, const arb_t n, const arb_t m, const arb_t z, slong prec)
```

```
void arb_hypgeom_hermite_h(arb_t res, const arb_t nu, const arb_t z, slong prec)
```

Computes Chebyshev, Jacobi, Gegenbauer, Laguerre or Hermite polynomials, or their extensions to non-integer orders.

```
void arb_hypgeom_legendre_p(arb_t res, const arb_t n, const arb_t m, const arb_t z, int type, slong
                           prec)
```

```
void arb_hypgeom_legendre_q(arb_t res, const arb_t n, const arb_t m, const arb_t z, int type, slong
                           prec)
```

Computes Legendre functions of the first and second kind. See [acb_hypgeom_legendre_p\(\)](#) and [acb_hypgeom_legendre_q\(\)](#) for definitions.

```
void arb_hypgeom_legendre_p_ui_deriv_bound(mag_t dp, mag_t dp2, ulong n, const arb_t x, const
                                           arb_t x2sub1)
```

Sets dp to an upper bound for $P'_n(x)$ and $dp2$ to an upper bound for $P''_n(x)$ given x assumed to represent a real number with $|x| \leq 1$. The variable $x2sub1$ must contain the precomputed value $1 - x^2$ (or $x^2 - 1$). This method is used internally to bound the propagated error for Legendre polynomials.

```
void arb_hypgeom_legendre_p_ui_zero(arb_t res, arb_t res_prime, ulong n, const arb_t x, slong K,
                                    slong prec)
```

```
void arb_hypgeom_legendre_p_ui_one(arb_t res, arb_t res_prime, ulong n, const arb_t x, slong K,
                                   slong prec)
```

```
void arb_hypgeom_legendre_p_ui_asymp(arb_t res, arb_t res_prime, ulong n, const arb_t x, slong
                                     K, slong prec)
```

```
void arb_hypgeom_legendre_p_ui_rec(arb_t res, arb_t res_prime, ulong n, const arb_t x, slong
                                   prec)
```

```
void arb_hypgeom_legendre_p_ui(arb_t res, arb_t res_prime, ulong n, const arb_t x, slong prec)
```

Evaluates the ordinary Legendre polynomial $P_n(x)$. If *res_prime* is non-NULL, simultaneously evaluates the derivative $P'_n(x)$.

The overall algorithm is described in [JM2018].

The versions *zero*, *one* respectively use the hypergeometric series expansions at $x = 0$ and $x = 1$ while the *asymp* version uses an asymptotic series on $(-1, 1)$ intended for large n . The parameter K specifies the exact number of expansion terms to use (if the series expansion truncated at this point does not give the exact polynomial, an error bound is computed automatically). The asymptotic expansion with error bounds is given in [Bog2012]. The *rec* version uses the forward recurrence implemented using fixed-point arithmetic; it is only intended for the interval $(-1, 1)$, moderate n and modest precision.

The default version attempts to choose the best algorithm automatically. It also estimates the amount of cancellation in the hypergeometric series and increases the working precision to compensate, bounding the propagated error using derivative bounds.

```
void arb_hypgeom_legendre_p_ui_root(arb_t res, arb_t weight, ulong n, ulong k, slong prec)
```

Sets *res* to the k -th root of the Legendre polynomial $P_n(x)$. We index the roots in decreasing order

$$1 > x_0 > x_1 > \dots > x_{n-1} > -1$$

(which corresponds to ordering the roots of $P_n(\cos(\theta))$ in order of increasing θ). If *weight* is non-NULL, it is set to the weight corresponding to the node x_k for Gaussian quadrature on $[-1, 1]$. Note that only $\lceil n/2 \rceil$ roots need to be computed, since the remaining roots are given by $x_k = -x_{n-1-k}$.

We compute an enclosing interval using an asymptotic approximation followed by some number of Newton iterations, using the error bounds given in [Pet1999]. If very high precision is requested, the root is subsequently refined using interval Newton steps with doubling working precision.

9.18.14 Dilogarithm

```
void arb_hypgeom_dilog(arb_t res, const arb_t z, slong prec)
```

Computes the dilogarithm $\text{Li}_2(z)$.

9.18.15 Hypergeometric sums

```
void arb_hypgeom_sum_fmpq_arb_forward(arb_t res, const fmpq *a, slong alen, const fmpq *b, slong
                                       blen, const arb_t z, int reciprocal, slong N, slong prec)
```

```
void arb_hypgeom_sum_fmpq_arb_rs(arb_t res, const fmpq *a, slong alen, const fmpq *b, slong blen,
                                  const arb_t z, int reciprocal, slong N, slong prec)
```

```
void arb_hypgeom_sum_fmpq_arb(arb_t res, const fmpq *a, slong alen, const fmpq *b, slong blen,
                               const arb_t z, int reciprocal, slong N, slong prec)
```

Sets *res* to the finite hypergeometric sum $\sum_{n=0}^{N-1} (\mathbf{a})_n z^n / (\mathbf{b})_n$ where $\mathbf{x}_n = (x_1)_n (x_2)_n \dots$, given vectors of rational parameters a (of length *alen*) and b (of length *blen*). If *reciprocal* is set, replace z by $1/z$. The *forward* version uses the forward recurrence, optimized by delaying divisions, the *rs* version uses rectangular splitting, and the default version uses an automatic algorithm choice.

```

void arb_hypgeom_sum_fmpq_imag_arb_forward(arb_t res1, arb_t res2, const fmpq *a, slong alen,
                                           const fmpq *b, slong blen, const arb_t z, int
                                           reciprocal, slong N, slong prec)

void arb_hypgeom_sum_fmpq_imag_arb_rs(arb_t res1, arb_t res2, const fmpq *a, slong alen, const
                                       fmpq *b, slong blen, const arb_t z, int reciprocal, slong N,
                                       slong prec)

void arb_hypgeom_sum_fmpq_imag_arb_bs(arb_t res1, arb_t res2, const fmpq *a, slong alen, const
                                       fmpq *b, slong blen, const arb_t z, int reciprocal, slong N,
                                       slong prec)

void arb_hypgeom_sum_fmpq_imag_arb(arb_t res1, arb_t res2, const fmpq *a, slong alen, const fmpq
                                   *b, slong blen, const arb_t z, int reciprocal, slong N, slong
                                   prec)

```

Sets *res1* and *res2* to the real and imaginary part of the finite hypergeometric sum $\sum_{n=0}^{N-1} (\mathbf{a})_n (iz)^n / (\mathbf{b})_n$. If *reciprocal* is set, replace *z* by $1/z$.

9.19 acb_elliptic.h – elliptic integrals and functions of complex variables

This module supports computation of elliptic (doubly periodic) functions, and their inverses, elliptic integrals. See [acb_modular.h](#) for the closely related modular forms and Jacobi theta functions.

Warning: incomplete elliptic integrals have very complicated branch structure when extended to complex variables. For some functions in this module, branch cuts may be artifacts of the evaluation algorithm rather than having a natural mathematical justification. The user should, accordingly, watch out for edge cases where the functions implemented here may differ from other systems or literature. There may also exist points where a function should be well-defined but the implemented algorithm fails to produce a finite result due to artificial internal singularities.

9.19.1 Complete elliptic integrals

```
void acb_elliptic_k(acb_t res, const acb_t m, slong prec)
```

Computes the complete elliptic integral of the first kind

$$K(m) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{dt}{(\sqrt{1 - t^2})(\sqrt{1 - mt^2})}$$

using the arithmetic-geometric mean: $K(m) = \pi / (2M(\sqrt{1 - m}))$.

```
void acb_elliptic_k_jet(acb_ptr res, const acb_t m, slong len, slong prec)
```

Sets the coefficients in the array *res* to the power series expansion of the complete elliptic integral of the first kind at the point *m* truncated to length *len*, i.e. $K(m + x) \in \mathbb{C}[[x]]$.

```
void _acb_elliptic_k_series(acb_ptr res, acb_srcptr m, slong mlen, slong len, slong prec)
```

```
void acb_elliptic_k_series(acb_poly_t res, const acb_poly_t m, slong len, slong prec)
```

Sets *res* to the complete elliptic integral of the first kind of the power series *m*, truncated to length *len*.

```
void acb_elliptic_e(acb_t res, const acb_t m, slong prec)
```

Computes the complete elliptic integral of the second kind

$$E(m) = \int_0^{\pi/2} \sqrt{1 - m \sin^2 t} dt = \int_0^1 \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}} dt$$

using $E(m) = (1 - m)(2mK'(m) + K(m))$ (where the prime denotes a derivative, not a complementary integral).

void **acb_elliptic_pi**(*acb_t* res, const *acb_t* n, const *acb_t* m, *long* prec)

Evaluates the complete elliptic integral of the third kind

$$\Pi(n, m) = \int_0^{\pi/2} \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}} = \int_0^1 \frac{dt}{(1 - nt^2) \sqrt{1 - t^2} \sqrt{1 - mt^2}}.$$

This implementation currently uses the same algorithm as the corresponding incomplete integral. It is therefore less efficient than the implementations of the first two complete elliptic integrals which use the AGM.

9.19.2 Legendre incomplete elliptic integrals

void **acb_elliptic_f**(*acb_t* res, const *acb_t* phi, const *acb_t* m, int pi, *long* prec)

Evaluates the Legendre incomplete elliptic integral of the first kind, given by

$$F(\phi, m) = \int_0^\phi \frac{dt}{\sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{(\sqrt{1 - t^2}) (\sqrt{1 - mt^2})}$$

on the standard strip $-\pi/2 \leq \operatorname{Re}(\phi) \leq \pi/2$. Outside this strip, the function extends quasiperiodically as

$$F(\phi + n\pi, m) = 2nK(m) + F(\phi, m), n \in \mathbb{Z}.$$

Inside the standard strip, the function is computed via the symmetric integral R_F .

If the flag *pi* is set to 1, the variable ϕ is replaced by $\pi\phi$, changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the first kind when $\phi = \frac{\pi}{2}$; that is, $F(\frac{\pi}{2}, m) = K(m)$.

void **acb_elliptic_e_inc**(*acb_t* res, const *acb_t* phi, const *acb_t* m, int pi, *long* prec)

Evaluates the Legendre incomplete elliptic integral of the second kind, given by

$$E(\phi, m) = \int_0^\phi \sqrt{1 - m \sin^2 t} dt = \int_0^{\sin \phi} \frac{\sqrt{1 - mt^2}}{\sqrt{1 - t^2}} dt$$

on the standard strip $-\pi/2 \leq \operatorname{Re}(\phi) \leq \pi/2$. Outside this strip, the function extends quasiperiodically as

$$E(\phi + n\pi, m) = 2nE(m) + E(\phi, m), n \in \mathbb{Z}.$$

Inside the standard strip, the function is computed via the symmetric integrals R_F and R_D .

If the flag *pi* is set to 1, the variable ϕ is replaced by $\pi\phi$, changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the second kind when $\phi = \frac{\pi}{2}$; that is, $E(\frac{\pi}{2}, m) = E(m)$.

void **acb_elliptic_pi_inc**(*acb_t* res, const *acb_t* n, const *acb_t* phi, const *acb_t* m, int pi, *long* prec)

Evaluates the Legendre incomplete elliptic integral of the third kind, given by

$$\Pi(n, \phi, m) = \int_0^\phi \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}} = \int_0^{\sin \phi} \frac{dt}{(1 - nt^2) \sqrt{1 - t^2} \sqrt{1 - mt^2}}$$

on the standard strip $-\pi/2 \leq \operatorname{Re}(\phi) \leq \pi/2$. Outside this strip, the function extends quasiperiodically as

$$\Pi(n, \phi + k\pi, m) = 2k\Pi(n, m) + \Pi(n, \phi, m), k \in \mathbb{Z}.$$

Inside the standard strip, the function is computed via the symmetric integrals R_F and R_J .

If the flag `pi` is set to 1, the variable ϕ is replaced by $\pi\phi$, changing the quasiperiod to 1.

The function reduces to a complete elliptic integral of the third kind when $\phi = \frac{\pi}{2}$; that is, $\Pi(n, \frac{\pi}{2}, m) = \Pi(n, m)$.

9.19.3 Carlson symmetric elliptic integrals

Carlson symmetric forms are the preferred form of incomplete elliptic integrals, due to their neat properties and relatively simple computation based on duplication theorems. There are five named functions: R_F, R_G, R_J , and R_C, R_D which are special cases of R_F and R_J respectively. We largely follow the definitions and algorithms in [Car1995] and chapter 19 in [NIST2012].

void **acb_elliptic_rf**(*acb_t* res, const *acb_t* x, const *acb_t* y, const *acb_t* z, int flags, *slong* prec)

Evaluates the Carlson symmetric elliptic integral of the first kind

$$R_F(x, y, z) = \frac{1}{2} \int_0^\infty \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}$$

where the square root extends continuously from positive infinity. The integral is well-defined for $x, y, z \notin (-\infty, 0)$, and with at most one of x, y, z being zero. When some parameters are negative real numbers, the function is still defined by analytic continuation.

In general, one or more duplication steps are applied until x, y, z are close enough to use a multivariate Taylor series.

The special case $R_C(x, y) = R_F(x, y, y) = \frac{1}{2} \int_0^\infty (t+x)^{-1/2} (t+y)^{-1} dt$ may be computed by setting y and z to the same variable. (This case is not yet handled specially, but might be optimized in the future.)

The *flags* parameter is reserved for future use and currently does nothing. Passing 0 results in default behavior.

void **acb_elliptic_rg**(*acb_t* res, const *acb_t* x, const *acb_t* y, const *acb_t* z, int flags, *slong* prec)

Evaluates the Carlson symmetric elliptic integral of the second kind

$$R_G(x, y, z) = \frac{1}{4} \int_0^\infty \frac{t}{\sqrt{(t+x)(t+y)(t+z)}} \left(\frac{x}{t+x} + \frac{y}{t+y} + \frac{z}{t+z} \right) dt$$

where the square root is taken continuously as in R_F . The evaluation is done by expressing R_G in terms of R_F and R_D . There are no restrictions on the variables.

void **acb_elliptic_rj**(*acb_t* res, const *acb_t* x, const *acb_t* y, const *acb_t* z, const *acb_t* p, int flags, *slong* prec)

void **acb_elliptic_rj_carlson**(*acb_t* res, const *acb_t* x, const *acb_t* y, const *acb_t* z, const *acb_t* p, int flags, *slong* prec)

```
void acb_elliptic_rj_integration(acb_t res, const acb_t x, const acb_t y, const acb_t z, const
                                acb_t p, int flags, slong prec)
```

Evaluates the Carlson symmetric elliptic integral of the third kind

$$R_J(x, y, z, p) = \frac{3}{2} \int_0^\infty \frac{dt}{(t+p)\sqrt{(t+x)(t+y)(t+z)}}$$

where the square root is taken continuously as in R_F .

Three versions of this function are available: the *carlson* version applies one or more duplication steps until x, y, z, p are close enough to use a multivariate Taylor series.

The duplication algorithm is not correct for all possible combinations of complex variables, since the square roots taken during the computation can introduce spurious branch cuts. According to [Car1995], a sufficient (but not necessary) condition for correctness is that x, y, z have nonnegative real part and that p has positive real part.

In other cases, the algorithm *might* still be correct, but no attempt is made to check this; it is up to the user to verify that the duplication algorithm is appropriate for the given parameters before calling this function.

The *integration* algorithm uses explicit numerical integration to translate the parameters to the right half-plane. This is reliable but can be slow.

The default method uses the *carlson* algorithm when it is certain to be correct, and otherwise falls back to the slow *integration* algorithm.

The special case $R_D(x, y, z) = R_J(x, y, z, z)$ may be computed by setting z and p to the same variable. This case is handled specially to avoid redundant arithmetic operations. In this case, the *carlson* algorithm is correct for all x, y and z .

The *flags* parameter is reserved for future use and currently does nothing. Passing 0 results in default behavior.

```
void acb_elliptic_rc1(acb_t res, const acb_t x, slong prec)
```

This helper function computes the special case $R_C(1, 1+x) = \operatorname{atan}(\sqrt{x})/\sqrt{x} = {}_2F_1(1, 1/2, 3/2, -x)$, which is needed in the evaluation of R_J .

9.19.4 Weierstrass elliptic functions

Elliptic functions may be defined on a general lattice $\Lambda = \{m2\omega_1 + n2\omega_2 : m, n \in \mathbb{Z}\}$ with half-periods ω_1, ω_2 . We simplify by setting $2\omega_1 = 1, 2\omega_2 = \tau$ with $\operatorname{im}(\tau) > 0$. To evaluate the functions on a general lattice, it is enough to make a linear change of variables. The main reference is chapter 23 in [NIST2012].

```
void acb_elliptic_p(acb_t res, const acb_t z, const acb_t tau, slong prec)
```

Computes Weierstrass's elliptic function

$$\wp(z, \tau) = \frac{1}{z^2} + \sum_{n^2+m^2 \neq 0} \left[\frac{1}{(z+m+n\tau)^2} - \frac{1}{(m+n\tau)^2} \right]$$

which satisfies $\wp(z, \tau) = \wp(z+1, \tau) = \wp(z+\tau, \tau)$. To evaluate the function efficiently, we use the formula

$$\wp(z, \tau) = \pi^2 \theta_2^2(0, \tau) \theta_3^2(0, \tau) \frac{\theta_4^2(z, \tau)}{\theta_1^2(z, \tau)} - \frac{\pi^2}{3} [\theta_2^4(0, \tau) + \theta_3^4(0, \tau)] .$$

```
void acb_elliptic_p_prime(acb_t res, const acb_t z, const acb_t tau, slong prec)
```

Computes the derivative $\wp'(z, \tau)$ of Weierstrass's elliptic function $\wp(z, \tau)$.

void **acb_elliptic_p_jet**(*acb_ptr* res, const *acb_t* z, const *acb_t* tau, *slong* len, *slong* prec)

Computes the formal power series $\wp(z+x, \tau) \in \mathbb{C}[[x]]$, truncated to length *len*. In particular, with *len* = 2, simultaneously computes $\wp(z, \tau), \wp'(z, \tau)$ which together generate the field of elliptic functions with periods 1 and τ .

void **_acb_elliptic_p_series**(*acb_ptr* res, *acb_srcptr* z, *slong* zlen, const *acb_t* tau, *slong* len, *slong* prec)

void **acb_elliptic_p_series**(*acb_poly_t* res, const *acb_poly_t* z, const *acb_t* tau, *slong* len, *slong* prec)

Sets *res* to the Weierstrass elliptic function of the power series *z*, with periods 1 and *tau*, truncated to length *len*.

void **acb_elliptic_invariants**(*acb_t* g2, *acb_t* g3, const *acb_t* tau, *slong* prec)

Computes the lattice invariants g_2, g_3 . The Weierstrass elliptic function satisfies the differential equation $[\wp'(z, \tau)]^2 = 4[\wp(z, \tau)]^3 - g_2\wp(z, \tau) - g_3$. Up to constant factors, the lattice invariants are the first two Eisenstein series (see [acb_modular_eisenstein\(\)](#)).

void **acb_elliptic_roots**(*acb_t* e1, *acb_t* e2, *acb_t* e3, const *acb_t* tau, *slong* prec)

Computes the lattice roots e_1, e_2, e_3 , which are the roots of the polynomial $4z^3 - g_2z - g_3$.

void **acb_elliptic_inv_p**(*acb_t* res, const *acb_t* z, const *acb_t* tau, *slong* prec)

Computes the inverse of the Weierstrass elliptic function, which satisfies $\wp(\wp^{-1}(z, \tau), \tau) = z$. This function is given by the elliptic integral

$$\wp^{-1}(z, \tau) = \frac{1}{2} \int_z^\infty \frac{dt}{\sqrt{(t-e_1)(t-e_2)(t-e_3)}} = R_F(z-e_1, z-e_2, z-e_3).$$

void **acb_elliptic_zeta**(*acb_t* res, const *acb_t* z, const *acb_t* tau, *slong* prec)

Computes the Weierstrass zeta function

$$\zeta(z, \tau) = \frac{1}{z} + \sum_{n^2+m^2 \neq 0} \left[\frac{1}{z-m-n\tau} + \frac{1}{m+n\tau} + \frac{z}{(m+n\tau)^2} \right]$$

which is quasiperiodic with $\zeta(z+1, \tau) = \zeta(z, \tau) + \zeta(1/2, \tau)$ and $\zeta(z+\tau, \tau) = \zeta(z, \tau) + \zeta(\tau/2, \tau)$.

void **acb_elliptic_sigma**(*acb_t* res, const *acb_t* z, const *acb_t* tau, *slong* prec)

Computes the Weierstrass sigma function

$$\sigma(z, \tau) = z \prod_{n^2+m^2 \neq 0} \left[\left(1 - \frac{z}{m+n\tau} \right) \exp \left(\frac{z}{m+n\tau} + \frac{z^2}{2(m+n\tau)^2} \right) \right]$$

which is quasiperiodic with $\sigma(z+1, \tau) = -e^{2\zeta(1/2, \tau)(z+1/2)}\sigma(z, \tau)$ and $\sigma(z+\tau, \tau) = -e^{2\zeta(\tau/2, \tau)(z+\tau/2)}\sigma(z, \tau)$.

9.20 acb_modular.h – modular forms of complex variables

This module provides methods for numerical evaluation of modular forms and Jacobi theta functions. See [acb_elliptic.h](#) for the closely related elliptic functions and integrals.

In the context of this module, *tau* or τ always denotes an element of the complex upper half-plane $\mathbb{H} = \{z \in \mathbb{C} : \text{Im}(z) > 0\}$. We also often use the variable *q*, variously defined as $q = e^{2\pi i \tau}$ (usually in relation to modular forms) or $q = e^{\pi i \tau}$ (usually in relation to theta functions) and satisfying $|q| < 1$. We will clarify the local meaning of *q* every time such a quantity appears as a function of τ .

As usual, the numerical functions in this module compute strict error bounds: if *tau* is represented by an *acb_t* whose content overlaps with the real line (or lies in the lower half-plane), and *tau* is passed to a function defined only on $\mathbb{H}$, then the output will have an infinite radius. The analogous behavior holds for functions requiring $|q| < 1$.

9.20.1 The modular group

type `psl2z_struct`

type `psl2z_t`

Represents an element of the modular group $\mathrm{PSL}(2, \mathbb{Z})$, namely an integer matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $ad - bc = 1$, and with signs canonicalized such that $c \geq 0$, and $d > 0$ if $c = 0$. The struct members a, b, c, d are of type `fmpz`.

void `psl2z_init`(`psl2z_t` g)

Initializes g and set it to the identity element.

void `psl2z_clear`(`psl2z_t` g)

Clears g .

void `psl2z_swap`(`psl2z_t` f , `psl2z_t` g)

Swaps f and g efficiently.

void `psl2z_set`(`psl2z_t` f , const `psl2z_t` g)

Sets f to a copy of g .

void `psl2z_one`(`psl2z_t` g)

Sets g to the identity element.

int `psl2z_is_one`(const `psl2z_t` g)

Returns nonzero iff g is the identity element.

void `psl2z_print`(const `psl2z_t` g)

Prints g to standard output.

void `psl2z_fprint`(FILE $*file$, const `psl2z_t` g)

Prints g to the stream $file$.

int `psl2z_equal`(const `psl2z_t` f , const `psl2z_t` g)

Returns nonzero iff f and g are equal.

void `psl2z_mul`(`psl2z_t` h , const `psl2z_t` f , const `psl2z_t` g)

Sets h to the product of f and g , namely the matrix product with the signs canonicalized.

void `psl2z_inv`(`psl2z_t` h , const `psl2z_t` g)

Sets h to the inverse of g .

int `psl2z_is_correct`(const `psl2z_t` g)

Returns nonzero iff g contains correct data, i.e. satisfying $ad - bc = 1$, $c \geq 0$, and $d > 0$ if $c = 0$.

void `psl2z_randtest`(`psl2z_t` g , `flint_rand_t` $state$, `slong` $bits$)

Sets g to a random element of $\mathrm{PSL}(2, \mathbb{Z})$ with entries of bit length at most $bits$ (or 1, if $bits$ is not positive). We first generate a and d , compute their Bezout coefficients, divide by the GCD, and then correct the signs.

9.20.2 Modular transformations

void **acb_modular_transform**(*acb_t* w, const *psl2z_t* g, const *acb_t* z, *slong* prec)

Applies the modular transformation g to the complex number z , evaluating

$$w = gz = \frac{az + b}{cz + d}.$$

void **acb_modular_fundamental_domain_approx_d**(*psl2z_t* g, double x, double y, double one_minus_eps)

void **acb_modular_fundamental_domain_approx_arf**(*psl2z_t* g, const *arf_t* x, const *arf_t* y, const *arf_t* one_minus_eps, *slong* prec)

Attempts to determine a modular transformation g that maps the complex number $x + yi$ to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by *one_minus_eps*.

The inputs are assumed to be finite numbers, with y positive.

Uses floating-point iteration, repeatedly applying either the transformation $z \leftarrow z + b$ or $z \leftarrow -1/z$. The iteration is terminated if $|x| \leq 1/2$ and $x^2 + y^2 \geq 1 - \varepsilon$ where $1 - \varepsilon$ is passed as *one_minus_eps*. It is also terminated if too many steps have been taken without convergence, or if the numbers end up too large or too small for the working precision.

The algorithm can fail to produce a satisfactory transformation. The output g is always set to *some* correct modular transformation, but it is up to the user to verify a posteriori that g maps $x + yi$ close enough to the fundamental domain.

void **acb_modular_fundamental_domain_approx**(*acb_t* w, *psl2z_t* g, const *acb_t* z, const *arf_t* one_minus_eps, *slong* prec)

Attempts to determine a modular transformation g that maps the complex number z to the fundamental domain or just slightly outside the fundamental domain, where the target tolerance (not a strict bound) is specified by *one_minus_eps*. It also computes the transformed value $w = gz$.

This function first tries to use **acb_modular_fundamental_domain_approx_d()** and checks if the result is acceptable. If this fails, it calls **acb_modular_fundamental_domain_approx_arf()** with higher precision. Finally, $w = gz$ is evaluated by a single application of g .

The algorithm can fail to produce a satisfactory transformation. The output g is always set to *some* correct modular transformation, but it is up to the user to verify a posteriori that w is close enough to the fundamental domain.

int **acb_modular_is_in_fundamental_domain**(const *acb_t* z, const *arf_t* tol, *slong* prec)

Returns nonzero if it is certainly true that $|z| \geq 1 - \varepsilon$ and $|\operatorname{Re}(z)| \leq 1/2 + \varepsilon$ where ε is specified by *tol*. Returns zero if this is false or cannot be determined.

9.20.3 Addition sequences

void **acb_modular_fill_addseq**(*slong* *tab, *slong* len)

Builds a near-optimal addition sequence for a sequence of integers which is assumed to be reasonably dense.

As input, the caller should set each entry in *tab* to -1 if that index is to be part of the addition sequence, and to 0 otherwise. On output, entry i in *tab* will either be zero (if the number is not part of the sequence), or a value j such that both j and $i - j$ are also marked. The first two entries in *tab* are ignored (the number 1 is always assumed to be part of the sequence).

9.20.4 Jacobi theta functions

Unfortunately, there are many inconsistent notational variations for Jacobi theta functions in the literature. Unless otherwise noted, we use the functions

$$\begin{aligned}\theta_1(z, \tau) &= -i \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i [(n+1/2)^2 \tau + (2n+1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} (-1)^n q^{n(n+1)} \sin((2n+1)\pi z) \\ \theta_2(z, \tau) &= \sum_{n=-\infty}^{\infty} \exp(\pi i [(n+1/2)^2 \tau + (2n+1)z]) = 2q_{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos((2n+1)\pi z) \\ \theta_3(z, \tau) &= \sum_{n=-\infty}^{\infty} \exp(\pi i [n^2 \tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} q^{n^2} \cos(2n\pi z) \\ \theta_4(z, \tau) &= \sum_{n=-\infty}^{\infty} (-1)^n \exp(\pi i [n^2 \tau + 2nz]) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n q^{n^2} \cos(2n\pi z)\end{aligned}$$

where $q = \exp(\pi i \tau)$ and $q_{1/4} = \exp(\pi i \tau / 4)$. Note that many authors write $q_{1/4}$ as $q^{1/4}$, but the principal fourth root $(q)^{1/4} = \exp(\frac{1}{4} \log q)$ differs from $q_{1/4}$ in general and some formulas are only correct if one reads “ $q^{1/4} = \exp(\pi i \tau / 4)$ ”. To avoid confusion, we only write q^k when k is an integer.

void **acb_modular_theta_transform**(int *R, int *S, int *C, const *psl2z_t* g)

We wish to write a theta function with quasiperiod τ in terms of a theta function with quasiperiod $\tau' = g\tau$, given some $g = (a, b; c, d) \in \text{PSL}(2, \mathbb{Z})$. For $i = 0, 1, 2, 3$, this function computes integers R_i and S_i (R and S should be arrays of length 4) and $C \in \{0, 1\}$ such that

$$\theta_{1+i}(z, \tau) = \exp(\pi i R_i / 4) \cdot A \cdot B \cdot \theta_{1+S_i}(z', \tau')$$

where $z' = z$, $A = B = 1$ if $C = 0$, and

$$z' = \frac{-z}{c\tau + d}, \quad A = \sqrt{\frac{i}{c\tau + d}}, \quad B = \exp\left(-\pi i c \frac{z^2}{c\tau + d}\right)$$

if $C = 1$. Note that A is well-defined with the principal branch of the square root since $A^2 = i/(c\tau + d)$ lies in the right half-plane.

Firstly, if $c = 0$, we have $\theta_i(z, \tau) = \exp(-\pi i b/4) \theta_i(z, \tau + b)$ for $i = 1, 2$, whereas θ_3 and θ_4 remain unchanged when b is even and swap places with each other when b is odd. In this case we set $C = 0$.

For an arbitrary g with $c > 0$, we set $C = 1$. The general transformations are given by Rademacher [Rad1973]. We need the function $\theta_{m,n}(z, \tau)$ defined for $m, n \in \mathbb{Z}$ by (beware of the typos in [Rad1973])

$$\theta_{0,0}(z, \tau) = \theta_3(z, \tau), \quad \theta_{0,1}(z, \tau) = \theta_4(z, \tau)$$

$$\theta_{1,0}(z, \tau) = \theta_2(z, \tau), \quad \theta_{1,1}(z, \tau) = i\theta_1(z, \tau)$$

$$\theta_{m+2,n}(z, \tau) = (-1)^n \theta_{m,n}(z, \tau)$$

$$\theta_{m,n+2}(z, \tau) = \theta_{m,n}(z, \tau).$$

Then we may write

$$\begin{aligned}\theta_1(z, \tau) &= \varepsilon_1 AB \theta_1(z', \tau') \\ \theta_2(z, \tau) &= \varepsilon_2 AB \theta_{1-c, 1+a}(z', \tau') \\ \theta_3(z, \tau) &= \varepsilon_3 AB \theta_{1+d-c, 1-b+a}(z', \tau') \\ \theta_4(z, \tau) &= \varepsilon_4 AB \theta_{1+d, 1-b}(z', \tau')\end{aligned}$$

where ε_i is an 8th root of unity. Specifically, if we denote the 24th root of unity in the transformation formula of the Dedekind eta function by $\varepsilon(a, b, c, d) = \exp(\pi i R(a, b, c, d)/12)$ (see `acb_modular_epsilon_arg()`), then:

$$\begin{aligned}\varepsilon_1(a, b, c, d) &= \exp(\pi i [R(-d, b, c, -a) + 1]/4) \\ \varepsilon_2(a, b, c, d) &= \exp(\pi i [-R(a, b, c, d) + (5 + (2 - c)a)]/4) \\ \varepsilon_3(a, b, c, d) &= \exp(\pi i [-R(a, b, c, d) + (4 + (c - d - 2)(b - a))]/4) \\ \varepsilon_4(a, b, c, d) &= \exp(\pi i [-R(a, b, c, d) + (3 - (2 + d)b)]/4)\end{aligned}$$

These formulas are easily derived from the formulas in [Rad1973] (Rademacher has the transformed/untransformed variables exchanged, and his “ ε ” differs from ours by a constant offset in the phase).

void `acb_modular_addseq_theta`(*slong* *exponents, *slong* *aindex, *slong* *bindex, *slong* num)

Constructs an addition sequence for the first *num* squares and triangular numbers interleaved (excluding zero), i.e. 1, 2, 4, 6, 9, 12, 16, 20, 25, 30 etc.

void `acb_modular_theta_sum`(*acb_ptr* theta1, *acb_ptr* theta2, *acb_ptr* theta3, *acb_ptr* theta4, const *acb_t* w, int w_is_unit, const *acb_t* q, *slong* len, *slong* prec)

Simultaneously computes the first *len* coefficients of each of the formal power series

$$\begin{aligned}\theta_1(z + x, \tau)/q_{1/4} &\in \mathbb{C}[[x]] \\ \theta_2(z + x, \tau)/q_{1/4} &\in \mathbb{C}[[x]] \\ \theta_3(z + x, \tau) &\in \mathbb{C}[[x]] \\ \theta_4(z + x, \tau) &\in \mathbb{C}[[x]]\end{aligned}$$

given $w = \exp(\pi i z)$ and $q = \exp(\pi i \tau)$, by summing a finite truncation of the respective theta function series. In particular, with *len* equal to 1, computes the respective value of the theta function at the point z . We require *len* to be positive. If *w_is_unit* is nonzero, w is assumed to lie on the unit circle, i.e. z is assumed to be real.

Note that the factor $q_{1/4}$ is removed from θ_1 and θ_2 . To get the true theta function values, the user has to multiply this factor back. This convention avoids unnecessary computations, since the user can compute $q_{1/4} = \exp(\pi i \tau/4)$ followed by $q = (q_{1/4})^4$, and in many cases when computing products or quotients of theta functions, the factor $q_{1/4}$ can be eliminated entirely.

This function is intended for $|q| \ll 1$. It can be called with any q , but will return useless intervals if convergence is not rapid. For general evaluation of theta functions, the user should only call this function after applying a suitable modular transformation.

We consider the sums together, alternatingly updating (θ_1, θ_2) or (θ_3, θ_4) . For $k = 0, 1, 2, \dots$, the powers of q are $\lfloor (k+2)^2/4 \rfloor = 1, 2, 4, 6, 9$ etc. and the powers of w are $\pm(k+2) = \pm 2, \pm 3, \pm 4, \dots$ etc. The scheme is illustrated by the following table:

	θ_1, θ_2	q^0	$(w^1 \pm w^{-1})$
$k = 0$	θ_3, θ_4	q^1	$(w^2 \pm w^{-2})$
$k = 1$	θ_1, θ_2	q^2	$(w^3 \pm w^{-3})$
$k = 2$	θ_3, θ_4	q^4	$(w^4 \pm w^{-4})$
$k = 3$	θ_1, θ_2	q^6	$(w^5 \pm w^{-5})$
$k = 4$	θ_3, θ_4	q^9	$(w^6 \pm w^{-6})$
$k = 5$	θ_1, θ_2	q^{12}	$(w^7 \pm w^{-7})$

For some integer $N \geq 1$, the summation is stopped just before term $k = N$. Let $Q = |q|$, $W = \max(|w|, |w^{-1}|)$, $E = \lfloor (N+2)^2/4 \rfloor$ and $F = \lfloor (N+1)/2 \rfloor + 1$. The error of the zeroth derivative can be bounded as

$$2Q^E W^{N+2} [1 + Q^F W + Q^{2F} W^2 + \dots] = \frac{2Q^E W^{N+2}}{1 - Q^F W}$$

provided that the denominator is positive (otherwise we set the error bound to infinity). When *len* is greater than 1, consider the derivative of order r . The term of index k and order r picks up a

factor of magnitude $(k+2)^r$ from differentiation of w^{k+2} (it also picks up a factor π^r , but we omit this until we rescale the coefficients at the end of the computation). Thus we have the error bound

$$2Q^E W^{N+2} (N+2)^r \left[1 + Q^F W \frac{(N+3)^r}{(N+2)^r} + Q^{2F} W^2 \frac{(N+4)^r}{(N+2)^r} + \dots \right]$$

which by the inequality $(1 + m/(N+2))^r \leq \exp(mr/(N+2))$ can be bounded as

$$\frac{2Q^E W^{N+2} (N+2)^r}{1 - Q^F W \exp(r/(N+2))},$$

again valid when the denominator is positive.

To actually evaluate the series, we write the even cosine terms as $w^{2n} + w^{-2n}$, the odd cosine terms as $w(w^{2n} + w^{-2n-2})$, and the sine terms as $w(w^{2n} - w^{-2n-2})$. This way we only need even powers of w and w^{-1} . The implementation is not yet optimized for real z , in which case further work can be saved.

This function does not permit aliasing between input and output arguments.

```
void acb_modular_theta_const_sum_basecase(acb_t theta2, acb_t theta3, acb_t theta4, const
                                          acb_t q, slong N, slong prec)
```

```
void acb_modular_theta_const_sum_rs(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q,
                                     slong N, slong prec)
```

Computes the truncated theta constant sums $\theta_2 = \sum_{k(k+1) < N} q^{k(k+1)}$, $\theta_3 = \sum_{k^2 < N} q^{k^2}$, $\theta_4 = \sum_{k^2 < N} (-1)^k q^{k^2}$. The *basecase* version uses a short addition sequence. The *rs* version uses rectangular splitting. The algorithms are described in [EHJ2016].

```
void acb_modular_theta_const_sum(acb_t theta2, acb_t theta3, acb_t theta4, const acb_t q, slong
                                  prec)
```

Computes the respective theta constants by direct summation (without applying modular transformations). This function selects an appropriate N , calls either *acb_modular_theta_const_sum_basecase()* or *acb_modular_theta_const_sum_rs()* or depending on N , and adds a bound for the truncation error.

```
void acb_modular_theta_notransform(acb_t theta1, acb_t theta2, acb_t theta3, acb_t theta4, const
                                   acb_t z, const acb_t tau, slong prec)
```

Evaluates the Jacobi theta functions $\theta_i(z, \tau)$, $i = 1, 2, 3, 4$ simultaneously. This function does not move τ to the fundamental domain. This is generally worse than *acb_modular_theta()*, but can be slightly better for moderate input.

```
void acb_modular_theta(acb_t theta1, acb_t theta2, acb_t theta3, acb_t theta4, const acb_t z,
                       const acb_t tau, slong prec)
```

Evaluates the Jacobi theta functions $\theta_i(z, \tau)$, $i = 1, 2, 3, 4$ simultaneously. This function moves τ to the fundamental domain and then also reduces z modulo τ before calling *acb_modular_theta_sum()*.

```
void acb_modular_theta_jet_notransform(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr
                                       theta4, const acb_t z, const acb_t tau, slong len, slong
                                       prec)
```

```
void acb_modular_theta_jet(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4, const
                            acb_t z, const acb_t tau, slong len, slong prec)
```

Evaluates the Jacobi theta functions along with their derivatives with respect to z , writing the first *len* coefficients in the power series $\theta_i(z + x, \tau) \in \mathbb{C}[[x]]$ to each respective output variable. The *notransform* version does not move τ to the fundamental domain or reduce z during the computation.

```
void _acb_modular_theta_series(acb_ptr theta1, acb_ptr theta2, acb_ptr theta3, acb_ptr theta4,
                              acb_srcptr z, slong zlen, const acb_t tau, slong len, slong prec)
```

```
void acb_modular_theta_series(acb_poly_t theta1, acb_poly_t theta2, acb_poly_t theta3,
                             acb_poly_t theta4, const acb_poly_t z, const acb_t tau, slong len,
                             slong prec)
```

Evaluates the respective Jacobi theta functions of the power series z , truncated to length len . Either of the output variables can be *NULL*.

9.20.5 Dedekind eta function

```
void acb_modular_addseq_eta(slong *exponents, slong *aindex, slong *bindex, slong num)
```

Constructs an addition sequence for the first num generalized pentagonal numbers (excluding zero), i.e. 1, 2, 5, 7, 12, 15, 22, 26, 35, 40 etc.

```
void acb_modular_eta_sum(acb_t eta, const acb_t q, slong prec)
```

Evaluates the Dedekind eta function without the leading 24th root, i.e.

$$\exp(-\pi i \tau / 12) \eta(\tau) = \sum_{n=-\infty}^{\infty} (-1)^n q^{(3n^2-n)/2}$$

given $q = \exp(2\pi i \tau)$, by summing the defining series.

This function is intended for $|q| \ll 1$. It can be called with any q , but will return useless intervals if convergence is not rapid. For general evaluation of the eta function, the user should only call this function after applying a suitable modular transformation.

The series is evaluated using either a short addition sequence or rectangular splitting, depending on the number of terms. The algorithms are described in [EHJ2016].

```
int acb_modular_epsilon_arg(const psl2z_t g)
```

Given $g = (a, b; c, d)$, computes an integer R such that $\varepsilon(a, b, c, d) = \exp(\pi i R / 12)$ is the 24th root of unity in the transformation formula for the Dedekind eta function,

$$\eta\left(\frac{a\tau + b}{c\tau + d}\right) = \varepsilon(a, b, c, d) \sqrt{c\tau + d} \eta(\tau).$$

```
void acb_modular_eta(acb_t r, const acb_t tau, slong prec)
```

Computes the Dedekind eta function $\eta(\tau)$ given τ in the upper half-plane. This function applies the functional equation to move τ to the fundamental domain before calling `acb_modular_eta_sum()`.

9.20.6 Modular forms

```
void acb_modular_j(acb_t r, const acb_t tau, slong prec)
```

Computes Klein's j -invariant $j(\tau)$ given τ in the upper half-plane. The function is normalized so that $j(i) = 1728$. We first move τ to the fundamental domain, which does not change the value of the function. Then we use the formula $j(\tau) = 32(\theta_2^8 + \theta_3^8 + \theta_4^8)^3 / (\theta_2 \theta_3 \theta_4)^8$ where $\theta_i = \theta_i(0, \tau)$.

```
void acb_modular_lambda(acb_t r, const acb_t tau, slong prec)
```

Computes the lambda function $\lambda(\tau) = \theta_2^4(0, \tau) / \theta_3^4(0, \tau)$, which is invariant under modular transformations $(a, b; c, d)$ where a, d are odd and b, c are even.

```
void acb_modular_delta(acb_t r, const acb_t tau, slong prec)
```

Computes the modular discriminant $\Delta(\tau) = \eta(\tau)^{24}$, which transforms as

$$\Delta\left(\frac{a\tau + b}{c\tau + d}\right) = (c\tau + d)^{12} \Delta(\tau).$$

The modular discriminant is sometimes defined with an extra factor $(2\pi)^{12}$, which we omit in this implementation.

void **acb_modular_eisenstein**(*acb_ptr* r, const *acb_t* tau, *slong* len, *slong* prec)

Computes simultaneously the first *len* entries in the sequence of Eisenstein series $G_4(\tau), G_6(\tau), G_8(\tau), \dots$, defined by

$$G_{2k}(\tau) = \sum_{m^2+n^2 \neq 0} \frac{1}{(m+n\tau)^{2k}}$$

and satisfying

$$G_{2k}\left(\frac{a\tau+b}{c\tau+d}\right) = (c\tau+d)^{2k} G_{2k}(\tau).$$

We first evaluate $G_4(\tau)$ and $G_6(\tau)$ on the fundamental domain using theta functions, and then compute the Eisenstein series of higher index using a recurrence relation.

9.20.7 Elliptic integrals and functions

See the *acb_elliptic.h* module for elliptic integrals and functions. The following wrappers are available for backwards compatibility.

void **acb_modular_elliptic_k**(*acb_t* w, const *acb_t* m, *slong* prec)

void **acb_modular_elliptic_k_cpx**(*acb_ptr* w, const *acb_t* m, *slong* len, *slong* prec)

void **acb_modular_elliptic_e**(*acb_t* w, const *acb_t* m, *slong* prec)

void **acb_modular_elliptic_p**(*acb_t* wp, const *acb_t* z, const *acb_t* tau, *slong* prec)

void **acb_modular_elliptic_p_zpx**(*acb_ptr* wp, const *acb_t* z, const *acb_t* tau, *slong* len, *slong* prec)

9.20.8 Class polynomials

void **acb_modular_hilbert_class_poly**(*fmprz_poly_t* res, *slong* D)

Sets *res* to the Hilbert class polynomial of discriminant D , defined as

$$H_D(x) = \prod_{(a,b,c)} \left(x - j \left(\frac{-b + \sqrt{D}}{2a} \right) \right)$$

where (a,b,c) ranges over the primitive reduced positive definite binary quadratic forms of discriminant $b^2 - 4ac = D$.

The Hilbert class polynomial is only defined if $D < 0$ and D is congruent to 0 or 1 mod 4. If some other value of D is passed as input, *res* is set to the zero polynomial.

9.21 acb_theta.h – Riemann theta functions

This module provides methods for the numerical evaluation of theta functions in any dimension $g \geq 1$. The algorithms will be detailed in the forthcoming paper [EK2025]. In the case $g = 1$, we rely on, but also improve on functionality from *acb_modular.h*. We also provide functionality to evaluate derivatives of theta functions, and also to evaluate Siegel modular forms in terms of theta functions when $g = 2$.

In this module, *tau* or τ always denotes an element of the Siegel upper half-space $\mathcal{H}_g$, i.e. τ is a symmetric $g \times g$ complex matrix with positive definite imaginary part, encoded as an *acb_mat_t*, while *z* or *zs* denotes a vector (resp. a tuple of vectors) in $\mathbb{C}^g$, encoded as an *acb_ptr* of length *g* (resp. *nb* times *g*). For each $a, b \in \{0, 1\}^g$, the Riemann theta function (of level 2) of characteristic (a, b) is the following analytic function in $\tau \in \mathcal{H}_g$ and $z \in \mathbb{C}^g$:

$$\theta_{a,b}(z, \tau) = \sum_{n \in \mathbb{Z}^g + \frac{a}{2}} \exp(\pi i n^T \tau n + 2\pi i n^T (z + \frac{b}{2})),$$

considering a , b and z as column vectors.

The numerical functions in this module always compute certified error bounds: for instance, if τ is represented by an `acb_mat_t` whose imaginary part is not certainly positive definite at the chosen working precision, then the output theta values will have an infinite radius.

9.21.1 Main user functions

First, if the user knows (a, b) as a vector `vec` of length $2g$ with entries 0 or 1, then the function

```
ulong acb_theta_char_set_slong_vec(const slong *vec, slong len)
```

(where `len` is $2g$) computes the `ulong` between 0 and $2^{2g} - 1$ whose binary expansion is exactly (a, b) (most significant bits first). Throughout, we order vectors of theta values according to this encoding: in other words, the computed `ulong` is exactly the index of $\theta_{a,b}$ inside the vector of theta values. For instance, the characteristic (a, b) where $a = (1, 0)$ and $b = (0, 0)$ in dimension $g = 2$ is numbered 8. With these conventions, the output of `acb_modular_theta()` in dimension 1 is $(-\theta_3, \theta_2, \theta_0, \theta_1)$. When we manipulate a or b individually (and hence compute only 2^g theta values), we map them to integers between 0 and $2^g - 1$ instead, taking `len` to be g .

There are two main user-facing functions in this modules:

```
void acb_theta_one(acb_t th, acb_srcptr z, const acb_mat_t tau, ulong ab, slong prec)
```

Sets `th` to the single value $\theta_{a,b}(z, \tau)$.

```
void acb_theta_all(acb_ptr th, acb_srcptr z, const acb_mat_t tau, int sqr, slong prec)
```

If `sqr` is zero, this function computes $\theta_{a,b}(z, \tau)$ for all $a, b \in \{0, 1\}^g$; if `sqr` is nonzero, it computes $\theta_{a,b}(z, \tau)^2$ instead using a faster algorithm. In both cases, the output, stored in `th`, is a vector of length 2^{2g} , and `acb_theta_all()` is only slightly slower than `acb_theta_one()`.

In both `acb_theta_one()` and `acb_theta_all()`, we handle the final argument `prec` as follows. Barring unexpected cancellations, the absolute value of $\theta_{a,b}(z, \tau)$ should be roughly

$$|\theta_{a,b}(z, \tau)| \approx e^{\pi y^T Y^{-1} y} e^{-d^2}$$

where

- Y and y denote the imaginary parts of τ and z respectively (we keep this notation throughout);
- d denotes the distance between the point $v = -Y^{-1}y \in \mathbb{R}^g$ and the shifted lattice $\mathbb{Z}^g + \frac{a}{2} \subset \mathbb{R}^g$ for the Euclidean norm given by the Gram matrix πY .

This leads us to define the normalized functions

$$\tilde{\theta}_{a,b}(z, \tau) = e^{-\pi y^T Y^{-1} y} \theta_{a,b}(z, \tau)$$

which are no longer holomorphic, but are uniformly bounded on $\mathbb{C}^g$ for a fixed τ . We use those internally for easier precision management: an argument `prec` means that $\tilde{\theta}_{a,b}(z, \tau)$ is computed with an absolute error bound of roughly 2^{-prec} . The expected error bound on the output of `acb_theta_all()` and similar functions is approximately $\exp(\pi y^T Y^{-1} y) \cdot 2^{-prec}$ to avoid unreasonable computations when y is very far from zero. Some internal functions also take the factor $\exp(-d^2)$ into account, and are documented as such.

Both functions `acb_theta_one()` and `acb_theta_all()` are in fact interfaces to the more complete method

```
void acb_theta_jet(acb_ptr th, acb_srcptr zs, slong nb, const acb_mat_t tau, slong ord, ulong ab,
                  int all, int sqr, slong prec)
```

which allows for several input vectors z (for the same matrix τ) for more efficiency, and also evaluates derivatives of theta functions. Its parameters are as follows:

- nb is the number of vectors z that we consider, and the input vector zs should have length nb times g . The output vector th is the concatenation of the nb individual outputs for each z .
- ord is the order of derivatives to be computed: we compute all the partial derivatives of theta functions $\theta_{a,b}$ with respect to the entries of z of total order up to ord . (Partial derivatives with respect to the entries of τ are then accounted for by the heat equation.) We refer to the documentation below for conventions on the normalization and ordering of those derivatives.
- ab indicates the characteristic (a,b) we are interested in. It is ignored if all is set to true.
- all is a boolean: if true (nonzero), then the output is a concatenation of 2^{2g} vectors of partial derivatives of theta functions, one for each characteristic. If false (zero), then we evaluate (partial derivatives of) a single theta function $\theta_{a,b}$ using a slightly faster algorithm.
- sqr is as in `acb_theta_all()`, but is ignored if ord is positive.

Behind the scenes, `acb_theta_jet()` works as follows: it first reduces the inputs (z, τ) using the action of the Siegel modular group $\mathrm{Sp}_{2g}(\mathbb{Z})$ (the symplectic group) on $\mathbb{C}^g \times \mathcal{H}_g$, then evaluates (partial derivatives of) theta functions on the reduced arguments, and finally applies the transformation formula for theta functions under $\mathrm{Sp}_{2g}(\mathbb{Z})$. The second step (evaluating theta functions) uses an advanced algorithm based on duplication formulas that has a uniform, quasi-linear complexity in terms of the required precision.

9.21.2 Example of usage

The following code snippet constructs the period matrix $\tau = iI_2$ for $g = 2$, computes the associated theta values at $z = 0$ at 100000 bits of precision in less than 10 ms, and prints them.

```
#include "acb.h"
#include "acb_mat.h"
#include "acb_theta.h"

int main()
{
    acb_mat_t tau;
    acb_ptr th, z;
    slong prec = 10000;

    acb_mat_init(tau, 2, 2);
    z = _acb_vec_init(2);
    th = _acb_vec_init(16);

    acb_mat_onei(tau);
    acb_theta_all(th, z, tau, 0, prec);
    _acb_vec_printd(th, 16, 5);

    acb_mat_clear(tau);
    _acb_vec_clear(z, 2);
    _acb_vec_clear(th, 16);
    flint_cleanup();
    return 0;
}
```

```
(1.1803 + 0j) +/- (3.34e-3010, 2.34e-3010j), (0.99254 + 0j) +/- (3.28e-3010, 2.
↪78e-3010j), (0.99254 + 0j) +/- (2.37e-3010, 1.87e-3010j), (0.83463 + 0j) +/- (2.
↪73e-3010, 2.23e-3010j), (0.99254 + 0j) +/- (1.08e-3010, 5.79e-3011j), (0 + 0j) +/-
↪- (1.35e-3009, 1.35e-3009j), (0.83463 + 0j) +/- (9.64e-3011, 4.63e-3011j), (0 +
↪0j) +/- (1.13e-3009, 1.13e-3009j), (0.99254 + 0j) +/- (3.20e-3009, 3.15e-3009j),
↪(0.83463 + 0j) +/- (3.79e-3009, 3.74e-3009j), (0 + 0j) +/- (4.04e-3011, 4.04e-
↪3011j), (0 + 0j) +/- (4.80e-3011, 4.80e-3011j), (0.83463 + 0j) +/- (8.30e-3010,
↪7.80e-3010j), (0 + 0j) +/- (5.18e-3009, 5.18e-3009j), (0 + 0j) +/- (1.00e-3011,
↪1.00e-3011j), (-6.1135e-3020 + 0j) +/- (2.80e-3010, 2.80e-3010j)
```

9.21.3 The Siegel modular group

We use the type `fmpz_mat_t` to handle matrices in $\mathrm{Sp}_{2g}(\mathbb{Z})$. In addition to the functions in this section, methods from `fmpz_mat.h` such as `fmpz_mat_equal()` can thus be used on symplectic matrices directly.

In the following functions (with the exception of `sp2gz_is_correct()`) we always assume that the input matrix `mat` is square of even size $2g$, and write it as

$$m = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

where $\alpha, \beta, \gamma, \delta$ are $g \times g$ blocks.

`slong sp2gz_dim(const fmpz_mat_t mat)`

Returns g , which is half the number of rows (or columns) of `mat`.

`void sp2gz_set_blocks(fmpz_mat_t mat, const fmpz_mat_t alpha, const fmpz_mat_t beta, const fmpz_mat_t gamma, const fmpz_mat_t delta)`

Sets `mat` to $\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$. The dimensions must match.

`void sp2gz_j(fmpz_mat_t mat)`

Sets `mat` to the symplectic matrix $J = \begin{pmatrix} 0 & I_g \\ -I_g & 0 \end{pmatrix}$.

`void sp2gz_block_diag(fmpz_mat_t mat, const fmpz_mat_t U)`

Sets `mat` to the symplectic matrix $\begin{pmatrix} U & 0 \\ 0 & U^{-t} \end{pmatrix}$. We require that $U \in \mathrm{GL}_g(\mathbb{Z})$.

`void sp2gz_trig(fmpz_mat_t mat, const fmpz_mat_t S)`

Sets `mat` to $\begin{pmatrix} I_g & S \\ 0 & I_g \end{pmatrix}$, where S is a symmetric $g \times g$ matrix.

`void sp2gz_embed(fmpz_mat_t res, const fmpz_mat_t mat)`

Assuming that `mat` is a symplectic matrix of size $2r \times 2r$ and `res` is square of size $2g \times 2g$ for some $g \geq r$, sets `res` to the symplectic matrix

$$\begin{pmatrix} \alpha & & \beta & \\ & I_{g-r} & & 0_{g-r} \\ \gamma & & \delta & \\ & 0_{g-r} & & I_{g-r} \end{pmatrix}$$

where $\alpha, \beta, \gamma, \delta$ are the $r \times r$ blocks of `mat`.

`void sp2gz_restrict(fmpz_mat_t res, const fmpz_mat_t mat)`

Assuming that `mat` is a symplectic matrix of size $2g \times 2g$ and `res` is square of size $2r \times 2r$ for some $r \leq g$, sets `res` to the matrix whose $r \times r$ blocks are the upper left corners of the corresponding $g \times g$ block of `mat`. The result may not be a symplectic matrix in general.

slong **sp2gz_nb_fundamental**(*slong* *g*)

Returns the number of fundamental symplectic matrices used in the reduction algorithm on $\mathcal{H}_g$. This number is 1 when $g = 1$ (the J matrix) and 19 when $g = 2$ [Got1959]. When $g > 2$, a complete set of matrices defining the boundary of a fundamental domain for the action of $\mathrm{Sp}_{2g}(\mathbb{Z})$ is not currently known. As a substitute, we consider two types of matrices: the $19g(g-1)/2$ matrices obtained by mimicking the $g = 2$ matrices on any pair of indices between 0 and $g-1$, and the 2^g matrices obtained by embedding a copy of a lower-dimensional J matrix on any subset of indices.

void **sp2gz_fundamental**(*fmpz_mat_t* *mat*, *slong* *j*)

Sets *mat* to the j -th fundamental symplectic matrix as defined above.

int **sp2gz_is_correct**(const *fmpz_mat_t* *mat*)

Returns true (nonzero) iff *mat* is a symplectic matrix.

int **sp2gz_is_j**(const *fmpz_mat_t* *mat*)

Returns true (nonzero) iff the symplectic matrix *mat* is the J matrix.

int **sp2gz_is_block_diag**(const *fmpz_mat_t* *mat*)

Returns true (nonzero) iff the symplectic matrix *mat* is of block-diagonal form as in *sp2gz_block_diag()*.

int **sp2gz_is_trig**(const *fmpz_mat_t* *mat*)

Returns true (nonzero) iff the symplectic matrix *mat* is of trigonal form as in *sp2gz_trig()*.

int **sp2gz_is_embedded**(*fmpz_mat_t* *res*, const *fmpz_mat_t* *mat*)

Assuming that *mat* is a $2g \times 2g$ symplectic matrix and *res* is square of size $2r$ for some $r \leq g$, returns true (nonzero) iff *mat* can be obtained as the result of *sp2gz_embed()* from a $2r \times 2r$ symplectic matrix, and store this matrix in *res*. Otherwise, returns false (0) and leaves *res* undefined.

void **sp2gz_inv**(*fmpz_mat_t* *inv*, const *fmpz_mat_t* *mat*)

Sets *inv* to the inverse of the symplectic matrix *mat*. In contrast with *fmpz_mat_inv()*, this involves no computation.

fmpz_mat_struct ***sp2gz_decompose**(*slong* **nb*, const *fmpz_mat_t* *mat*)

Returns a vector *res* of symplectic matrices and store its length in *nb* such that the following holds: *mat* is the product of the elements of *res* from left to right, and each element of *res* is block-diagonal, trigonal, the J matrix, an embedded J matrix from a lower dimension, or an embedded matrix from dimension 1. The output vector *res* will later have to be freed by the user as follows:

```
slong k;
for (k = 0; k < *nb; k++)
{
    fmpz_mat_clear(&res[k]);
}
flint_free(res);
```

void **sp2gz_randtest**(*fmpz_mat_t* *mat*, *flint_rand_t* *state*, *slong* *bits*)

Sets *mat* to a random symplectic matrix whose coefficients have length approximately *bits*, obtained as a product of block-diagonal and trigonal symplectic matrices and the J matrix.

9.21.4 The Siegel half space

We continue to denote by $\alpha, \beta, \gamma, \delta$ the $g \times g$ blocks of mat , which is always assumed to be symplectic.

void **acb_siegel_cocycle**(*acb_mat_t* c, const *fmpz_mat_t* mat, const *acb_mat_t* tau, *slong* prec)

Sets c to $\gamma\tau + \delta$.

void **acb_siegel_transform_cocycle_inv**(*acb_mat_t* w, *acb_mat_t* c, *acb_mat_t* cinv, const *fmpz_mat_t* mat, const *acb_mat_t* tau, *slong* prec)

Sets w , c and $cinv$ to $(\alpha\tau + \beta)(\gamma\tau + \delta)^{-1}$, $\gamma\tau + \delta$ and $(\gamma\tau + \delta)^{-1}$ respectively.

void **acb_siegel_transform**(*acb_mat_t* w, const *fmpz_mat_t* mat, const *acb_mat_t* tau, *slong* prec)

Sets w to $(\alpha\tau + \beta)(\gamma\tau + \delta)^{-1}$.

void **acb_siegel_cho_yinv**(*arb_mat_t* cho, *arb_mat_t* yinv, const *acb_mat_t* tau, *slong* prec)

Sets $yinv$ to the inverse of the imaginary part Y of tau , and sets cho to an upper-triangular Cholesky matrix for πY , i.e. to the upper-triangular matrix C with positive diagonal entries such that $\pi Y = C^T C$. If one cannot determine that Y is positive definite at the current working precision, $yinv$ and cho are set to indeterminate matrices.

void **acb_siegel_reduce**(*fmpz_mat_t* mat, const *acb_mat_t* tau, *slong* prec)

Sets mat to a symplectic matrix such that $mat \cdot \tau$ is as reduced as possible, repeatedly reducing the imaginary and real parts of τ and applying fundamental symplectic matrices. If the coefficients of τ do not have a reasonable size or if $\det Y$ is vanishingly small, we simply set mat to the identity.

int **acb_siegel_is_reduced**(const *acb_mat_t* tau, *slong* tol_exp, *slong* prec)

Returns true (nonzero) iff it is certainly true that τ belongs to the reduced domain defined by the tolerance parameter $\varepsilon = 2^{tol\_exp}$. This means the following: $|\operatorname{Re}(\tau_{j,k})| < \frac{1}{2} + \varepsilon$ for all $0 \leq j, k < g$; the imaginary part of τ passes **arb_mat_spd_is_lll_reduced()** with the same parameters; and for every matrix obtained from **sp2gz_fundamental()**, the determinant of the corresponding cocycle is at least $1 - \varepsilon$.

slong **acb_siegel_kappa**(*acb_t* sqrtDET, const *fmpz_mat_t* mat, const *acb_mat_t* tau, int *sqr*, *slong* prec)

If sqr is zero, returns $0 \leq r < 8$ such that $\kappa(mat) = \zeta_8^r$ and sets $sqrtDET$ to the corresponding square root of $\det(\gamma\tau + \delta)$ in the theta transformation formula. If sqr is nonzero, the function instead returns $0 \leq r < 4$ such that $\kappa(mat)^2 = i^r$ and sets $sqrtDET$ to $\det(\gamma\tau + \delta)$.

By [Igu1972], p. 176 and [Mum1983], p. 189, for any symplectic matrix m , any $(z, \tau) \in \mathbb{C}^g \times \mathcal{H}_g$, and any characteristic (a, b) , we have

$$\theta_{a,b}(m \cdot (z, \tau)) = \kappa(m) \zeta_8^{e(m,a,b)} \det(\gamma\tau + \delta)^{1/2} e^{\pi i z^T (\gamma\tau + \delta)^{-1} \gamma z} \theta_{a',b'}(z, \tau)$$

where

- γ, δ are the lower $g \times g$ blocks of m ,
- a', b' is another characteristic depending on m, a, b ,
- $\zeta_8 = \exp(i\pi/4)$,
- $e(m, a, b)$ is an integer given by an explicit formula in terms of m, a, b (this is ϕ_m in Igusa's notation), and
- $\kappa(m)$ is an 8th root of unity, only well-defined up to sign unless we choose a particular branch of $\det(\gamma\tau + \delta)^{1/2}$ on $\mathcal{H}_g$. Hence $\kappa(m)^2$ is a well-defined 4th root of unity.

We proceed as follows. After applying **sp2gz_decompose()**, we only have to consider four special cases for mat . If mat is trigonal or block-diagonal, one can compute its action on $\theta_{0,0}$ directly. If mat is an embedded matrix from $\operatorname{SL}_2(\mathbb{Z})$, we rely on **acb_modular_theta_transform()**. Finally,

if mat is an embedded J matrix from dimension $2 \leq r \leq g$, then $\kappa(m)\zeta_8^{e(m,0,0)}i^{r/2}\det(\tau_0)^{1/2} = 1$, where τ_0 denotes the upper left $r \times r$ submatrix of τ and the branch of the square root is chosen such that the result is $i^{g/2}\det(Y)$ when $\tau = iY$ is purely imaginary.

To compute $\det(\tau_0)^{1/2}$ (assuming that sqr is false), we pick a purely imaginary matrix A and consider the polynomial $P(t) = \det(A + \frac{t+1}{2}(\tau_0 - A))$. Up to choosing another A , we may assume that it has degree g and that its roots (as complex balls) do not intersect the segment $[-1, 1] \subset \mathbb{C}$. We then find the correct branch of $P(t)^{1/2}$ between $t = -1$ and $t = 1$ following [MN2019].

slong **acb_siegel_kappa2**(const *fmpz_mat_t* mat)

Returns $0 \leq r < 3$ such that $\kappa(mat)^2 = i^r$, which makes sense without reference to a branch of $\det(\gamma\tau + \delta)^{1/2}$.

This is a simpler interface to **acb_siegel_kappa()** when sqr is true.

void **acb_siegel_randtest**(*acb_mat_t* tau, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Sets τ to a random matrix in $\mathcal{H}_g$, possibly far from being reduced.

void **acb_siegel_randtest_reduced**(*acb_mat_t* tau, *flint_rand_t* state, *slong* prec, *slong* mag_bits)

Sets τ to a random reduced matrix in $\mathcal{H}_g$ whose imaginary part possibly has large entries.

void **acb_siegel_randtest_compact**(*acb_mat_t* tau, *flint_rand_t* state, int exact, *slong* prec)

Sets τ to a random reduced matrix in $\mathcal{H}_g$ whose imaginary part has bounded entries. If *exact* is nonzero, then the entries of τ are set to exact (dyadic) complex numbers.

void **acb_siegel_randtest_vec**(*acb_ptr* z, *flint_rand_t* state, *slong* g, *slong* prec)

Sets z to a random vector of length g , possibly with large entries.

void **acb_siegel_randtest_vec_reduced**(*acb_ptr* zs, *flint_rand_t* state, *slong* nb, const *acb_mat_t* tau, int exact, *slong* prec)

Sets zs to the concatenation of nb random vectors z sampled from $[-1, 1]^g + \tau[-1, 1]^g$, i.e. close to being reduced with respect to τ . If *exact* is nonzero, then the entries of zs are set to exact (dyadic) complex numbers.

9.21.5 Theta characteristics

int **acb_theta_char_bit**(*ulong* ch, *slong* j, *slong* n)

Returns the j -th bit of ch seen as an element of $\{0, 1\}^n$ with the above conventions, counting from $j = 0$ to $n - 1$.

void **acb_theta_char_get_arb**(*arb_ptr* v, *ulong* a, *slong* g)

void **acb_theta_char_get_acb**(*acb_ptr* v, *ulong* a, *slong* g)

Sets v to $a/2$ seen as an element of $\mathbb{R}^g$ or $\mathbb{C}^g$ respectively.

slong **acb_theta_char_dot**(*ulong* a, *ulong* b, *slong* g)

Returns $\sum_{i=0}^{g-1} a_i b_i$ modulo 4 as an integer between 0 and 3, where a_i, b_i for $0 \leq i < g$ denote the bits of a and b respectively.

slong **acb_theta_char_dot_slong**(*ulong* a, const *slong* *n, *slong* g)

Returns $\sum_{i=0}^{g-1} a_i n_i$ modulo 4 as an integer between 0 and 3.

int **acb_theta_char_is_even**(*ulong* ab, *slong* g)

Returns true iff the characteristic (a, b) is even, i.e. $a^T b$ is divisible by 2. Odd characteristics (a, b) have the property that $\theta_{a,b}(0, \tau)$ is identically zero.

void **acb_theta_char_table**(*ulong* *ch, *slong* *e, const *fmpz_mat_t* mat, *ulong* ab, int all)

If *all* is false, sets ch to the theta characteristic (a', b') and sets e to $e(mat, a, b)$ as in the transformation formula (see **acb_siegel_kappa()**). If *all* is true, then ab is ignored, and ch and e are set to vectors of length 2^{2g} containing this output for all characteristics from 0 to $2^{2g} - 1$.

void **acb_theta_char_shuffle**(*acb_ptr* res, const *fmpz_mat_t* mat, *acb_srcptr* th, int sqr, *slong* prec)

Partially applies the theta transformation formula to the given vector *th* for the symplectic matrix *mat* and stores the output in *res*. This omits the κ , determinant, and exponential factors from the formula. If *sqr* is nonzero, then ζ_8 is replaced by i in the formula to mimic the transformation formula on squared theta values. This is only used for testing.

9.21.6 Toolbox for derivatives

In this module, we only consider the successive partial derivatives of $\theta_{a,b}(z, \tau)$ with respect to the g coordinates of z , because derivatives with respect to τ are accounted for by the heat equation

$$\frac{\partial \theta_{a,b}}{\partial \tau_{j,k}} = \frac{1}{2\pi i(1 + \delta_{j,k})} \frac{\partial^2 \theta_{a,b}}{\partial z_j \partial z_k}.$$

where δ is the Kronecker symbol. We encode tuples of derivation orders, henceforth called “derivation tuples”, as vectors of type *slong* and length g . In agreement with *acb_modular.h*, we also normalize derivatives in the same way as in the Taylor expansion, so that the tuple $(k_0, \dots, k_{g-1})$ corresponds to the differential operator

$$\frac{1}{k_0!} \cdots \frac{1}{k_{g-1}!} \cdot \frac{\partial^{|k|}}{\partial z_0^{k_0} \cdots \partial z_{g-1}^{k_{g-1}}}$$

where $|k| := \sum k_i$. We always consider all derivation tuples up to a total order *ord*, and order them first by their total order, then reverse-lexicographically. For example, in the case $g = 2$, the sequence of orders is $(0, 0)$, $(1, 0)$, $(0, 1)$, $(2, 0)$, $(1, 1)$, etc.

This sections gathers methods to work with partial derivatives of holomorphic functions in general.

slong **acb_theta_jet_nb**(*slong* ord, *slong* g)

Returns the number of derivation tuples with total order at most *ord*. The result is zero if *ord* is negative.

slong **acb_theta_jet_total_order**(const *slong* *tup, *slong* g)

Returns the total derivation order for the given tuple *tup* of length g .

void **acb_theta_jet_tuples**(*slong* *tups, *slong* ord, *slong* g)

Sets *tups* to the concatenation of all derivation tuples up to total order *ord*.

slong **acb_theta_jet_index**(const *slong* *tup, *slong* g)

Returns n such that *tup* is the n -th derivation tuple of length g .

void **acb_theta_jet_mul**(*acb_ptr* res, *acb_srcptr* v1, *acb_srcptr* v2, *slong* ord, *slong* g, *slong* prec)

Sets *res* to the vector of derivatives of the product fg , assuming that *v1* and *v2* contains the derivatives of f and g respectively.

void **acb_theta_jet_compose**(*acb_ptr* res, *acb_srcptr* v, const *acb_mat_t* N, *slong* ord, *slong* prec)

Sets *res* to the vector of derivatives of the composition $f(Nz)$, assuming that *v* contains the derivatives of f at the point Nz . The dimension g is obtained as the size of the square matrix N .

void **acb_theta_jet_exp_pi_i**(*acb_ptr* res, *arb_srcptr* a, *slong* ord, *slong* g, *slong* prec)

Sets *res* to the vector of derivatives of the function $\exp(\pi i(a_0 z_1 + \cdots + a_{g-1} z_{g-1}))$ at $z = 0$, where $a_0, \dots, a_{g-1}$ are the entries of a .

void **acb_theta_jet_exp_qf**(*acb_ptr* res, *acb_srcptr* z, const *acb_mat_t* N, *slong* ord, *slong* prec)

Sets *res* to the vector of derivatives of the function $\exp(\pi i z^T N z)$ at the chosen point z . The dimension g is obtained as the size of the square matrix N .

9.21.7 Ellipsoids

The most direct way of evaluating Riemann theta functions consists in evaluating a partial sum of the exponential series defining them, then adding an error bound coming from the tail of the series. We refer to this strategy as the *summation algorithms*.

The upper bound on the tail is obtained from the triangle inequality. First, we use the relation

$$\theta_{a,b}(z, \tau) = \exp(\pi i a^T (z + \frac{b}{2}) + \pi i a^T \tau a / 4) \theta_{0,b}(z + \tau \frac{a}{2}, \tau)$$

to avoid summing over $\mathbb{Z}^g + \frac{a}{2}$ with a nonzero a . Next, to analyze the absolute value of each term in the sum defining $\theta_{0,b}(z, \tau)$, we write:

$$|\exp(i\pi n^T \tau n + 2n^T (z + \frac{b}{2}))| = \exp(\pi y^T Y^{-1} y) \exp(-\|n + Y^{-1} y\|_\tau^2)$$

(notation as in the introduction). Thus, the exponential terms whose absolute values are less than a given threshold correspond to lattice points $n \in \mathbb{Z}^g$ lying outside a certain ball centered in $v = -Y^{-1}y$ for $\|\cdot\|_\tau$; in other words, we should be computing partial sums over points $n \in \mathbb{Z}^g$ lying in certain ellipsoids, as in [DHBHS2004]. This section gathers methods to manipulate such ellipsoids directly.

Fix an upper-triangular matrix C with positive diagonal entries (henceforth called a “Cholesky matrix”), a radius $R \geq 0$, a vector $v \in \mathbb{R}^g$, and $1 \leq d \leq g$. Consider the ellipsoid E consisting of points $n = (n_0, \dots, n_{g-1})$ satisfying $(v + Cn)^T (v + Cn) \leq R^2$ and such that their last coordinates $n_d, \dots, n_{g-1}$ are fixed. We encode E as follows: we store the endpoints and midpoint of the interval of allowed values for n_{d-1} as *slong*’s, and if $d \geq 1$, we store a $(d-1)$ -dimensional “child” of E for each value of n_{d-1} as another ellipsoid in a recursive way. Children are partitioned between left and right children depending on the position of n_{d-1} relative to the midpoint. When $d = g$ and for a fixed Cholesky matrix C , this representation uses $O(R^{g-1})$ space for an ellipsoid of radius R containing approximately $O(R^g)$ points.

type `acb_theta_eld_struct`

type `acb_theta_eld_t`

An `acb_theta_eld_t` is an array of length one of type `acb_theta_eld_struct` encoding an ellipsoid as described above, allowing it to be passed by reference.

void `acb_theta_eld_init(acb_theta_eld_t E, slong d, slong g)`

Initializes E as a d -dimensional ellipsoid in ambient dimension g . We require $1 \leq d \leq g$.

void `acb_theta_eld_clear(acb_theta_eld_t E)`

Clears E as well as any recursive data contained in it.

int `acb_theta_eld_set(acb_theta_eld_t E, const arb_mat_t C, const arf_t R2, arb_srcptr v)`

Assuming that C is upper-triangular with positive diagonal entries, attempts to set E to represent an ellipsoid as defined above, where $R2$ indicates R^2 , and returns 1 upon success. If the ellipsoid points do not fit in *slong*’s or if the ellipsoid is unreasonably large, returns 0 instead and leaves E undefined.

The following functions are available after E has been initialized and then computed using `acb_theta_eld_init()` and `acb_theta_eld_set()`.

slong `acb_theta_eld_nb_pts(acb_theta_eld_t E)`

Returns the number of points contained in E , which is stored in the data structure.

void `acb_theta_eld_points(slong *pts, const acb_theta_eld_t E)`

Sets pts to the list of all the points in E , as a concatenation of vectors of length g . The vector pts must be pre-allocated to the correct length.

slong **acb_theta_eld_box**(const *acb_theta_eld_t* E, *slong* j)

Returns an upper bound on the absolute value of the j -th coordinate of any point stored in E . We require $0 \leq j < g$.

slong **acb_theta_eld_nb_border**(*acb_theta_eld_t* E)

Returns the number of points in the “border” of E , a certain set of points lying just outside E . This number is stored in the data structure.

void **acb_theta_eld_border**(*slong* *pts, const *acb_theta_eld_t* E)

Sets pts to the list of all the points in the border of E . The vector pts must be pre-allocated to the correct length. This is only used for testing.

int **acb_theta_eld_contains**(const *acb_theta_eld_t* E, *slong* *pt)

Returns true (nonzero) iff pt is contained in E . The vector pt must be of length g .

void **acb_theta_eld_print**(const *acb_theta_eld_t* E)

Prints a faithful description of E . This may be unwieldy in high dimensions.

void **acb_theta_eld_distances**(*arb_ptr* ds, *acb_srcptr* zs, *slong* nb, const *acb_mat_t* tau, *slong* prec)

Sets ds to the concatenation of the following nb vectors of length 2^g : for each input vector z , we compute $\text{Dist}_\tau(-Y^{-1}y, \mathbb{Z}^g + \frac{a}{2})^2$ for all $a \in \{0, 1\}^g$, where Dist_τ denotes the distance attached to $\|\cdot\|_\tau$.

We first round the coordinates of $-Y^{-1}y$ to obtain an element of $\mathbb{Z}^g + \frac{a}{2}$ providing an upper bound on the distance, then enumerate all the points in the ellipsoid of that radius to find all the closer points, if any.

9.21.8 Error bounds in summation algorithms

To compute the correct ellipsoids in summation algorithms for a target working precision, we use the following upper bound on the tail of the series: by [EK2025], for any $v \in \mathbb{R}^g$, any upper-triangular Cholesky matrix C , any nonnegative ord , and any $R \geq 0$, we have

$$\sum_{n \in C\mathbb{Z}^g + Cv, \|n\|^2 \geq R^2} \|n\|^{ord} e^{-\|n\|^2} \leq (1 + \sqrt{\frac{8}{\pi}}) \max\{2, R\}^{g-1} R^p e^{-R^2} \prod_{j=0}^{g-1} (1 + \frac{\sqrt{2\pi}}{\gamma_j})$$

where $\gamma_0, \dots, \gamma_{g-1}$ are the diagonal coefficients of C .

void **acb_theta_sum_radius**(*arf_t* R2, *arf_t* eps, const *arb_mat_t* cho, *slong* ord, *slong* prec)

Sets eps to 2^{-prec} and chooses $R2$ such that the above upper bound for $R2$ and the given ord is at most eps , where C is cho . When $ord = 0$, the square root of $R2$ is a suitable ellipsoid radius for a partial sum of the theta series, and eps is an upper bound on the absolute value of the tail of the series defining $\tilde{\theta}_{a,b}$.

void **acb_theta_sum_jet_radius**(*arf_t* R2, *arf_t* eps, const *arb_mat_t* cho, *arb_srcptr* v, *slong* ord, *slong* prec)

Computes a suitable squared radius $R2$ and error bound eps on the tail of the theta series as in **acb_theta_sum_radius()**, but in the context of evaluating partial derivatives of theta functions up to order ord . The input vector v should be $-CY^{-1}y$, where C is the Cholesky matrix for πY .

We can rewrite the the differentiated series as

$$\begin{aligned} \frac{\partial^{|k|} \theta_{a,b}}{\partial z_0^{k_0} \dots \partial z_{g-1}^{k_{g-1}}}(z, \tau) &= (2\pi i)^{|k|} \sum_{n \in \mathbb{Z}^g + \frac{a}{2}} n_0^{k_0} \dots n_{g-1}^{k_{g-1}} e^{\pi i n^T \tau n + 2\pi i n^T (z + \frac{b}{2})} \\ &= (2\pi i)^{|k|} e^{\pi y^T Y^{-1} y} \sum_{n \in \mathbb{Z}^g + \frac{a}{2}} n_0^{k_0} \dots n_{g-1}^{k_{g-1}} \xi_n e^{-\pi (n + Y^{-1}y)^T Y (n + Y^{-1}y)}. \end{aligned}$$

where $|\xi_n| = 1$. We ignore the leading multiplicative factor. Writing $m = Cn + v$, we have

$$n_0^{k_0} \cdots n_{g-1}^{k_{g-1}} \leq (\|C^{-1}\|_\infty \|n\|_2 + \|Y^{-1}y\|_\infty)^{|k|}.$$

Using the upper bound from `acb_theta_sum_radius()`, we see that the absolute value of the tail of the series, when summing outside the ellipsoid centered in v of radius R , is bounded above by

$$(\|C^{-1}\|_\infty R + \|Y^{-1}y\|_\infty)^{|k|} 2^{2g+2} R^{g-1} e^{-R^2} \prod_{j=0}^{g-1} (1 + \gamma_j^{-1}).$$

The output values `R2` and `eps` are such that this upper bound is at most `eps` when R is the square root of `R2`.

To obtain them, we first compute `R2` and `eps` using `acb_theta_sum_radius()` with `ord = 0`. If $R \leq \|Y^{-1}y\|_\infty / \|C^{-1}\|_\infty$, we simply multiply `eps` by $\max\{1, 2\|Y^{-1}y\|_\infty\}^{ord}$. Otherwise, we compute `R2` and `eps` using `acb_theta_sum_radius()` with the given value of `ord`. We can then set `R2` to the maximum of `R2` and $\|Y^{-1}y\|_\infty / \|C^{-1}\|_\infty$, and multiply `eps` by $\max\{1, 2\|C^{-1}\|_\infty\}^{ord}$.

void `acb_theta_sum_term`(`acb_t` res, `acb_sreptr` z, const `acb_mat_t` tau, `slong` *tup, `slong` *n, `slong` prec)

Sets `res` to $n_0^{k_0} \cdots n_{g-1}^{k_{g-1}} \exp(\pi i(n^T \tau n + 2n^T z))$, where the k_j and n_j denotes the j -th entry in `tup` and `n` respectively. The vector `tup` may be `NULL`, which is understood to mean the zero tuple. This is only used for testing.

`slong` `acb_theta_sum_addprec`(const `arb_t` d)

Returns an integer that is close to $d/\log(2)$ if d is finite and of reasonable size, and otherwise returns 0.

9.21.9 Context structures in summation algorithms

After the relevant ellipsoid has been computed, summation algorithms only involve exponential terms in τ and z . Sometimes, especially in the setting of the quasi-linear algorithms below, these exponentials can be computed once, and then used for several calls to summation functions. This section introduces context structures to make these manipulations easier.

type `acb_theta_ctx_tau_struct`

type `acb_theta_ctx_tau_t`

An `acb_theta_ctx_tau_t` is an array of length one of type `acb_theta_ctx_tau_struct` containing all the necessary data to run the summation algorithm on a given matrix $\tau \in \mathcal{H}_g$. In particular, it contains a matrix `exp_tau_div_4` whose (j, k) entry (when $j \leq k$) is $\exp(\pi i(2 - \delta_{j,k})\tau_{j,k}/4)$, where δ denotes the Kronecker symbol. It also contains the Cholesky matrix for πY if $g > 1$.

type `acb_theta_ctx_z_struct`

type `acb_theta_ctx_z_t`

An `acb_theta_ctx_z_t` is an array of length one of type `acb_theta_ctx_z_struct` containing all the necessary data to run the summation algorithm on a given vector z (provided that an element of type `acb_theta_ctx_tau_t` is also given.) In particular, it contains the values $\exp(2\pi i z_j)$ for all $1 \leq j \leq g$. If $g > 1$, it also contains the center of the ellipsoids used in summation algorithms at z .

void `acb_theta_ctx_tau_init`(`acb_theta_ctx_tau_t` ctx, int allow_shift, `slong` g)

Initializes `ctx` for use in dimension g . If `allow_shift` is nonzero, then additional fields in `ctx` are initialized to allow for the evaluation of theta functions $\theta_{a,0}$ for nonzero a .

void **acb_theta_ctx_tau_clear**(*acb_theta_ctx_tau_t* ctx)

Clears *ctx*.

void **acb_theta_ctx_z_init**(*acb_theta_ctx_z_t* ctx, *slong* g)

Initializes *ctx* for use in dimension *g*.

void **acb_theta_ctx_z_clear**(*acb_theta_ctx_z_t* ctx)

Clears *ctx*.

acb_theta_ctx_z_struct ***acb_theta_ctx_z_vec_init**(*slong* nb, *slong* g)

Returns a pointer to a vector of *nb* initialized elements of type *acb_theta_ctx_z_struct* in dimension *g*.

void **acb_theta_ctx_z_vec_clear**(*acb_theta_ctx_z_struct* *vec, *slong* nb)

Clears the elements of type *acb_theta_ctx_z_struct* pointed to by *vec* as well as the pointer itself.

void **acb_theta_ctx_exp_inv**(*acb_t* exp_inv, const *acb_t* exp, const *acb_t* x, int is_real, *slong* prec)

Given a complex value *x* and given *exp* containing $\exp(\pi i x)$, sets *exp_inv* to $\exp(-\pi i x)$.

This is computed by complex conjugation from *exp* if the boolean *is_real* is nonzero. Otherwise, it is computed by inverting *exp*, except if the result is indeterminate, in which case we recompute *exp_inv* from *x* directly.

void **acb_theta_ctx_sqr_inv**(*acb_t* sqr_inv, const *acb_t* inv, const *acb_t* sqr, int is_real, *slong* prec)

Given *inv* and *sqr* containing complex values $\exp(-\pi i x)$ and $\exp(2\pi i x)$ respectively, sets *sqr_inv* to $\exp(-2\pi i x)$.

This uses complex conjugation from *sqr* if *is_real* is nonzero, and otherwise a complex squaring from *inv*.

void **acb_theta_ctx_tau_set**(*acb_theta_ctx_tau_t* ctx, const *acb_mat_t* tau, *slong* prec)

Computes and stores in *ctx* the required data for the input matrix τ . The dimensions must match.

void **acb_theta_ctx_tau_dupl**(*acb_theta_ctx_tau_t* ctx, *slong* prec)

Modifies *ctx* in place to correspond to the matrix 2τ instead of τ . This is much cheaper than calling *acb_theta_ctx_tau_set()* again.

int **acb_theta_ctx_tau_overlaps**(const *acb_theta_ctx_tau_t* ctx1, const *acb_theta_ctx_tau_t* ctx2)

Returns true iff the data contained in *ctx1* and *ctx2* overlap in the sense of *acb_overlaps()*. This is only used for testing.

void **acb_theta_ctx_z_set**(*acb_theta_ctx_z_t* ctx, *acb_srcptr* z, const *acb_theta_ctx_tau_t* ctx_tau, *slong* prec)

Computes and stores in *ctx* the required data for the complex vector *z*. Here *ctx_tau* should contain context data for the matrix τ . The dimensions must match.

void **acb_theta_ctx_z_dupl**(*acb_theta_ctx_z_t* ctx, *slong* prec)

Modifies *ctx* in place to correspond to the pair $(2z, 2\tau)$ instead of (z, τ) . This is much cheaper than calling *acb_theta_ctx_z_set()* again.

void **acb_theta_ctx_z_add_real**(*acb_theta_ctx_z_t* res, const *acb_theta_ctx_z_t* ctx, const *acb_theta_ctx_z_t* ctx_real, *slong* prec)

Assuming that *ctx* and *ctx_real* correspond to pairs (z, τ) and (t, τ) respectively where *t* is a real vector, sets *res* to a valid context for the pair $(z + t, \tau)$.

void **acb_theta_ctx_z_common_v**(*arb_ptr* v, const *acb_theta_ctx_z_struct* *vec, *slong* nb, *slong* prec)

Given a vector *vec* of valid contexts for pairs $(z_1, \tau), \dots, (z_n, \tau)$, sets *v* to a valid ellipsoid center for use in *acb_theta_eld_set()* when running the summation algorithm for all these pairs, constructed using *arb_union()*.

```
int acb_theta_ctx_z_overlaps(const acb_theta_ctx_z_t ctx1, const acb_theta_ctx_z_t ctx2)
```

Returns true iff the data contained in *ctx1* and *ctx2* overlap in the sense of `acb_overlaps()`. This is only used for testing.

9.21.10 Summation algorithms

In this module, summation algorithms are mainly used for low to moderate precisions due to their higher asymptotic complexity. Since summations at low precisions are a key step in the quasi-linear algorithms, the summation functions below optimized in many ways and should already compare favorably to other software packages that evaluate theta functions.

We always assume in this section that the inputs (z, τ) have been reduced. In particular, this allows us to use only one ellipsoid when several vectors z are given.

After the relevant ellipsoid E has been computed, the main worker inside each version of the summation algorithm processes one line (i.e. 1-dimensional ellipsoid) in E . Before calling this worker, for fixed τ and z and fixed coordinates $n_1, \dots, n_{g-1}$ defining a line inside the ellipsoid, if $n_{\min}$ are $n_{\max}$ are the endpoints of the interval of allowed values for n_0 , we (efficiently) compute:

- the vector v_1 with entries $\exp(\pi i j^2 \tau_{0,0})$ for $n_{\min} \leq j \leq n_{\max}$,
- the vector v_2 with entries x^j for $n_{\min} \leq j \leq n_{\max}$, where

$$x = \exp(2\pi i z_0) \prod_{k=1}^{g-1} \exp(2\pi i n_k \tau_{0,k}),$$

- the cofactor $c \in \mathbb{C}$ given by

$$c = \prod_{k=1}^{g-1} \exp(2\pi i n_k z_k) \cdot \prod_{1 \leq j \leq k < g} \exp(\pi i (2 - \delta_{j,k}) n_j n_k \tau_{j,k}).$$

This allow us to use `acb_dot()` in the workers while maintaining reasonable memory costs, and to use an average of strictly less than two complex multiplications per lattice point as $R \rightarrow \infty$. Moreover, these multiplications are performed at only a fraction of the full precision for lattice points far from the ellipsoid center. The different versions of the summation algorithm rely on slightly different workers, so introducing a function pointer type is helpful to avoid code duplication.

When $g = 1$, the code does not rely on ellipsoids and worker functions, and calls `acb_modular_theta_sum()` from `acb_modular.h` instead.

type `acb_theta_sum_worker_t`

A function pointer type. A function *worker* of this type has the following signature:

```
void worker(acb_ptr th, acb_srcptr v1, acb_srcptr v2, const slong *prec, slong len, const acb_t c, const slong *coords, slong ord, slong g, slong prec, slong fullprec)
```

where:

- *th* denotes the output vector of theta values to which terms are added,
- *v1*, *v2* and *c* are precomputed as above,
- *prec* contains working precisions for each term $n_{\min} \leq j \leq n_{\max}$,
- $len = n_{\max} - n_{\min} + 1$ is the common length of *v1*, *v2* and *prec*,
- *coords* is $(n_{\min}, n_1, \dots, n_{g-1})$,
- *ord* is the maximal derivation order,

- *prec* is the working precision for this line inside the ellipsoid, and finally
- *fullprec* is the working precision for summing into *th*.

```
void acb_theta_sum_sqr_pow(acb_ptr *sqr_pow, const acb_mat_t exp_tau, const acb_theta_eld_t
                          E, slong prec)
```

For each $0 \leq j \leq g-1$, sets *sqr_pow*[*j*] to a vector of length $B_j + 1$ containing $e_j^{n^2}$ for $0 \leq n \leq B_j$, where B_j is an upper bound on the absolute values of *j*-th coordinates of points in *E* (obtained by *acb_theta_eld_box()*) and e_j is the *j*-th diagonal entry of *exp_tau*. The entries of *sqr_pow* need to be preallocated to the correct lengths.

```
void acb_theta_sum_work(acb_ptr th, slong len, acb_srcptr exp_z, acb_srcptr exp_z_inv, const
                      acb_mat_t exp_tau, const acb_mat_t exp_tau_inv, const acb_ptr
                      *sqr_pow, const acb_theta_eld_t E, slong ord, slong prec,
                      acb_theta_sum_worker_t worker)
```

Runs the summation algorithm on the ellipsoid *E*, assuming $g \geq 2$. The input is as follows:

- for each $1 \leq j \leq k \leq g$, the (j, k) entries of the matrices *exp_tau* and *exp_tau_inv* whose should contain $\exp(\pi i(2 - \delta_{j,k})\tau_{j,k})$ and its inverse, respectively.
- *E* is the ellipsoid we want to sum on.
- *sqr_pow* should be as output by *acb_theta_sum_sqr_pow()* on *exp_tau* and *E*.
- the vectors *exp_zs* and *exp_zs_inv* should have length *nb* times *g*. For each *z* stored in *zs*, the corresponding pieces of *exp_zs* and *exp_zs_inv* should contain $\exp(\pi i z_j)$ for $1 \leq j \leq g$ and their inverses, respectively.
- the parameters *len*, *ord* and the output vector *th* are passed to *worker* when processing each individual line in *E*.

The data associated with *zs* and τ is typically stored in contexts of type *acb_theta_ctx_tau_t* and *acb_theta_ctx_z_t* respectively. No error bound coming from the tail of the theta series is added.

```
void acb_theta_sum(acb_ptr th, const acb_theta_ctx_z_struct *vec, slong nb, const
                  acb_theta_ctx_tau_t ctx_tau, arb_srcptr distances, int all_a, int all_b, int
                  tilde, slong prec)
```

Evaluates theta functions at each of the *nb* pairs (z, τ) corresponding to a context stored in *vec* together with *ctx_tau* using summation. Precisely what we compute depends on the parameters *all_a*, *all_b* and *tilde*:

- if *all_a* is false (zero), we only compute $\theta_{a,b}$ for $a = 0$, otherwise *a* varies from 0 to $2^g - 1$.
- if *all_b* is false (zero), we only compute $\theta_{a,b}$ for $b = 0$, otherwise *b* varies from 0 to $2^g - 1$.
- if *tilde* is true (nonzero), then we compute $\tilde{\theta}_{a,b}$ instead of $\theta_{a,b}$.

In this function, the absolute error radius we add from the tail of the exponential series depend on *a*. The amount of precision added is controlled by *distances*, a vector of length 2^g (the same for all vectors *z*). One could for instance set *distances* to zero, or compute it as in *acb_theta_eld_distances()*, which makes sense when the different values of *z* differ by real vectors.

```
void acb_theta_sum_jet(acb_ptr th, const acb_theta_ctx_z_struct *vec, slong nb, const
                     acb_theta_ctx_tau_t ctx_tau, slong ord, int all_a, int all_b, slong prec)
```

Sets *th* to the vector of derivatives of theta functions up to total order *ord*, at each of the *nb* pairs (z, τ) specified by the contexts, using summation. Precisely which characteristics (a, b) we consider is controlled by the parameters *all_a* and *all_b* as in *acb_theta_sum()*.

9.21.11 AGM steps

The quasi-linear algorithm to evaluate theta functions uses the following *duplication formula*: for all $z, z' \in \mathbb{C}^g$ and $\tau \in \mathcal{H}_g$,

$$\theta_{a,0}(z, \tau) \theta_{a,0}(z', \tau) = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(z + z', 2\tau) \theta_{a+a',0}(z - z', 2\tau).$$

Applying the duplication formula amounts to taking a step in a (generalized) AGM sequence. Note that the formula still holds if we replace $\theta_{a,0}$ by the normalized version $\tilde{\theta}_{a,0}$.

This section gathers methods to apply duplication formulas efficiently while minimizing precision losses. In the case $z = z'$, the duplication formula is typically followed by an extraction of square roots using low-precision approximations to make the correct choice.

void **acb_theta_agm_sqrt**(*acb_ptr* res, *acb_srcptr* a, *acb_srcptr* roots, *slong* nb, *slong* prec)

Sets each of the nb entries of res to a square root of the corresponding entry of a . The choice of sign is determined by $roots$: each entry of res will overlap the corresponding entry of $roots$ but not its opposite. When this is not possible, we set the corresponding entry of res to the **acb_union()** of both square roots (when both overlap $roots$) or an indeterminate value (when none overlap $roots$).

void **acb_theta_agm_mul**(*acb_ptr* res, *acb_srcptr* a1, *acb_srcptr* a2, *slong* g, int all, *slong* prec)

For each $0 \leq k < 2^g$, sets the k -th entry of res to

$$\sum_{b \in \{0,1\}^g} a_{1,b} a_{2,b+k}$$

where addition is meant in $(\mathbb{Z}/2\mathbb{Z})^g$ (a bitwise xor). If all is nonzero, then we additionally compute, for each $1 \leq a \leq 2^g - 1$, the vector of length 2^g whose k -th entry contains

$$\sum_{b \in \{0,1\}^g} (-1)^{a^T b} a_{1,b} a_{2,b+k},$$

so res has total length 2^{2g} in that case.

Following [LT2016], we apply the Hadamard matrix twice with multiplications in-between. This causes precision losses when the absolute values of the entries in $a1$ and/or $a2$ are of different orders of magnitude. This function is faster when $a1$ and $a2$ are equal as pointers, as we can use squarings instead of multiplications.

void **acb_theta_agm_mul_tight**(*acb_ptr* res, *acb_srcptr* a0, *acb_srcptr* a, *arb_srcptr* d0, *arb_srcptr* d, *slong* g, int all, *slong* prec)

Assuming that $d0$ and d are obtained as the result of **acb_theta_eld_distances()** on pairs $(0, \tau)$ and (z, τ) respectively, performs the same computation as **acb_theta_agm_mul()** on the vectors $a0$ and a (and the parameter all) with a different management of error bounds. The resulting error bounds on res is tighter when the absolute value of a_k is roughly e^{-d_k} for each $0 \leq k < 2^g$, and similarly for $a0$ and $d0$, for instance when applying the duplication formula on normalized theta values.

We first compute m, ε such that the following holds: for each $0 \leq k < nb$, if d_k (resp. a_k) denotes the k -th entry of d (resp. a), then the absolute value of a_k is at most $m \cdot e^{-d_k}$ and the radius of the complex ball a_k is at most $\varepsilon \cdot e^{-d_k}$. We proceed similarly on $a0$ and $d0$ to obtain m_0, ε_0 . Then we call **acb_theta_agm_mul()** on the midpoints of $a0$ and a at a higher working precision, and finally add $2^g e^{-d_k} (m_0 \varepsilon + m \varepsilon_0 + \varepsilon \varepsilon_0)$ to the error bound on the k -th entry of res . This is valid because of the parallelogram identity: keeping notation from **acb_theta_eld_distances()**, for each $b \in \{0,1\}^g$, we have

$$\text{Dist}_\tau(-Y^{-1}y, \mathbb{Z}^g + \frac{b}{2})^2 + \text{Dist}_\tau(-Y^{-1}y, \mathbb{Z}^g + \frac{b+k}{2})^2 \leq \text{Dist}_\tau(-Y^{-1}y, \mathbb{Z}^g + \frac{k}{2})^2.$$

9.21.12 Quasilinear algorithms on reduced input

The general duplication formula specializes to the three following equalities:

$$\begin{aligned}\theta_{a,0}(z, \tau)^2 &= \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(2z, 2\tau) \theta_{a+a',0}(0, 2\tau), \\ \theta_{a,0}(0, \tau) \theta_{a,0}(z, \tau) &= \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(z, 2\tau) \theta_{a+a',0}(z, 2\tau), \\ \theta_{a,0}(0, \tau)^2 &= \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(0, 2\tau) \theta_{a+a',0}(0, 2\tau).\end{aligned}$$

Suppose that we wish to compute $\theta_{a,0}(0, \tau)$ for all $a \in \{0, 1\}^g$ and a reduced matrix $\tau \in \mathcal{H}_g$. Applying the last of the above duplication formulas n times, we reduce to evaluating $\theta_{a,0}(0, 2^n \tau)$. We expect that the absolute value of this complex number is roughly $\exp(-d^2)$ for $d = 2^n \text{Dist}_\tau(0, \mathbb{Z}^g + \frac{a}{2})$, where Dist_τ denotes the distance in $\mathbb{R}^g$ attached to the quadratic form πY . Provided that 2^n is roughly *prec*, we have to sum only $O_g(1)$ terms in the summation algorithm to evaluate $\theta_{a,0}(0, 2^n \tau)$ at “shifted absolute precision” *prec*, i.e. absolute precision $\text{prec} + d^2 / \log(2)$.

In order to recover $\theta_{a,0}(0, \tau)$, we then perform n AGM steps. *Assuming* that each $|\theta_{a,0}(0, 2^k \tau)|$ is indeed of the expected order of magnitude, we can ensure that the precision loss is $O_g(1)$ bits at each step in terms of shifted absolute precision, and we can make the correct choices of square roots at each step by computing low-precision approximations with the summation algorithm. However, depending on the choice of τ , this assumption may not always hold.

We make the following adjustments to make the algorithm work for all τ , for theta values at $z \neq 0$, and for all characteristics:

- If we discover that some value $\theta_{a,0}(0, 2^k \tau)$ is too small, we introduce an auxiliary real vector t . At each step, starting from $\theta_{a,0}(0, 2^{k+1} \tau)$, $\theta_{a,0}(2^{k+1}t, 2^{k+1} \tau)$ and $\theta_{a,0}(2^{k+2}t, 2^{k+1} \tau)$, we compute $\theta_{a,0}(2^k t, 2^k \tau)$ and $\theta_{a,0}(2^{k+1}t, 2^k \tau)$ using square roots (first formula above), then $\theta_{a,0}(0, 2^k \tau)$ using divisions (second formula). For a huge majority of such t , none of the values $\theta_{a,0}(2^k t, 2^k \tau)$ and $\theta_{a,0}(2^{k+1}t, 2^k \tau)$ are too small. In practice, we choose t at random and obtain a probabilistic algorithm with a negligible failure probability.
- When computing $\theta_{a,0}(z, \tau)$ for a nonzero z , we compute $\theta_{a,0}(0, 2^k \tau)$ and $\theta_{a,0}(2^k z, 2^k \tau)$ using the first and third formulas at each step.
- These two techniques can be combined by evaluating theta values at the six vectors $2^k v$ for $v = 0, t, 2t, z, z+t, z+2t$. Note that we only have to compute $\theta_{a,0}(2^k z, 2^k \tau)$ at the last step $k = 0$.
- To simplify the precision management, we use `acb_theta_agm_mul_tight()` and work with normalized theta values throughout, which also satisfy the duplication formulas.
- If the eigenvalues of Y have different orders of magnitude, then as we consider $\tau, 2\tau, 4\tau$, etc., the ellipsoids we would consider in the summation algorithm become very thin in one direction while still being thick in other directions. In such a case, we can rewrite theta values as a sum of $O(1)$ theta values in lower dimensions. This increases the efficiency of the algorithm while ensuring that the absolute precisions we consider are always of the order of *prec*.
- Finally, we note that the duplication formulas also have analogues for all theta values, not just $\theta_{a,0}$: for instance, we have

$$\theta_{a,b}(0, \tau)^2 = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} (-1)^{a'^T b} \theta_{a',0}(0, 2\tau) \theta_{a+a',0}(0, 2\tau).$$

We use those generalized formulas for the very last duplication step when needed.

We always assume in this section that the inputs (z, τ) have been reduced.

int **acb_theta_ql_nb_steps**(*slong* *pattern, const *acb_mat_t* tau, int cst, *slong* prec)

Determines how many duplication steps we should perform to evaluate theta functions at τ at precision *prec*, and at which steps we should fall back to lower dimensions, if any. The flag *cst* should be set to nonzero iff theta functions at $z \neq 0$ are to be computed.

The output is stored in *pattern*, a vector of length g . Roughly speaking, the j -th entry of *pattern* is a nonnegative integer m such that $2^m \gamma_j^2$ is of the order of *prec*, where γ_j denotes the j -th diagonal coefficient of the Cholesky matrix for πY . In other words, the ellipsoid we need to consider in the summation algorithm at $2^m \tau$ has width $O(1)$ in the direction of the j -th canonical basis vector in $\mathbb{R}^g$. Because τ is assumed to be reduced, we expect *pattern* to be a roughly decreasing vector.

If some entries of the Cholesky matrix are interminate or too extreme for a reasonable m to be computed, then the output is 0, and otherwise 1.

Modifying this function is the main way to tune the behavior of the quasi-linear algorithms to evaluate theta functions.

int **acb_theta_ql_lower_dim**(*acb_ptr* *new_zs, *acb_ptr* *cofactors, *slong* **pts, *slong* *nb, *arf_t* err, *slong* *fullprec, *acb_srcptr* z, const *acb_mat_t* tau, *arb_srcptr* distances, *slong* s, *ulong* a, *slong* prec)

Implements the dimension-lowering strategy for evaluating theta functions. The input is as follows:

- (z, τ) should be an exact element of $\mathbb{C}^g \times \mathcal{H}_g$ (ideally reduced).
- *distances* should be the output of **acb_theta_eld_distances()** on this pair.
- s should be an integer between 1 and $g - 1$; we will reduce the evaluation of theta functions from dimension g to dimension s .
- a should be an integer between 0 and $2^{g-s} - 1$ included; we only decompose $\tilde{\theta}_{a',0}(z, \tau)$ when the last $g - s$ bits of a' correspond to those of a .

We then proceed as follows:

- *fullprec* is set to the binary precision at which those theta values $\tilde{\theta}_{a',0}(z, \tau)$ should be computed. We take distances into account, so *fullprec* is *prec* plus additional guard bits derived from the maximum of the entries in *distances* corresponding to the possible characteristics a' .
- *R2* and *err* are set as in **acb_theta_sum_radius()** for this choice of *fullprec*. (*R2* is not part of the output.) Thus, $\tilde{\theta}_{a',0}(z, \tau)$ can be obtained by summing over an ellipsoid of squared radius *R2* and adding an error *err* coming from the tail. We do *not* compute that possibly huge ellipsoid.
- Let $n \in \mathbb{Z}^g + \frac{a'}{2}$ be a point in that ellipsoid. Write $a' = (a_0, a)$ and $n = (n_0, n_1)$ where $n_0 \in \mathbb{Z}^s + \frac{a_0}{2}$ and $n_1 \in \mathbb{Z}^{g-s} + \frac{a}{2}$. By the Pythagorean theorem, the possible values for n_1 all lie in an ellipsoid of radius *R2* in dimension $g - s$, whose Cholesky matrix is the lower-right part of a Cholesky matrix for πY . This new ellipsoid is meant to contain very few points. We list all possible values for $n_1 - \frac{a}{2}$ (which lies in $\mathbb{Z}^g$) in *pts*, and set *nb* to the number of those points. Note that *pts* will have to be freed by the user afterwards.
- For each $n_1 - \frac{a}{2}$ listed in *pts*, then the sum of the corresponding terms in the theta series is

$$e^{\pi i (n_1^T \tau_1 n_1 + 2n_1^T z_1 + \pi y_0^T Y_0 y_0 - \pi y^T Y y)} \tilde{\theta}_{a_0,0}(z_0 + x n_1, \tau_0).$$

where $\tau = \begin{pmatrix} \tau_0 & x \\ x^T & \tau_1 \end{pmatrix}$ and $z = (z_0, z_1)$. Thus, we allocate *new_zs* to contain *nb* vectors of length g and set the corresponding entry to $z_0 + x n_1$ (which is still exact). We also allocate *cofactors* to be a vector of length *nb* and set its corresponding entry to the above exponential factor. Both *new_zs* and *cofactors* will have to be freed by the user.

```
void acb_theta_ql_recombine(acb_ptr th, acb_srcptr th0, acb_srcptr cofactors, const slong *pts,
                           slong nb, const arf_t err, slong fullprec, slong s, ulong a, int all, slong
                           g, slong prec)
```

Performs the converse to `acb_theta_ql_lower_dim()`, namely recovers normalized theta values $\tilde{\theta}_{a',0}(z, \tau)$ from the output of `acb_theta_ql_lower_dim()` and theta values in dimension s . The input is as follows:

- `cofactors`, `pts`, `nb`, `err`, `fullprec`, `s`, `a`, `g`, `prec` should be as output by `acb_theta_ql_lower_dim()`.
- If `all` is true (nonzero), then `th0` should be a concatenation of `nb` vectors of length 2^{2s} containing $\tilde{\theta}_{a_0,b_0}(z_0, \tau_0)$ for all characteristics (a_0, b_0) in dimension s , where z_0 runs through `new_zs` as output by `acb_theta_ql_lower_dim()`, and τ_0 is defined as above. If `all` is false (zero), then `th0` should be a concatenation of `nb` vectors of length 2^s containing $\tilde{\theta}_{a_0,0}(z_0, \tau_0)$ only.

The output, stored in `th`, is either the vector containing $\tilde{\theta}_{a,b}(z, \tau)$ for all g -dimensional characteristics (a, b) (if `all` is true) or only $\tilde{\theta}_{a,0}(z, \tau)$ for all a (if `all` is false), where (z, τ) was the initial input to `acb_theta_ql_lower_dim()`.

```
int acb_theta_ql_setup(acb_ptr rts, acb_ptr rts_all, acb_ptr t, slong *guard, slong *easy_steps,
                      acb_srcptr zs, slong nb, const acb_mat_t tau, arb_srcptr distances, slong
                      nb_steps, int all, slong prec)
```

Sets up the structure of AGM steps to evaluate theta functions at the given `nb` pairs (z, τ) where z runs through `zs`, which are assumed to be exact and reduced, using `nb_steps` duplication steps. The parameters `nb` and `nb_steps` must be at least one, and `zs` must begin with the zero vector in $\mathbb{C}^g$. If (z, τ) are not exact, then the output is still mathematically correct, but NaN values or undesired precision losses might occur.

The rest of the input is as follows:

- `distances` should be the concatenation of `nb` vectors of length 2^g computed by `acb_theta_eld_distances()` for each pair (z, τ) .
- `nb_steps` should be the number of times we wish to apply the duplication formulas before falling back to either the summation algorithms or the dimension-lowering strategy.
- if `all` is nonzero, then we compute $\tilde{\theta}_{a,b}(z, \tau)$ for all characteristics (a, b) , and otherwise only $\tilde{\theta}_{a,0}(z, \tau)$.

The vectors `rts`, `rts_all`, `t`, and `easy_steps` should be preinitialized with lengths $2^g \times 3 \times nb \times nb\_steps$, $2^{2g} \times nb$ (only used if `all` is true), g and `nb` respectively, while `guard` is a pointer to one `slong`.

We proceed as follows. Initially, we work at a very low precision such as 8.

1. For each z , we use the summation algorithms to obtain approximations of $\tilde{\theta}_{a,b}(z, \tau)$ (if `all` is true) or $\tilde{\theta}_{a,0}(z, \tau)$ (if `all` is false), and store them in `rts_all` or `rts` respectively. We adjust the error bounds in terms of `distances`, so that the computed approximations do not contain zero with a good probability. If none of the computed approximations contains zero, it means that we can successfully apply the last (and simplest) duplication formula for the last step of the quasi-linear algorithm. In that case, we go on and compute approximations of $\tilde{\theta}_{a,0}(2^k z, 2^k \tau)$, for $k = 1, 2$, etc., up to `nb_steps`-1 or until one of the approximations we compute contains zero, taking distances into account at each step. We store the computed values in `rts`, and set the corresponding entry of `easy_steps` to be the number of steps for which the simplest duplication formula can be applied.
2. At that point, if the entries of `easy_steps` are all equal to `nb_steps`, we are done. Otherwise, we pick an auxiliary vector `t` at random. The 1st entry of `easy_steps`, corresponding to $z = 0$, is set to the minimal value in `easy_steps` (this is necessary to be able to apply the duplication formulas.)
3. For each z , if m denotes the corresponding entry of `easy_steps`, we use the summation algorithms to compute approximations of $\tilde{\theta}_{a,0}(2^k(z + t), 2^k \tau)$ and $\theta_{a,0}(2^k(z + 2t), 2^k \tau)$ for each k

between m and $nb\_steps-1$ included at low precision. (We only need the second vector for $k = 0$.) If one of these values contains zero, we restart step 3 with another t (we allow a small number of such retries, such as 4). We store those approximations in rts . If $k = 0$ and all is true, then the values we need are $\tilde{\theta}_{a,b}(z + 2t, \tau)$ for all (a, b) instead; those are stored in rts_all .

4. If no suitable t was found in step 4, then we double the working precision and go back to step 1. We allow this until the working precision reaches $prec$. After that, if t still cannot be found, then we declare failure and output 0. This should only happen with negligible probability for well-formed input. The output value is 1 if a suitable t was found.

Finally, we set *guard* to the total number of bits of precision we expect to lose when actually performing the suggested duplication steps later on.

```
void acb_theta_q1_exact(acb_ptr th, acb_srcptr zs, slong nb, const acb_mat_t tau, const slong
                        *pattern, int all, int shifted_prec, slong prec)
```

Runs the full quasi-linear algorithm to evaluate theta functions at the given nb pairs (z, τ) where z runs through zs , which are assumed to be exact and reduced. If (z, τ) are not exact, then the function is still correct, but NaN values or undesired precision losses might occur.

The output is either the collection of theta values $\tilde{\theta}_{a,b}(z, \tau)$ for all a, b or $\tilde{\theta}_{a,0}(z, \tau)$ for all a (depending on whether *all* is true or not) for each vector z in zs , and is stored in th . If *shifted_prec* is nonzero, then the precision to which these values are computed takes distances into account similarly to *acb_theta_sum()*.

The input *pattern* conditions how many duplication steps are performed and when to apply the dimension-lowering strategy (if at all). If zero duplication steps are needed, we call *acb_theta_sum()* directly. Otherwise, we call *acb_theta_q1_setup()*, which we expect to succeed with overwhelming probability. The initial theta values required in the duplication formulas are computed either by the summation algorithms or, if the dimension-lowering strategy is used, by calling *acb_theta_q1_lower_dim()*, making a recursive call to *acb_theta_q1_exact()* in a lower dimension but (possibly) a longer list of vectors zs , and finally recombining the values with *acb_theta_q1_recombine()*.

```
void acb_theta_q1_local_bound(arb_t c, arb_t rho, acb_srcptr z, const acb_mat_t tau, slong ord)
```

Sets c and ρ such that on every ball centered at (a point contained in) z of radius ρ , the functions $|\theta_{a,b}(\cdot, \tau)|$ for all characteristics (a, b) are uniformly bounded by c . The choice of ρ is tuned to get interesting upper bounds on derivatives of $\theta_{a,b}$ up to order ord in the context of finite differences (see *acb_theta_q1_jet_fd()* below). We always ensure that ρ is at most 1 and c is at least 1.

We proceed as follows. First, we compute c_0, c_1, c_2 such that for any choice of ρ , one can take $c = c_0 \exp((c_1 + c_2\rho)^2)$ above. We can take

$$c_0 = 2^g \prod_{j=0}^{g-1} (1 + 2\gamma_j^{-1}),$$

$$c_1 = \sqrt{\pi y^T Y^{-1} y},$$

$$c_2 = \sup_{\|x\|_\infty \leq 1} \sqrt{\pi x^T Y^{-1} x}.$$

One can easily compute an upper bound on c_2 from the Cholesky decomposition of πY^{-1} . We then look for a value of ρ that minimizes $\exp((c_1 + c_2\rho)^2)/\rho^{2ord+1}$, i.e. we set ρ to minimum of 1 and the positive root of $2c_2\rho(c_1 + c_2\rho) = 2ord + 1$.

```
void acb_theta_q1_jet_error(arb_ptr err, acb_srcptr z, const acb_mat_t tau, acb_srcptr dth,
                           slong ord, slong prec)
```

Assuming that *dth* contains (approximations of) the derivatives of a theta function $\theta_{a,b}$ up to total order $ord + 2$ at (z, τ) , sets *err* to a vector with the following property. Let (z_0, τ_0) be the midpoint

of (z, τ) , and let (z_1, τ_1) be any point inside the ball specified by the given z and τ . Then the vectors of derivatives of $\theta_{a,b}$ at (z_0, τ_0) and (z_1, τ_1) up to total order ord differ by at most err elementwise. This uses the heat equation and a Lipschitz-type inequality.

```
void acb_theta_q1_jet_fd(acb_ptr th, acb_srcptr zs, slong nb, const acb_mat_t tau, slong ord, int
                        all, slong prec)
```

Evaluates partial derivatives of theta functions $\theta_{a,b}$ with respect to z at the given nb points (z, τ) . The characteristic a varies from 0 to $2^g - 1$, and so does b if all is true (nonzero). If all is false, then we only consider $b = 0$. The result is a concatenation of nb times 2^{2g} (or 2^g) vectors of derivatives.

We rely on finite differences on the output of `acb_theta_q1_exact()`, as follows. Consider the Taylor expansion:

$$\theta_{a,b}(z + h, \tau) = \sum_{k \in \mathbb{Z}^g, k \geq 0} a_k h_0^{k_0} \cdots h_{g-1}^{k_{g-1}}.$$

If one chooses $h = h_n = (\varepsilon \zeta^{n_0}, \dots, \varepsilon \zeta^{n_{g-1}})$ where $\varepsilon > 0$ and ζ is a primitive m -th root of unity and lets n run through all vectors in $\{0, \dots, m-1\}^g$, then taking a discrete Fourier transform of the resulting values yields the individual Taylor coefficient for each derivation tuple that is bounded by $m-1$ elementwise. (A constant proportion, for fixed g , of this set consists of all tuples of total order at most $m-1$.) More precisely, fix $p \in \mathbb{Z}^g$. Then

$$\sum_{n \in \{0, \dots, m-1\}^g} \zeta^{-p^T n} \theta_{a,b}(z + h_n, \tau) = m^g \sum_{\substack{k \in \mathbb{Z}^g, k \geq 0, \\ k \equiv p \pmod{m}}} a_k \varepsilon^{|k|}.$$

We obtain an upper bound on the tail of this series from the Cauchy integration formula: if $|\theta_{a,b}(z, \tau)| \leq c$ uniformly on a ball of radius ρ centered in z for $\|\cdot\|_\infty$, then the sum is $m^g(a_p \varepsilon^{|p|} + T)$ with

$$|T| \leq 2cg \frac{\varepsilon^{|p|+m}}{\rho^m}.$$

Since we divide by $\varepsilon^{|p|}$ to get a_p , we add an error of $2cg\varepsilon^m/\rho^{m+|p|}$ to the result of the discrete Fourier transform.

The algorithm based on finite differences computes c and ρ using `acb_theta_q1_local_bound()`, chooses a suitable ε , strips (z, τ) of their error bounds, increases the working precision to account for division by $\varepsilon^{ord \cdot (ord+1)^g}$, calls `acb_theta_q1_exact()` on all the auxiliary points for $m = ord+1$ at a higher working precision, performs the relevant discrete Fourier transforms, and finally restores provably correct error bounds on the results using `acb_theta_q1_jet_error()` and derivatives to order $ord+2$ computed at low precision. This algorithm runs in quasi-linear time in $prec \cdot ord^g$ for any fixed g .

```
void acb_theta_q1_jet(acb_ptr th, acb_srcptr zs, slong nb, const acb_mat_t tau, slong ord, int all,
                    slong prec)
```

Same as `acb_theta_q1_jet_fd()`, but makes an automatic choice of algorithm between finite differences and direct summation depending on the working precision.

9.21.13 Reduction and main functions

void **acb_theta_jet_notransform**(*acb_ptr* th, *acb_srcptr* zs, *long* nb, const *acb_mat_t* tau, *long* ord, *ulong* ab, int all, int sqr, *long* prec)

Same as *acb_theta_jet()*, but does not attempt to reduce the input pairs (z, τ) . This function should only be used when the input is already known to be reduced.

Depending on the cases, we use the following formulas before calling *acb_theta_gl_jet()*:

- if *ord* is zero and *all* and *sqr* are both true, we use the duplication formula

$$\theta_{a,b}(z, \tau)^2 = \sum_{a' \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a',0}(0, 2\tau) \theta_{a+a',0}(2z, 2\tau).$$

- if *all* is false and *ab* is zero, we use the formula

$$\theta_{0,0}(z, \tau) = \sum_{a \in (\mathbb{Z}/2\mathbb{Z})^g} \theta_{a,0}(2z, 4\tau).$$

- if *all* is false and *ab* is not zero, we use the formula

$$\theta_{a,b}(z, \tau) = e^{\pi i(a^T \tau a/4 + a^T b + 2a^T z)} \theta_{0,0}(z + \tau \frac{a}{2} + \frac{b}{2}, \tau).$$

to fall back to the case where *ab* is zero.

int **acb_theta_reduce_tau**(*acb_ptr* new_zs, *acb_mat_t* new_tau, *fmpz_mat_t* mat, *acb_mat_t* N, *acb_mat_t* ct, *acb_ptr* exps, *acb_srcptr* zs, *long* nb, const *acb_mat_t* tau, *long* prec)

Reduces the input matrix τ under the action of the modular group $\mathrm{Sp}_{2g}(\mathbb{Z})$ and modifies the *nb* input vectors stored in *zs* according to the theta transformation formula.

The output is as follows:

- *new_tau* is the reduced matrix,
- *mat* is the symplectic matrix such that *new_tau* is the result of *mat* acting on *tau*,
- *ct* is the transpose of $(\gamma\tau + \delta)^{-1}$ where γ, δ are the lower $g \times g$ blocks of *mat*,
- *N* is the matrix $i\pi\gamma \cdot ct$ that appears in the transformation formula,
- and finally *new_zs* is the list of *nb* vectors in $\mathbb{C}^g$ corresponding to the elements in *zs* multiplied by *ct* on the left.

If reduction was unsuccessful (usually indicating that the input is malformed or that the working precision is insufficient to detect that Y is positive definite), the return value is 0 and the above output is left undefined. Otherwise, the return value is 1.

int **acb_theta_reduce_z**(*acb_ptr* new_zs, *arb_ptr* rs, *acb_ptr* cs, *acb_srcptr* zs, *long* nb, const *acb_mat_t* tau, *long* prec)

Reduces the *nb* vectors stored in *zs* in the context of evaluating theta functions with *tau* as a second argument.

The output vectors *new_zs*, *rs* and *cs* should have lengths *nb* times *g*, *nb* times *g*, and *nb* respectively. For a given vector z appearing in *zs*, we round $Y^{-1}y$ to the nearest even, integral vector r , and store it in *rs*. Then, we consider the vector $z - \tau r$, subtract the nearest even integral vector from its real part, and store the result z' into *new_zs*. For all characteristics a, b , we have

$$\theta_{a,b}(z', \tau) = e^{-i\pi r^T(z+z')} \theta_{a,b}(z, \tau).$$

Finally, we store this exponential factor as the corresponding entry of *cs*.

If rounding the imaginary part to integers does not succeed due to extreme values, then the return value is 0 and the output vectors are left undefined. Otherwise, the return value is 1.

We then assemble the main function `acb_theta_jet()` as follows. If `acb_theta_reduce_tau()`, then `acb_theta_reduce_z()` succeed, we call `acb_theta_jet_notransform()` on the reduced pairs (z, τ) . We finally apply the transformation formula with the help of `acb_theta_char_table()` and `acb_siegel_kappa()`. If reduction does not succeed, then the output is set to indeterminate values.

9.21.14 Dimension 2 specifics

In the $g = 2$ case, one can use theta functions to evaluate many fundamental Siegel modular forms. This section contains methods to do so, in analogy with `acb_modular_delta()` or `acb_modular_eisenstein()` when $g = 1$.

We use the following notation. Fix $k, j \geq 0$. A Siegel modular form of weight $\det^k \otimes \text{Sym}^j$ is by definition an analytic function $f : \mathcal{H}_g \rightarrow \mathbb{C}_j[X]$ (the vector space of polynomials of degree at most j) such that for any $\tau \in \mathcal{H}_g$ and $m \in \text{Sp}_4(\mathbb{Z})$, we have

$$f((\alpha\tau + \beta)(\gamma\tau + \delta)^{-1}) = \det(\gamma\tau + \delta)^k \cdot \text{Sym}^j(\gamma\tau + \delta)(f(\tau)).$$

Here $\alpha, \beta, \gamma, \delta$ are the $g \times g$ blocks of m , and the notation $\text{Sym}^j(r)$ where $r = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \text{GL}_2(\mathbb{C})$ stands for the map

$$P(X) \mapsto (bX + d)^j P\left(\frac{aX+c}{bX+d}\right).$$

For a nonzero f to exist, j must be even.

Siegel modular forms generate a bi-graded ring which is not finitely generated. However, if we relax the definition of a Siegel modular form and allow them to have a pole along the diagonal $\mathcal{H}_1^2 = \left\{ \begin{pmatrix} \tau_1 & 0 \\ 0 & \tau_2 \end{pmatrix} \right\} \subset \mathcal{H}_2$ of a certain order (depending on the weight), we indeed find a finitely generated ring corresponding to classical “covariants” of a binary sextic. Historically, covariants are classified in terms of their degree k and index j , corresponding to Siegel modular functions of weight $\det^{k-j/2} \otimes \text{Sym}^j$. See [CFG2017] for more details on the correspondence between modular forms and covariants.

```
void acb_theta_g2_detk_symj(acb_poly_t res, const acb_mat_t m, const acb_poly_t f, slong k,
                           slong j, slong prec)
```

Sets *res* to $\det(m)^k \text{Sym}^j(m)(f)$. The polynomial f should be of degree at most j (any coefficients of larger degree are ignored).

```
void acb_theta_g2_transvectant(acb_poly_t res, const acb_poly_t g, const acb_poly_t h, slong m,
                              slong n, slong k, int lead, slong prec)
```

Sets *res* to the k -th transvectant of the polynomials g and h of degrees m and n . If *lead* is true (nonzero), we instead set *res* to the constant polynomial containing the leading coefficient of this transvectant.

Considering g and h as homogeneous polynomials of degree m (resp. n) in x_1, x_2 , this sets *res* to (the leading coefficient of) the polynomial

$$(g, h)_k := \frac{(m-k)!(n-k)!}{m!n!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} \frac{\partial^k g}{\partial x_1^{k-j} \partial x_2^j} \frac{\partial^k h}{\partial x_1^j \partial x_2^{k-j}}.$$

Any coefficients of g or h of larger degree than m (resp. n) are ignored.

slong **acb_theta_g2_character**(const *fmpr_mat_t* mat)

Returns the value in $\mathbb{Z}/2\mathbb{Z}$ (0 or 1) of the unique nontrivial character of $\mathrm{Sp}_4(\mathbb{Z})$ at *mat*, following [CFG2019], §12.

void **acb_theta_g2_even_weight**(*acb_t* psi4, *acb_t* psi6, *acb_t* chi10, *acb_t* chi12, *acb_srcptr* th2, *slong* prec)

Sets *psi4*, *psi6*, *chi10*, and *chi12* to the value of the Eisenstein series ψ_4 , ψ_6 and cusp forms χ_{10} , χ_{12} corresponding to the given vector *th2* of squared theta values (of length 16).

We use the formulas from §7.1 in [Str2014], with the following normalizations:

$$\psi_4 = h_4/4, \quad \psi_6 = h_6/4, \quad \chi_{10} = -2^{-12}h_{10}, \quad \chi_{12} = 2^{-15}h_{12}.$$

We warn that χ_{10} and χ_{12} differ from the classical notation of Igusa [Igu1979] by scalar factors. Writing $\tau = \begin{pmatrix} \tau_1 & \tau_2 \\ \tau_2 & \tau_3 \end{pmatrix}$ and $q_j = \exp(2\pi i\tau_j)$, the Fourier expansions of these modular forms begin as follows:

$$\begin{aligned} \psi_4(\tau) &= 1 + 240(q_1 + q_3) + \cdots \\ \psi_6(\tau) &= 1 - 504(q_1 + q_3) + \cdots \\ \chi_{10}(\tau) &= (q_2 - 2 + q_2^{-1})q_1q_3 + \cdots \\ \chi_{12}(\tau) &= (q_2 + 10 + q_2^{-1})q_1q_3 + \cdots \end{aligned}$$

void **acb_theta_g2_chi5**(*acb_t* res, *acb_srcptr* th, *slong* prec)

Sets *res* to the value of $\chi_5 = -2^{-6} \prod_{(a,b) \text{ even}} \theta_{a,b}$ corresponding to the given theta values *th*. The form χ_5 is a Siegel cusp form with character: see [CFG2019] for more details.

void **acb_theta_g2_chi35**(*acb_t* res, *acb_srcptr* th, *slong* prec)

Sets *res* to the value of the cusp form χ_{35} corresponding to the vector of theta values *th*. The form χ_{35} is the unique scalar-valued Siegel modular form of weight $\det^{35} \otimes \mathrm{Sym}^0$ up to scalars, and is normalized as follows:

$$\chi_{35}(\tau) = q_1^2 q_3^2 (q_1 - q_3)(q_2 - q_2^{-1}) + \cdots$$

An explicit formula for χ_{35} in terms of theta values is given in [Bol1887]. See also [Mum1984], Prop. 6.2 p. 98 for how to translate Bolza's notation in terms of theta characteristics.

void **acb_theta_g2_chi3_6**(*acb_poly_t* res, *acb_srcptr* dth, *slong* prec)

Sets *res* to the value of the vector-valued cusp form with character $\chi_{6,3}$ of weight $\det^3 \otimes \mathrm{Sym}^6$ corresponding to the given values of *dth*, computed as in e.g. *acb_theta_jet()* with *ord* = 1 and *all* set to true. We have by [CFG2017]:

$$\chi_{3,6}(\tau) = \frac{1}{64\pi^6} \prod_{(a,b) \text{ odd}} \left(\frac{\partial \theta_{a,b}}{\partial z_1}(0, \tau)x_1 + \frac{\partial \theta_{a,b}}{\partial z_2}(0, \tau)x_2 \right).$$

void **acb_theta_g2_sextic_chi5**(*acb_poly_t* res, *acb_t* chi5, const *acb_mat_t* tau, *slong* prec)

Sets *res* and *chi5* to the values of $\chi_{-2,6}$ and χ_5 at τ . We reduce τ to the Siegel fundamental domain, call *acb_theta_jet_notransform()*, then apply the transformation formula for Siegel modular forms (which is simpler than the transformation formula on derivatives of theta functions.) Under the correspondence between Siegel modular functions and covariants of binary sextics, $\chi_{-2,6}$ corresponds to the binary sextic itself, hence the name.

void **acb_theta_g2_covariants**(*acb_poly_struct* *res, const *acb_poly_t* f, int lead, *slong* prec)

Sets *res* to the vector of 26 generators of the ring of covariants (or their leading coefficients, when *lead* is true) evaluated at the sextic *f* (any terms of degree > 6 are ignored). We use the following ordering:

0. $C_{1,6} = f$
1. $C_{2,0} = 60(f, f)_6$
2. $C_{2,4} = 75(f, f)_4$
3. $C_{2,8} = 90(f, f)_2$
4. $C_{3,2} = 30(f, C_{2,4})_4$
5. $C_{3,6} = 30(f, C_{2,4})_2$
6. $C_{3,8} = 6(f, C_{2,4})_1$
7. $C_{3,12} = 6(f, C_{2,8})_1$
8. $C_{4,0} = 2(C_{2,4}, C_{2,4})_4$
9. $C_{4,4} = 30(f, C_{3,2})_2$
10. $C_{4,6} = 6(f, C_{3,2})_1$
11. $C_{4,10} = 2(C_{2,8}, C_{2,4})_1$
12. $C_{5,2} = (C_{2,4}, C_{3,2})_2$
13. $C_{5,4} = \frac{2}{5}(C_{2,4}, C_{3,2})_1$
14. $C_{5,8} = 2(C_{2,8}, C_{3,2})_1$
15. $C_{6,0} = 2(C_{3,2}, C_{3,2})_2$
16. $C_{6,6}^{(1)} = \frac{2}{5}(C_{3,6}, C_{3,2})_1$
17. $C_{6,6}^{(2)} = \frac{8}{3}(C_{3,8}, C_{3,2})_2$
18. $C_{7,2} = 30(f, C_{3,2}^2)_4$
19. $C_{7,4} = 12(f, C_{3,2}^2)_3$
20. $C_{8,2} = \frac{2}{5}(C_{2,4}, C_{3,2}^2)_3$
21. $C_{9,4} = 4(C_{3,8}, C_{3,2}^2)_4$
22. $C_{10,0} = 20(f, C_{3,2}^3)_6$
23. $C_{10,2} = \frac{6}{5}(f, C_{3,2}^3)_5$
24. $C_{12,2} = \frac{8}{5}(C_{3,8}, C_{3,2}^3)_6$
25. $C_{15,0} = \frac{1}{30000}(C_{3,8}, C_{3,2}^4)_8$.

The scalar factors are chosen so that when evaluated at a formal sextic $f = \sum a_i x_1^{6-i} x_2^i$, the covariants are integral and primitive as multivariate polynomials in $a_0, \dots, a_6, x_1, x_2$.

9.21.15 Tests

```
./build/acb_theta/test/main acb_theta_sp2gz_set_blocks
```

Generates a random $2g \times 2g$ matrix, calls `sp2gz_set_blocks()` on its four $g \times g$ windows, and checks that the result equals the original matrix.

```
./build/acb_theta/test/main acb_theta_sp2gz_is_correct
```

Checks that the return value of `sp2gz_is_correct()` is 1 on matrices generated by `sp2gz_j()`, `sp2gz_block_diag()`, `sp2gz_trig()` and `sp2gz_fundamental()`, and 0 on the identity matrix if it is not square of even size.

```
./build/acb_theta/test/main acb_theta_sp2gz_inv
```

Checks that the result of `sp2gz_inv()` agrees with `fmpz_mat_inv()` on random input.

```
./build/acb_theta/test/main acb_theta_sp2gz_decompose
```

Checks that the result of `sp2gz_decompose()` on random input only consists of symplectic matrices of the allowed types, and that their product equals the original matrix.

```
./build/acb_theta/test/main acb_theta_siegel_cocycle
./build/acb_theta/test/main acb_theta_siegel_transform
```

Checks that the chain rules hold: if $m'' = m'm$ is a product of two symplectic matrices and $\tau \in \mathcal{H}_g$, then $\gamma''\tau + \delta'' = (\gamma'\tau' + \delta')(\gamma\tau + \delta)$ where $\tau' = m\tau$, and $m''\tau = m'\tau'$. These quantities are computed using `acb_siegel_cocycle()` and `acb_siegel_transform()`.

```
./build/acb_theta/test/main acb_theta_siegel_reduce
```

Generates an input matrix τ at a working precision that is not too low compared to the size of its coefficients, and calls `acb_siegel_reduce()`. Checks that the resulting matrix m is symplectic and that $m\tau$ is reduced with a tolerance of 2^{-10} using `acb_siegel_is_reduced()`.

```
./build/acb_theta/test/main acb_theta_siegel_is_reduced
```

Checks that `acb_siegel_is_reduced()` returns 1 on the matrix iI_g , but 0 on other matrices specially constructed to not be reduced.

```
./build/acb_theta/test/main acb_theta_siegel_kappa
```

Checks that the results of `acb_siegel_kappa()` are compatible under matrix multiplication, and when varying the `sqr` parameter.

```
./build/acb_theta/test/main acb_theta_char_set_slong_vec
```

Checks that the functions `acb_theta_char_set_slong_vec()` and `acb_theta_char_bit()` are inverses of each other on random input.

```
./build/acb_theta/test/main acb_theta_char_dot
```

Checks that dot products computed by `acb_theta_char_dot()`, and `acb_theta_char_dot_slong()` agree on random input.

```
./build/acb_theta/test/main acb_theta_char_table
```

Checks that the a part of characteristics remains invariant when calling `acb_theta_char_table()` on trigonal symplectic matrices as in `sp2gz_trig()`.

```
./build/acb_theta/test/main acb_theta_char_shuffle
```

Checks that calling `acb_theta_char_shuffle()` on a random matrix and its inverse yields inverse transformations.

```
./build/acb_theta/test/main acb_theta_jet_tuples
```

For random g and ord , generates the list of derivation tuples using `acb_theta_jet_tuples()`, picks an index i at random, and checks that the result of `acb_theta_jet_index()` on the i -th tuple is indeed i .

```
./build/acb_theta/test/main acb_theta_jet_mul
```

Checks that the results of `acb_theta_jet_mul()` agrees with the result of `fmpz_mpoly_mul()` on any input with integral entries.

```
./build/acb_theta/test/main acb_theta_jet_compose
```

Checks that the chain rule holds: if $N_3 = N_2N_1$, then applying `acb_theta_jet_compose()` with N_2 , then N_1 corresponds to applying `acb_theta_jet_compose()` with N_3 directly.

```
./build/acb_theta/test/main acb_theta_eld_points
```

Generates a random ellipsoid E using `acb_theta_eld_set()`. Then, generates random points pt : if pt is in E according to `acb_theta_eld_contains()`, then pt must appear in the list of points and be contained in exactly one child of E ; otherwise the norm of pt according to the chosen Cholesky matrix must be at least the radius of E .

```
./build/acb_theta/test/main acb_theta_eld_border
```

Generates a random ellipsoid E , computes its border using `acb_theta_eld_border()`, and checks that none of these border points lie in E .

```
./build/acb_theta/test/main acb_theta_eld_distances
```

Checks that when $y = Y \frac{a}{2}$ for some theta characteristic a , the result of `acb_theta_eld_distances()` on (z, τ) contains zero in its a -th entry.

```
./build/acb_theta/test/main acb_theta_sum_radius  
./build/acb_theta/test/main acb_theta_sum_jet_radius
```

Generates a reduced matrix τ in $\mathcal{H}_g$ and vector $z \in \mathbb{C}^g$, calls the tested function, constructs the associated ellipsoid E , and checks that the sums of absolute values of terms of the theta series (differentiated or not) on the border of E is at most the specified bound.

```
./build/acb_theta/test/main acb_theta_ctx_exp_inv  
./build/acb_theta/test/main acb_theta_ctx_sqr_inv
```

Checks that the output of both functions, even at low precision, contains the expected values and are never indeterminate.

```
./build/acb_theta/test/main acb_theta_ctx_tau_dupl  
./build/acb_theta/test/main acb_theta_ctx_z_dupl  
./build/acb_theta/test/main acb_theta_ctx_z_add_real
```

Checks that the result of those functions overlaps with new contexts constructed with `acb_theta_ctx_tau_set()` and/or `acb_theta_ctx_z_set()` at 2τ , $(2z, 2\tau)$, and $(z+t, \tau)$ respectively.

```
./build/acb_theta/test/main acb_theta_sum
```

Checks that the results of `acb_theta_sum()` agree with `acb_modular_theta()` as follows: if the input matrix τ is diagonal with coefficients $\tau_0, \dots, \tau_{g-1}$, then for all characteristics (a, b) and vectors z , we have

$$\theta_{a,b}(z, \tau) = \prod_{j=0}^{g-1} \theta_{a_j, b_j}(z_j, \tau_j).$$

```
./build/acb_theta/test/main acb_theta_sum_jet
```

Checks that the results of `acb_theta_sum_jet()` agree with `acb_modular_theta_jet()` as follows: if the input matrix τ is diagonal with coefficients $\tau_0, \dots, \tau_{g-1}$, then for all characteristics (a, b) , any vector z , and any derivation tuple $(k_0, \dots, k_{g-1})$, we have

$$\frac{\partial^{|k|} \theta_{a,b}}{\partial z_0^{k_0} \dots \partial z_{g-1}^{k_{g-1}}}(z, \tau) = \prod_{j=0}^{g-1} \frac{\partial^{k_j} \theta_{a_j, b_j}}{\partial z_j^{k_j}}(z_j, \tau_j).$$

```
./build/acb_theta/test/main acb_theta_agm_sqrt
```

Generates a random complex number t , sets `roots` to a low-precision rounding of t (possibly containing zero), and sets a to the square of t . Checks that the result of `acb_theta_agm_sqrt()` on this input is finite, contains t , and that the precision loss is small when `roots` does not contain zero.

```
./build/acb_theta/test/main acb_theta_agm_mul
```

Checks that the duplication formula holds: the result of `acb_theta_agm_mul()` on vectors containing $\theta_{0,b}(0, \tau)$ and $\theta_{0,b}(z, \tau)$ for all $b \in \{0, 1\}^g$ and any choice of (z, τ) contains the squared theta values $\theta_{0,b}^2(2z, 2\tau)$.

```
./build/acb_theta/test/main acb_theta_agm_mul_tight
```

Generates random τ and z at working precision `prec`, computes the associated vectors of distances `d0` and `d` using `acb_theta_eld_distances()`, and constructs vectors `a0` and `a` with entries of the form $x e^{-t}$ where x is uniformly random with $|x| \leq 1$ (generated by `acb_urandom()`) and t is the corresponding entry of `d0` (resp. `d`). Calls `acb_theta_agm_mul_tight()` at a lower precision `mprec`. For each $0 \leq k < 2^g$, checks that the absolute value of the k -th entry of the result `res` is at most e^{-d_k} , and that the error bound on that entry is at most $2^{-mprec+\delta} e^{-d_k}$ for a reasonable value of δ (e.g. 25).

```
./build/acb_theta/test/main acb_theta_ql_lower_dim
```

Checks that applying `acb_theta_ql_lower_dim()`, computing lower-dimensional theta values using `acb_theta_sum()`, then recombining them using `acb_theta_ql_recombine()` agrees with a call to summation algorithms in dimension g .

```
./build/acb_theta/test/main acb_theta_ql_setup
```

Calls `acb_theta_ql_setup()` on random input, checks that the output value is 1, then checks that all the computed low-precision approximations of theta values are indeed nonzero.

```
./build/acb_theta/test/main acb_theta_ql_exact
```

Checks that the result of `acb_theta_ql_exact()` agrees with `acb_theta_sum()` on random input, and is not indeterminate.

```
./build/acb_theta/test/main acb_theta_ql_local_bound
```

Generates random (z, τ) at a working precision that is not too low and calls `acb_theta_ql_local_bound()` to compute the bounds c and ρ . Checks that they are finite and that their definition is satisfied by sampling theta values on the corresponding neighborhood of z at low precisions with `acb_theta_sum()`.

```
./build/acb_theta/test/main acb_theta_ql_jet_error
```

Generates two pairs (z_1, τ_1) and (z_2, τ_2) close to each other but not overlapping, sets (z, τ) to be their reunion (as complex balls on each coefficient), and calls `acb_theta_ql_jet_error()` on (z, τ) for some choice of derivation order. The difference between the results of `acb_theta_sum_jet()` on (z_1, τ_1) and (z_2, τ_2) must then be at most two times the computed error.

```
./build/acb_theta/test/main acb_theta_ql_jet_fd
./build/acb_theta/test/main acb_theta_jet_notransform
```

Checks that the output of these functions agrees with `acb_theta_sum_jet()` on random input, and is not indeterminate.

```
./build/acb_theta/test/main acb_theta_reduce_z
```

Checks that on random input, the entries of the output r always consist of even integers, and the imaginary part of new_zs is indeed $y - Yr$.

```
./build/acb_theta/test/main acb_theta_jet
```

Checks that the output of `acb_theta_jet()` and `acb_theta_jet_notransform()` agree. The matrix τ is chosen to be a priori non-reduced but still reasonably close to the reduced domain.

```
./build/acb_theta/test/main acb_theta_g2_detk_symj
```

Checks that the chain rule holds for the representation $\det^k \text{Sym}^j$ of $\text{GL}_2(\mathbb{C})$ as computed by `acb_theta_g2_detk_symj()`.

```
./build/acb_theta/test/main acb_theta_g2_transvectant
```

Checks that on any sextic polynomial $f = \sum_{j=0}^6 a_j x^{6-j}$, the transvectant $(f, f)_6$ as computed by `acb_theta_g2_transvectant()` is $-3a_2^3 + 8a_2a_4 - 20a_1a_5 + 120a_0a_6$.

```
./build/acb_theta/test/main acb_theta_g2_character
```

Checks that the results of `acb_theta_g2_character()` and `acb_siegel_kappa2()` for $g = 2$ are compatible, using the fact that the product χ_5 of the ten even theta constants is a Siegel modular form with character.

```
./build/acb_theta/test/main acb_theta_g2_even_weight
./build/acb_theta/test/main acb_theta_g2_chi35
```

Checks that the computed values of those modular forms are either invariant, multiplied by ± 1 , or by a power of i (depending on the weight modulo 4) when applying `acb_theta_char_shuffle()` on any input vector. The multiplicative factor is given by `acb_siegel_kappa2()`.

```
./build/acb_theta/test/main acb_theta_g2_chi5
```

Checks that the result of `acb_theta_g2_chi5()` squares to χ_{10} on any input vector.

```
./build/acb_theta/test/main acb_theta_g2_chi3_6
```

Checks that the product $\chi_{8,6} = \chi_5\chi_{3,6}$, computed using `acb_theta_g2_chi5()` and `acb_theta_g2_chi3_6()`, indeed defines a modular form of weight $\det^8 \text{Sym}^6$ by evaluating both sides of the transformation law on random input.

```
./build/acb_theta/test/main acb_theta_g2_sextic_chi5
```

Checks that the discriminant of the computed sextic on a random matrix τ is $2^{12}\chi_{10}(\tau)$, as computed by `acb_theta_g2_even_weight()`, and also that $\chi_5(\tau)^2 = \chi_{10}(\tau)$.

```
./build/acb_theta/test/main acb_theta_g2_covariants
```

Checks that the output of `acb_theta_g2_covariants()` agrees with that of `acb_theta_g2_even_weight()` using the relation $20\psi_4 = -C_{2,0} + 3C_{4,0}$. Also checks that each covariant, when evaluated on the result of `acb_theta_g2_sextic_chi5()`, defines a Siegel modular function of the correct weight by evaluating the transformation law, and that covariants take integral values when the input polynomial is integral.

9.21.16 Profiling

```
./build/acb_theta/profile/p-sum g prec all_a all_b tilde skew
```

Profiles `acb_theta_sum()` for the given parameters on a random pair (z, τ) generated by `acb_siegel_randtest_compact()` and `acb_siegel_randtest_vec_reduced()`, except that the last line and last column of τ are multiplied by the integer *skew* (which should be at least 1).

This is to observe how the performance of different steps in the summation algorithm (computing distances, setting contexts, and summation itself) behaves, especially as *g* grows and at low precisions.

```
./build/acb_theta/profile/p-ql_setup g nb_steps all skew
```

Profiles `acb_theta_ql_setup()` for the given parameters on random pairs $(0, \tau)$ and (z, τ) , generated using *skew* as above.

This is to check that `acb_theta_ql_setup()` continues to succeed and to be cheap even as *prec* or *nb_steps* grow, thus validating how precisions are managed internally.

```
./build/acb_theta/profile/p-ql_exact g pmin pmax cst s exp
```

Profiles `acb_theta_ql_exact()` on random pairs $(0, \tau)$ and (z, τ) in dimension *g*. The input is generated by `acb_siegel_randtest_compact()` and `acb_siegel_randtest_vec_reduced()`, except that the last *s* lines and columns of τ are multiplied by 2^{exp} .

For growing precisions from *pmin* to *pmax* (by exponential steps), we call `acb_theta_ql_nb_steps()` to get a suggested *pattern*, then measure how `acb_theta_ql_exact()` performs on this pattern and neighboring ones. Ideally, the suggested pattern should be the optimal one, indicating that `acb_theta_ql_nb_steps()` gives an optimal answer for the given input.

```
./build/acb_theta/profile/p-ql_jet_fd g prec ord all
```

Profiles `acb_theta_ql_jet_fd()` and compares it with `acb_theta_sum_jet()` for the given parameters on random input (z, τ) generated by `acb_siegel_randtest_compact()` and `acb_siegel_randtest_vec_reduced()`. This is to gain information on which algorithm `acb_theta_ql_jet()` should use.

9.22 `acb_dirichlet.h` – Dirichlet L-functions, Riemann zeta and related functions

This module allows working with values of Dirichlet characters, Dirichlet L-functions, and related functions. A Dirichlet L-function is the analytic continuation of an L-series

$$L(s, \chi) = \sum_{k=1}^{\infty} \frac{\chi(k)}{k^s}$$

where $\chi(k)$ is a Dirichlet character. The trivial character $\chi(k) = 1$ gives the Riemann zeta function. Working with Dirichlet characters is documented in [dirichlet.h – Dirichlet characters](#).

The code in other modules for computing the Riemann zeta function, Hurwitz zeta function and polylogarithm will possibly be migrated to this module in the future.

9.22.1 Roots of unity

type `acb_dirichlet_roots_struct`

type `acb_dirichlet_roots_t`

void `acb_dirichlet_roots_init`(*acb_dirichlet_roots_t* roots, *ulong* n, *ulong* num, *ulong* prec)

Initializes *roots* with precomputed data for fast evaluation of roots of unity $e^{2\pi i k/n}$ of a fixed order *n*. The precomputation is optimized for *num* evaluations.

For very small *num*, only the single root $e^{2\pi i/n}$ will be precomputed, which can then be raised to a power. For small *prec* and large *n*, this method might even skip precomputing this single root if it estimates that evaluating roots of unity from scratch will be faster than powering.

If *num* is large enough, the whole set of roots in the first quadrant will be precomputed at once. However, this is automatically avoided for large *n* if too much memory would be used. For intermediate *num*, baby-step giant-step tables are computed.

void `acb_dirichlet_roots_clear`(*acb_dirichlet_roots_t* roots)

Clears the structure.

void `acb_dirichlet_root`(*acb_t* res, const *acb_dirichlet_roots_t* roots, *ulong* k, *ulong* prec)

Computes $e^{2\pi i k/n}$.

9.22.2 Truncated L-series and power sums

void `acb_dirichlet_powsum_term`(*acb_ptr* res, *arb_t* log_prev, *ulong* *prev, const *acb_t* s, *ulong* k, int integer, int critical_line, *ulong* len, *ulong* prec)

Sets *res* to $k^{-(s+x)}$ as a power series in *x* truncated to length *len*. The flags *integer* and *critical_line* respectively specify optimizing for *s* being an integer or having real part 1/2.

On input *log_prev* should contain the natural logarithm of the integer at *prev*. If *prev* is close to *k*, this can be used to speed up computations. If $\log(k)$ is computed internally by this function, then *log_prev* is overwritten by this value, and the integer at *prev* is overwritten by *k*, allowing *log_prev* to be recycled for the next term when evaluating a power sum.

void `acb_dirichlet_powsum_sieved`(*acb_ptr* res, const *acb_t* s, *ulong* n, *ulong* len, *ulong* prec)

Sets *res* to $\sum_{k=1}^n k^{-(s+x)}$ as a power series in *x* truncated to length *len*. This function stores a table of powers that have already been calculated, computing $(ij)^r$ as $i^r j^r$ whenever $k = ij$ is composite. As a further optimization, it groups all even *k* and evaluates the sum as a polynomial in $2^{-(s+x)}$. This scheme requires about $n/\log n$ powers, $n/2$ multiplications, and temporary storage of $n/6$ power series. Due to the extra power series multiplications, it is only faster than the naive algorithm when *len* is small.

void **acb_dirichlet_powsum_smooth**(*acb_ptr* res, const *acb_t* s, *ulong* n, *slong* len, *slong* prec)

Sets *res* to $\sum_{k=1}^n k^{-(s+x)}$ as a power series in x truncated to length *len*. This function performs partial sieving by adding multiples of 5-smooth k into separate buckets. Asymptotically, this requires computing 4/15 of the powers, which is slower than *sieved*, but only requires logarithmic extra space. It is also faster for large *len*, since most power series multiplications are traded for additions. A slightly bigger gain for larger n could be achieved by using more small prime factors, at the expense of space.

9.22.3 Riemann zeta function

void **acb_dirichlet_zeta**(*acb_t* res, const *acb_t* s, *slong* prec)

Computes $\zeta(s)$ using an automatic choice of algorithm.

void **acb_dirichlet_zeta_jet**(*acb_t* res, const *acb_t* s, int deflate, *slong* len, *slong* prec)

Computes the first *len* terms of the Taylor series of the Riemann zeta function at s . If *deflate* is nonzero, computes the deflated function $\zeta(s) - 1/(s-1)$ instead.

void **acb_dirichlet_zeta_bound**(*mag_t* res, const *acb_t* s)

Computes an upper bound for $|\zeta(s)|$ quickly. On the critical strip (and slightly outside of it), formula (43.3) in [Rad1973] is used. To the right, evaluating at the real part of s gives a trivial bound. To the left, the functional equation is used.

void **acb_dirichlet_zeta_deriv_bound**(*mag_t* der1, *mag_t* der2, const *acb_t* s)

Sets *der1* to a bound for $|\zeta'(s)|$ and *der2* to a bound for $|\zeta''(s)|$. These bounds are mainly intended for use in the critical strip and will not be tight.

void **acb_dirichlet_eta**(*acb_t* res, const *acb_t* s, *slong* prec)

Sets *res* to the Dirichlet eta function $\eta(s) = \sum_{k=1}^{\infty} (-1)^{k+1}/k^s = (1-2^{1-s})\zeta(s)$, also known as the alternating zeta function. Note that the alternating character $\{1, -1\}$ is not itself a Dirichlet character.

void **acb_dirichlet_xi**(*acb_t* res, const *acb_t* s, *slong* prec)

Sets *res* to the Riemann xi function $\xi(s) = \frac{1}{2}s(s-1)\pi^{-s/2}\Gamma(\frac{1}{2}s)\zeta(s)$. The functional equation for xi is $\xi(1-s) = \xi(s)$.

9.22.4 Riemann-Siegel formula

The Riemann-Siegel (RS) formula is implemented closely following J. Arias de Reyna [Ari2011]. For $s = \sigma + it$ with $t > 0$, the expansion takes the form

$$\zeta(s) = \mathcal{R}(s) + X(s)\overline{\mathcal{R}}(1-s), \quad X(s) = \pi^{s-1/2} \frac{\Gamma((1-s)/2)}{\Gamma(s/2)}$$

where

$$\mathcal{R}(s) = \sum_{k=1}^N \frac{1}{k^s} + (-1)^{N-1} U a^{-\sigma} \left[\sum_{k=0}^K \frac{C_k(p)}{a^k} + RS_K \right]$$

$$U = \exp \left(-i \left[\frac{t}{2} \log \left(\frac{t}{2\pi} \right) - \frac{t}{2} - \frac{\pi}{8} \right] \right), \quad a = \sqrt{\frac{t}{2\pi}}, \quad N = \lfloor a \rfloor, \quad p = 1 - 2(a - N).$$

The coefficients $C_k(p)$ in the asymptotic part of the expansion are expressed in terms of certain auxiliary coefficients $d_j^{(k)}$ and $F^{(j)}(p)$. Because of artificial discontinuities, s should be exact inside the evaluation.

void **acb_dirichlet_zeta_rs_f_coeffs**(*acb_ptr* f, const *arb_t* p, *slong* n, *slong* prec)

Computes the coefficients $F^{(j)}(p)$ for $0 \leq j < n$. Uses power series division. This method breaks down when $p = \pm 1/2$ (which is not problem if s is an exact floating-point number).

void **acb_dirichlet_zeta_rs_d_coeffs**(*arb_ptr* d, const *arb_t* sigma, *slong* k, *slong* prec)

Computes the coefficients $d_j^{(k)}$ for $0 \leq j \leq \lfloor 3k/2 \rfloor + 1$. On input, the array d must contain the coefficients for $d_j^{(k-1)}$ unless $k = 0$, and these coefficients will be updated in-place.

void **acb_dirichlet_zeta_rs_bound**(*mag_t* err, const *acb_t* s, *slong* K)

Bounds the error term RS_K following Theorem 4.2 in Arias de Reyna.

void **acb_dirichlet_zeta_rs_r**(*acb_t* res, const *acb_t* s, *slong* K, *slong* prec)

Computes $\mathcal{R}(s)$ in the upper half plane. Uses precisely K asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on s and the precision.

void **acb_dirichlet_zeta_rs**(*acb_t* res, const *acb_t* s, *slong* K, *slong* prec)

Computes $\zeta(s)$ using the Riemann-Siegel formula. Uses precisely K asymptotic terms in the RS formula if this input parameter is positive; otherwise chooses the number of terms automatically based on s and the precision.

void **acb_dirichlet_zeta_jet_rs**(*acb_ptr* res, const *acb_t* s, *slong* len, *slong* prec)

Computes the first len terms of the Taylor series of the Riemann zeta function at s using the Riemann Siegel formula. This function currently only supports $len = 1$ or $len = 2$. A finite difference is used to compute the first derivative.

9.22.5 Hurwitz zeta function

void **acb_dirichlet_hurwitz**(*acb_t* res, const *acb_t* s, const *acb_t* a, *slong* prec)

Computes the Hurwitz zeta function $\zeta(s, a)$. This function automatically delegates to the code for the Riemann zeta function when $a = 1$. Some other special cases may also be handled by direct formulas. In general, Euler-Maclaurin summation is used.

9.22.6 Hurwitz zeta function precomputation

type **acb_dirichlet_hurwitz_precomp_struct**

type **acb_dirichlet_hurwitz_precomp_t**

void **acb_dirichlet_hurwitz_precomp_init**(*acb_dirichlet_hurwitz_precomp_t* pre, const *acb_t* s, int deflate, *slong* A, *slong* K, *slong* N, *slong* prec)

Precomputes a grid of Taylor polynomials for fast evaluation of $\zeta(s, a)$ on $a \in (0, 1]$ with fixed s . A is the initial shift to apply to a , K is the number of Taylor terms, N is the number of grid points. The precomputation requires NK evaluations of the Hurwitz zeta function, and each subsequent evaluation requires $2K$ simple arithmetic operations (polynomial evaluation) plus A powers. As K grows, the error is at most $O(1/(2AN)^K)$.

This function can be called with A set to zero, in which case no Taylor series precomputation is performed. This means that evaluation will be identical to calling **acb_dirichlet_hurwitz()** directly.

Otherwise, we require that A , K and N are all positive. For a finite error bound, we require $K + \text{re}(s) > 1$. To avoid an initial “bump” that steals precision and slows convergence, AN should be at least roughly as large as $|s|$, e.g. it is a good idea to have at least $AN > 0.5|s|$.

If *deflate* is set, the deflated Hurwitz zeta function is used, removing the pole at $s = 1$.

void **acb_dirichlet_hurwitz_precomp_init_num**(*acb_dirichlet_hurwitz_precomp_t* pre, const *acb_t* s, int deflate, double num_eval, *slong* prec)

Initializes *pre*, choosing the parameters A , K , and N automatically to minimize the cost of *num_eval* evaluations of the Hurwitz zeta function at argument s to precision *prec*.

void **acb_dirichlet_hurwitz_precomp_clear**(*acb_dirichlet_hurwitz_precomp_t* pre)

Clears the precomputed data.

void **acb_dirichlet_hurwitz_precomp_choose_param**(*ulong* *A, *ulong* *K, *ulong* *N, const *acb_t* s, double num_eval, *slong* prec)

Chooses precomputation parameters A , K and N to minimize the cost of num_eval evaluations of the Hurwitz zeta function at argument s to precision prec . If it is estimated that evaluating each Hurwitz zeta function from scratch would be better than performing a precomputation, A , K and N are all set to 0.

void **acb_dirichlet_hurwitz_precomp_bound**(*mag_t* res, const *acb_t* s, *slong* A, *slong* K, *slong* N)

Computes an upper bound for the truncation error (not accounting for roundoff error) when evaluating $\zeta(s, a)$ with precomputation parameters A , K , N , assuming that $0 < a \leq 1$. For details, see *Algorithms for the Hurwitz zeta function*.

void **acb_dirichlet_hurwitz_precomp_eval**(*acb_t* res, const *acb_dirichlet_hurwitz_precomp_t* pre, *ulong* p, *ulong* q, *slong* prec)

Evaluates $\zeta(s, p/q)$ using precomputed data, assuming that $0 < p/q \leq 1$.

9.22.7 Lerch transcendent

void **acb_dirichlet_lerch_phi_integral**(*acb_t* res, const *acb_t* z, const *acb_t* s, const *acb_t* a, *slong* prec)

void **acb_dirichlet_lerch_phi_direct**(*acb_t* res, const *acb_t* z, const *acb_t* s, const *acb_t* a, *slong* prec)

void **acb_dirichlet_lerch_phi**(*acb_t* res, const *acb_t* z, const *acb_t* s, const *acb_t* a, *slong* prec)

Computes the Lerch transcendent

$$\Phi(z, s, a) = \sum_{k=0}^{\infty} \frac{z^k}{(k+a)^s}$$

which is analytically continued for $|z| \geq 1$.

The *direct* version evaluates a truncation of the defining series. The *integral* version uses the Hankel contour integral

$$\Phi(z, s, a) = -\frac{\Gamma(1-s)}{2\pi i} \int_C \frac{(-t)^{s-1} e^{-at}}{1 - ze^{-t}} dt$$

where the path is deformed as needed to avoid poles and branch cuts of the integrand. The default method chooses an algorithm automatically and also checks for some special cases where the function can be expressed in terms of simpler functions (Hurwitz zeta, polylogarithms).

9.22.8 Stieltjes constants

void **acb_dirichlet_stieltjes**(*acb_t* res, const *fmpz_t* n, const *acb_t* a, *slong* prec)

Given a nonnegative integer n , sets *res* to the generalized Stieltjes constant $\gamma_n(a)$ which is the coefficient in the Laurent series of the Hurwitz zeta function at the pole

$$\zeta(s, a) = \frac{1}{s-1} + \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n(a) (s-1)^n.$$

With $a = 1$, this gives the ordinary Stieltjes constants for the Riemann zeta function.

This function uses an integral representation to permit fast computation for extremely large n [JB2018]. If n is moderate and the precision is high enough, it falls back to evaluating the Hurwitz zeta function of a power series and reading off the last coefficient.

Note that for computing a range of values $\gamma_0(a), \dots, \gamma_n(a)$, it is generally more efficient to evaluate the Hurwitz zeta function series expansion once at $s = 1$ than to call this function repeatedly, unless n is extremely large (at least several hundred).

9.22.9 Dirichlet character evaluation

```
void acb_dirichlet_chi(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi, ulong n,
                      slong prec)
```

Sets *res* to $\chi(n)$, the value of the Dirichlet character *chi* at the integer *n*.

```
void acb_dirichlet_chi_vec(acb_ptr v, const dirichlet_group_t G, const dirichlet_char_t chi, slong
                          nv, slong prec)
```

Compute the *nv* first Dirichlet values.

```
void acb_dirichlet_pairing(acb_t res, const dirichlet_group_t G, ulong m, ulong n, slong prec)
```

```
void acb_dirichlet_pairing_char(acb_t res, const dirichlet_group_t G, const dirichlet_char_t a,
                                const dirichlet_char_t b, slong prec)
```

Sets *res* to the value of the Dirichlet pairing $\chi(m, n)$ at numbers *m* and *n*. The second form takes two characters as input.

9.22.10 Dirichlet character Gauss, Jacobi and theta sums

```
void acb_dirichlet_gauss_sum_naive(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
                                   chi, slong prec)
```

```
void acb_dirichlet_gauss_sum_factor(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
                                    chi, slong prec)
```

```
void acb_dirichlet_gauss_sum_order2(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
                                     chi, slong prec)
```

```
void acb_dirichlet_gauss_sum_theta(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
                                   chi, slong prec)
```

```
void acb_dirichlet_gauss_sum(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi,
                             slong prec)
```

Sets *res* to the Gauss sum

$$G_q(a) = \sum_{x \bmod q} \chi_q(a, x) e^{\frac{2i\pi x}{q}}$$

- the *naive* version computes the sum as defined.
- the *factor* version writes it as a product of local Gauss sums by chinese remainder theorem.
- the *order2* version assumes *chi* is real and primitive and returns $i^p \sqrt{q}$ where *p* is the parity of χ .
- the *theta* version assumes that *chi* is primitive to obtain the Gauss sum by functional equation of the theta series at $t = 1$. An abort will be raised if the theta series vanishes at $t = 1$. Only 4 exceptional characters of conductor 300 and 600 are known to have this particularity, and none with primepower modulus.
- the default version automatically combines the above methods.
- the *ui* version only takes the Conrey number *a* as parameter.

```
void acb_dirichlet_jacobi_sum_naive(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
                                    chi1, const dirichlet_char_t chi2, slong prec)
```

```

void acb_dirichlet_jacobi_sum_factor(acb_t res, const dirichlet_group_t G, const
    dirichlet_char_t chi1, const dirichlet_char_t chi2, slong
    prec)

void acb_dirichlet_jacobi_sum_gauss(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
    chi1, const dirichlet_char_t chi2, slong prec)

void acb_dirichlet_jacobi_sum(acb_t res, const dirichlet_group_t G, const dirichlet_char_t chi1,
    const dirichlet_char_t chi2, slong prec)

void acb_dirichlet_jacobi_sum_ui(acb_t res, const dirichlet_group_t G, ulong a, ulong b, slong
    prec)
    
```

Computes the Jacobi sum

$$J_q(a, b) = \sum_{x \bmod q} \chi_q(a, x) \chi_q(b, 1 - x)$$

- the *naive* version computes the sum as defined.
- the *factor* version writes it as a product of local Jacobi sums
- the *gauss* version assumes ab is primitive and uses the formula $J_q(a, b)G_q(ab) = G_q(a)G_q(b)$
- the default version automatically combines the above methods.
- the *ui* version only takes the Conrey numbers a and b as parameters.

```

void acb_dirichlet_chi_theta_arb(acb_t res, const dirichlet_group_t G, const dirichlet_char_t
    chi, const arb_t t, slong prec)

void acb_dirichlet_ui_theta_arb(acb_t res, const dirichlet_group_t G, ulong a, const arb_t t,
    slong prec)
    
```

Compute the theta series $\Theta_q(a, t)$ for real argument $t > 0$. Beware that if $t < 1$ the functional equation

$$t\theta(a, t) = \epsilon(\chi)\theta\left(\frac{1}{a}, \frac{1}{t}\right)$$

should be used, which is not done automatically (to avoid recomputing the Gauss sum).

We call *theta series* of a Dirichlet character the quadratic series

$$\Theta_q(a) = \sum_{n \geq 0} \chi_q(a, n) n^p x^{n^2}$$

where p is the parity of the character $\chi_q(a, \cdot)$.

For $\Re(t) > 0$ we write $x(t) = \exp(-\frac{\pi}{N}t^2)$ and define

$$\Theta_q(a, t) = \sum_{n \geq 0} \chi_q(a, n) x(t)^{n^2}.$$

```

ulong acb_dirichlet_theta_length(ulong q, const arb_t t, slong prec)
    
```

Compute the number of terms to be summed in the theta series of argument t so that the tail is less than $2^{-\text{prec}}$.

```

void acb_dirichlet_qseries_arb_powers_naive(acb_t res, const arb_t x, int p, const ulong *a,
    const acb_dirichlet_roots_t z, slong len, slong
    prec)
    
```

```
void acb_dirichlet_qseries_arb_powers_smallorder(acb_t res, const arb_t x, int p, const ulong
                                                *a, const acb_dirichlet_roots_t z, slong len,
                                                slong prec)
```

Compute the series $\sum n^p z^{a_n} x^{n^2}$ for exponent list a , precomputed powers z and parity p (being 0 or 1).

The *naive* version sums the series as defined, while the *smallorder* variant evaluates the series on the quotient ring by a cyclotomic polynomial before evaluating at the root of unity, ignoring its argument z .

9.22.11 Discrete Fourier transforms

If f is a function $\mathbb{Z}/q\mathbb{Z} \rightarrow \mathbb{C}$, its discrete Fourier transform is the function defined on Dirichlet characters mod q by

$$\hat{f}(\chi) = \sum_{x \bmod q} \overline{\chi(x)} f(x)$$

See the *acb_dft.h* – *Discrete Fourier transform* module.

Here we take advantage of the Conrey isomorphism $G \rightarrow \hat{G}$ to consider the Fourier transform on Conrey labels as

$$g(a) = \sum_{b \bmod q} \overline{\chi_q(a, b)} f(b)$$

```
void acb_dirichlet_dft_conrey(acb_ptr w, acb_srcptr v, const dirichlet_group_t G, slong prec)
```

Compute the DFT of v using Conrey indices. This function assumes v and w are vectors of size $G \rightarrow \phi_q$, whose values correspond to a lexicographic ordering of Conrey logs (as obtained using *dirichlet_char_next()* or by *dirichlet_char_index()*).

For example, if $q = 15$, the Conrey elements are stored in following order

index	log = [e,f]	number = $7^e 11^f$
0	[0, 0]	1
1	[0, 1]	7
2	[0, 2]	4
3	[0, 3]	13
4	[0, 4]	1
5	[1, 0]	11
6	[1, 1]	2
7	[1, 2]	14
8	[1, 3]	8
9	[1, 4]	11

```
void acb_dirichlet_dft(acb_ptr w, acb_srcptr v, const dirichlet_group_t G, slong prec)
```

Compute the DFT of v using Conrey numbers. This function assumes v and w are vectors of size $G \rightarrow q$. All values at index not coprime to $G \rightarrow q$ are ignored.

9.22.12 Dirichlet L-functions

void `acb_dirichlet_root_number_theta`(*acb_t* res, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *slong* prec)

void `acb_dirichlet_root_number`(*acb_t* res, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *slong* prec)

Sets *res* to the root number $\epsilon(\chi)$ for a primitive character *chi*, which appears in the functional equation (where *p* is the parity of χ):

$$\left(\frac{q}{\pi}\right)^{\frac{s+p}{2}} \Gamma\left(\frac{s+p}{2}\right) L(s, \chi) = \epsilon(\chi) \left(\frac{q}{\pi}\right)^{\frac{1-s+p}{2}} \Gamma\left(\frac{1-s+p}{2}\right) L(1-s, \bar{\chi})$$

- The *theta* variant uses the evaluation at $t = 1$ of the Theta series.
- The default version computes it via the gauss sum.

void `acb_dirichlet_l_hurwitz`(*acb_t* res, const *acb_t* s, const *acb_dirichlet_hurwitz_precomp_t* precomp, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *slong* prec)

Computes $L(s, \chi)$ using decomposition in terms of the Hurwitz zeta function

$$L(s, \chi) = q^{-s} \sum_{k=1}^q \chi(k) \zeta\left(s, \frac{k}{q}\right).$$

If $s = 1$ and χ is non-principal, the deflated Hurwitz zeta function is used to avoid poles.

If *precomp* is *NULL*, each Hurwitz zeta function value is computed directly. If a pre-initialized *precomp* object is provided, this will be used instead to evaluate the Hurwitz zeta function.

void `acb_dirichlet_l_euler_product`(*acb_t* res, const *acb_t* s, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *slong* prec)

void `_acb_dirichlet_euler_product_real_ui`(*arb_t* res, *ulong* s, const signed char *chi, int mod, int reciprocal, *slong* prec)

Computes $L(s, \chi)$ directly using the Euler product. This is efficient if s has large positive real part. As implemented, this function only gives a finite result if $\operatorname{re}(s) \geq 2$.

An error bound is computed via `mag_hurwitz_zeta_uiui()`. If s is complex, replace it with its real part. Since

$$\frac{1}{L(s, \chi)} = \prod_p \left(1 - \frac{\chi(p)}{p^s}\right) = \sum_{k=1}^{\infty} \frac{\mu(k) \chi(k)}{k^s}$$

and the truncated product gives all smooth-index terms in the series, we have

$$\left| \prod_{p < N} \left(1 - \frac{\chi(p)}{p^s}\right) - \frac{1}{L(s, \chi)} \right| \leq \sum_{k=N}^{\infty} \frac{1}{k^s} = \zeta(s, N).$$

The underscore version specialized for integer s assumes that χ is a real Dirichlet character given by the explicit list *chi* of character values at $0, 1, \dots, \operatorname{mod} - 1$. If *reciprocal* is set, it computes $1/L(s, \chi)$ (this is faster if the reciprocal can be used directly).

void `acb_dirichlet_l`(*acb_t* res, const *acb_t* s, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *slong* prec)

Computes $L(s, \chi)$ using a default choice of algorithm.

void `acb_dirichlet_l_fmpq`(*acb_t* res, const *fmpq_t* s, const *dirichlet_group_t* G, const *dirichlet_char_t* chi, *slong* prec)

```
void acb_dirichlet_l_fmpq_afe(acb_t res, const fmpq_t s, const dirichlet_group_t G, const
                             dirichlet_char_t chi, slong prec)
```

Computes $L(s, \chi)$ where s is a rational number. The *afe* version uses the approximate functional equation; the default version chooses an algorithm automatically.

```
void acb_dirichlet_l_vec_hurwitz(acb_ptr res, const acb_t s, const
                                acb_dirichlet_hurwitz_precomp_t precomp, const
                                dirichlet_group_t G, slong prec)
```

Compute all values $L(s, \chi)$ for $\chi \bmod q$, using the Hurwitz zeta function and a discrete Fourier transform. The output *res* is assumed to have length $G \rightarrow \phi_q$ and values are stored by lexicographically ordered Conrey logs. See `acb_dirichlet_dft_conrey()`.

If *precomp* is *NULL*, each Hurwitz zeta function value is computed directly. If a pre-initialized *precomp* object is provided, this will be used instead to evaluate the Hurwitz zeta function.

```
void acb_dirichlet_l_jet(acb_ptr res, const acb_t s, const dirichlet_group_t G, const
                        dirichlet_char_t chi, int deflate, slong len, slong prec)
```

Computes the Taylor expansion of $L(s, \chi)$ to length *len*, i.e. $L(s), L'(s), \dots, L^{(len-1)}(s)/(len-1)!$. If *deflate* is set, computes the expansion of

$$L(s, \chi) - \frac{\sum_{k=1}^q \chi(k)}{(s-1)q}$$

instead. If *chi* is a principal character, then this has the effect of subtracting the pole with residue $\sum_{k=1}^q \chi(k) = \phi(q)/q$ that is located at $s = 1$. In particular, when evaluated at $s = 1$, this gives the regular part of the Laurent expansion. When *chi* is non-principal, *deflate* has no effect.

```
void _acb_dirichlet_l_series(acb_ptr res, acb_srcptr s, slong slen, const dirichlet_group_t G,
                            const dirichlet_char_t chi, int deflate, slong len, slong prec)
```

```
void acb_dirichlet_l_series(acb_poly_t res, const acb_poly_t s, const dirichlet_group_t G, const
                            dirichlet_char_t chi, int deflate, slong len, slong prec)
```

Sets *res* to the power series $L(s, \chi)$ where s is a given power series, truncating the result to length *len*. See `acb_dirichlet_l_jet()` for the meaning of the *deflate* flag.

9.22.13 Hardy Z-functions

For convenience, setting both *G* and *chi* to *NULL* in the following methods selects the Riemann zeta function.

Currently, these methods require *chi* to be a primitive character.

```
void acb_dirichlet_hardy_theta(acb_ptr res, const acb_t t, const dirichlet_group_t G, const
                              dirichlet_char_t chi, slong len, slong prec)
```

Computes the phase function used to construct the Z-function. We have

$$\theta(t) = -\frac{t}{2} \log(\pi/q) - \frac{i \log(\epsilon)}{2} + \frac{\log \Gamma((s+\delta)/2) - \log \Gamma((1-s+\delta)/2)}{2i}$$

where $s = 1/2 + it$, δ is the parity of *chi*, and ϵ is the root number as computed by `acb_dirichlet_root_number()`. The first *len* terms in the Taylor expansion are written to the output.

```
void acb_dirichlet_hardy_z(acb_ptr res, const acb_t t, const dirichlet_group_t G, const
                           dirichlet_char_t chi, slong len, slong prec)
```

Computes the Hardy Z-function, also known as the Riemann-Siegel Z-function $Z(t) = e^{i\theta(t)} L(1/2 + it)$, which is real-valued for real t . The first *len* terms in the Taylor expansion are written to the output.

```
void _acb_dirichlet_hardy_theta_series(acb_ptr res, acb_srcptr t, slong tlen, const
                                     dirichlet_group_t G, const dirichlet_char_t chi, slong
                                     len, slong prec)
```

```
void acb_dirichlet_hardy_theta_series(acb_poly_t res, const acb_poly_t t, const
                                     dirichlet_group_t G, const dirichlet_char_t chi, slong len,
                                     slong prec)
```

Sets *res* to the power series $\theta(t)$ where *t* is a given power series, truncating the result to length *len*.

```
void _acb_dirichlet_hardy_z_series(acb_ptr res, acb_srcptr t, slong tlen, const dirichlet_group_t
                                   G, const dirichlet_char_t chi, slong len, slong prec)
```

```
void acb_dirichlet_hardy_z_series(acb_poly_t res, const acb_poly_t t, const dirichlet_group_t
                                   G, const dirichlet_char_t chi, slong len, slong prec)
```

Sets *res* to the power series $Z(t)$ where *t* is a given power series, truncating the result to length *len*.

9.22.14 Gram points

```
void acb_dirichlet_gram_point(arb_t res, const fmpz_t n, const dirichlet_group_t G, const
                              dirichlet_char_t chi, slong prec)
```

Sets *res* to the *n*-th Gram point g_n , defined as the unique solution in $[7, \infty)$ of $\theta(g_n) = \pi n$. Currently only the Gram points corresponding to the Riemann zeta function are supported and *G* and *chi* must both be set to *NULL*. Requires $n \geq -1$.

9.22.15 Riemann zeta function zeros

The following functions for counting and isolating zeros of the Riemann zeta function use the ideas from the implementation of Turing's method in mpmath [Joh2018b] by Juan Arias de Reyna, described in [Ari2012].

```
ulong acb_dirichlet_turing_method_bound(const fmpz_t p)
```

Computes an upper bound *B* for the minimum number of consecutive good Gram blocks sufficient to count nontrivial zeros of the Riemann zeta function using Turing's method [Tur1953] as updated by [Leh1970], [Bre1979], and [Tru2011].

Let $N(T)$ denote the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \text{Im}(s) \leq T$. If at least *B* consecutive Gram blocks with union $[g_n, g_p)$ satisfy Rosser's rule, then $N(g_n) \leq n + 1$ and $N(g_p) \geq p + 1$.

```
int _acb_dirichlet_definite_hardy_z(arb_t res, const arf_t t, slong *pprec)
```

Sets *res* to the Hardy Z-function $Z(t)$. The initial precision (* *pprec*) is increased as necessary to determine the sign of $Z(t)$. The sign is returned.

```
void _acb_dirichlet_isolate_gram_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)
```

Uses Gram's law to compute an interval (a, b) that contains the *n*-th zero of the Hardy Z-function and no other zero. Requires $1 \leq n \leq 126$.

```
void _acb_dirichlet_isolate_rosser_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)
```

Uses Rosser's rule to compute an interval (a, b) that contains the *n*-th zero of the Hardy Z-function and no other zero. Requires $1 \leq n \leq 13999526$.

```
void _acb_dirichlet_isolate_turing_hardy_z_zero(arf_t a, arf_t b, const fmpz_t n)
```

Computes an interval (a, b) that contains the *n*-th zero of the Hardy Z-function and no other zero, following Turing's method. Requires $n \geq 2$.

void **acb_dirichlet_isolate_hardy_z_zero**(*arf_t* a, *arf_t* b, const *fmpz_t* n)

Computes an interval (a, b) that contains the n -th zero of the Hardy Z-function and contains no other zero, using the most appropriate underscore version of this function. Requires $n \geq 1$.

void **_acb_dirichlet_refine_hardy_z_zero**(*arb_t* res, const *arf_t* a, const *arf_t* b, *slong* prec)

Sets *res* to the unique zero of the Hardy Z-function in the interval (a, b) .

void **acb_dirichlet_hardy_z_zero**(*arb_t* res, const *fmpz_t* n, *slong* prec)

Sets *res* to the n -th zero of the Hardy Z-function, requiring $n \geq 1$.

void **acb_dirichlet_hardy_z_zeros**(*arb_ptr* res, const *fmpz_t* n, *slong* len, *slong* prec)

Sets the entries of *res* to *len* consecutive zeros of the Hardy Z-function, beginning with the n -th zero. Requires positive n .

void **acb_dirichlet_zeta_zero**(*acb_t* res, const *fmpz_t* n, *slong* prec)

Sets *res* to the n -th nontrivial zero of $\zeta(s)$, requiring $n \geq 1$.

void **acb_dirichlet_zeta_zeros**(*acb_ptr* res, const *fmpz_t* n, *slong* len, *slong* prec)

Sets the entries of *res* to *len* consecutive nontrivial zeros of $\zeta(s)$ beginning with the n -th zero. Requires positive n .

void **_acb_dirichlet_exact_zeta_nzeros**(*fmpz_t* res, const *arf_t* t)

void **acb_dirichlet_zeta_nzeros**(*arb_t* res, const *arb_t* t, *slong* prec)

Compute the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \text{Im}(s) \leq t$.

void **acb_dirichlet_backlund_s**(*arb_t* res, const *arb_t* t, *slong* prec)

Compute $S(t) = \frac{1}{\pi} \arg \zeta(\frac{1}{2} + it)$ where the argument is defined by continuous variation of s in $\zeta(s)$ starting at $s = 2$, then vertically to $s = 2 + it$, then horizontally to $s = \frac{1}{2} + it$. In particular $\arg$ in this context is not the principal value of the argument, and it cannot be computed directly by *acb_arg()*. In practice $S(t)$ is computed as $S(t) = N(t) - \frac{1}{\pi}\theta(t) - 1$ where $N(t)$ is *acb_dirichlet_zeta_nzeros()* and $\theta(t)$ is *acb_dirichlet_hardy_theta()*.

void **acb_dirichlet_backlund_s_bound**(*mag_t* res, const *arb_t* t)

Compute an upper bound for $|S(t)|$ quickly. Theorem 1 and the bounds in (1.2) in [Tru2014] are used.

void **acb_dirichlet_zeta_nzeros_gram**(*fmpz_t* res, const *fmpz_t* n)

Compute $N(g_n)$. That is, compute the number of zeros (counted according to their multiplicities) of $\zeta(s)$ in the region $0 < \text{Im}(s) \leq g_n$ where g_n is the n -th Gram point. Requires $n \geq -1$.

slong **acb_dirichlet_backlund_s_gram**(const *fmpz_t* n)

Compute $S(g_n)$ where g_n is the n -th Gram point. Requires $n \geq -1$.

9.22.16 Riemann zeta function zeros (Platt's method)

The following functions related to the Riemann zeta function use the ideas and formulas described by David J. Platt in [Pla2017].

void **acb_dirichlet_platt_scaled_lambda**(*arb_t* res, const *arb_t* t, *slong* prec)

Compute $\Lambda(t)e^{\pi t/4}$ where

$$\Lambda(t) = \pi^{-\frac{it}{2}} \Gamma\left(\frac{\frac{1}{2} + it}{2}\right) \zeta\left(\frac{1}{2} + it\right)$$

is defined in the beginning of section 3 of [Pla2017]. As explained in [Pla2011] this function has the same zeros as $\zeta(1/2 + it)$ and is real-valued by the functional equation, and the exponential factor is designed to counteract the decay of the gamma factor as t increases.

```
void acb_dirichlet_platt_scaled_lambda_vec(arb_ptr res, const fmpz_t T, slong A, slong B, slong
prec)
```

```
void acb_dirichlet_platt_multieval(arb_ptr res, const fmpz_t T, slong A, slong B, const arb_t h,
const fmpz_t J, slong K, slong sigma, slong prec)
```

```
void acb_dirichlet_platt_multieval_threaded(arb_ptr res, const fmpz_t T, slong A, slong B,
const arb_t h, const fmpz_t J, slong K, slong
sigma, slong prec)
```

Compute `acb_dirichlet_platt_scaled_lambda()` at $N = AB$ points on a grid, following the notation of [Pla2017]. The first point on the grid is $T - B/2$ and the distance between grid points is $1/A$. The product $N = AB$ must be an even integer. The multieval versions evaluate the function at all points on the grid simultaneously using discrete Fourier transforms, and they require the four additional tuning parameters h , J , K , and σ . The *threaded* multieval version splits the computation over the number of threads returned by `flint_get_num_threads()`, while the default multieval version chooses whether to use multithreading automatically.

```
void acb_dirichlet_platt_ws_interpolation(arb_t res, arf_t deriv, const arb_t t0, arb_srcptr p,
const fmpz_t T, slong A, slong B, slong Ns_max,
const arb_t H, slong sigma, slong prec)
```

Compute `acb_dirichlet_platt_scaled_lambda()` at t_0 by Gaussian-windowed Whittaker-Shannon interpolation of points evaluated by `acb_dirichlet_platt_scaled_lambda_vec()`. The derivative is also approximated if the output parameter *deriv* is not *NULL*. Ns_max defines the maximum number of supporting points to be used in the interpolation on either side of t_0 . H is the standard deviation of the Gaussian window centered on t_0 to be applied before the interpolation. σ is an odd positive integer tuning parameter $\sigma \in 2\mathbb{Z}_{>0} + 1$ used in computing error bounds.

```
slong _acb_dirichlet_platt_local_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, const
fmpz_t T, slong A, slong B, const arb_t h,
const fmpz_t J, slong K, slong sigma_grid,
slong Ns_max, const arb_t H, slong
sigma_interp, slong prec)
```

```
slong acb_dirichlet_platt_local_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, slong
prec)
```

```
slong acb_dirichlet_platt_hardy_z_zeros(arb_ptr res, const fmpz_t n, slong len, slong prec)
```

Sets at most the first len entries of *res* to consecutive zeros of the Hardy Z-function starting with the n -th zero. The number of obtained consecutive zeros is returned. The first two function variants each make a single call to Platt's grid evaluation of the scaled Lambda function, whereas the third variant performs as many evaluations as necessary to obtain len consecutive zeros. The final several parameters of the underscored local variant have the same meanings as in the functions `acb_dirichlet_platt_multieval()` and `acb_dirichlet_platt_ws_interpolation()`. The non-underscored variants currently expect $10^4 \leq n \leq 10^{23}$. The user has the option of multi-threading through `flint_set_num_threads(numthreads)`.

```
slong acb_dirichlet_platt_zeta_zeros(acb_ptr res, const fmpz_t n, slong len, slong prec)
```

Sets at most the first len entries of *res* to consecutive zeros of the Riemann zeta function starting with the n -th zero. The number of obtained consecutive zeros is returned. It currently expects $10^4 \leq n \leq 10^{23}$. The user has the option of multi-threading through `flint_set_num_threads(numthreads)`.

9.23 bernoulli.h – support for Bernoulli numbers

This module provides helper functions for exact or approximate calculation of the Bernoulli numbers, which are defined by the exponential generating function

$$\frac{x}{e^x - 1} = \sum_{n=0}^{\infty} B_n \frac{x^n}{n!}.$$

Efficient algorithms are implemented for both multi-evaluation and calculation of isolated Bernoulli numbers. A global (or thread-local) cache is also provided, to support fast repeated evaluation of various special functions that depend on the Bernoulli numbers (including the gamma function and the Riemann zeta function).

9.23.1 Generation of Bernoulli numbers

type **bernoulli_rev_t**

An iterator object for generating a range of even-indexed Bernoulli numbers exactly in reverse order, i.e. computing the exact fractions $B_n, B_{n-2}, B_{n-4}, \dots, B_0$. The Bernoulli numbers are generated from scratch, i.e. no caching is performed.

The Bernoulli numbers are computed by direct summation of the zeta series. This is made fast by storing a table of powers (as done by [Blo2009]). As an optimization, we only include the odd powers, and use fixed-point arithmetic.

The reverse iteration order is preferred for performance reasons, as the powers can be updated using multiplications instead of divisions, and we avoid having to periodically recompute terms to higher precision. To generate Bernoulli numbers in the forward direction without having to store all of them, one can split the desired range into smaller blocks and compute each block with a single reverse pass.

void **bernoulli_rev_init**(*bernoulli_rev_t* iter, *ulong* n)

Initializes the iterator *iter*. The first Bernoulli number to be generated by calling *bernoulli_rev_next()* is B_n . It is assumed that n is even.

void **bernoulli_rev_next**(*fmpz_t* numer, *fmpz_t* denom, *bernoulli_rev_t* iter)

Sets *numer* and *denom* to the exact, reduced numerator and denominator of the Bernoulli number B_k and advances the state of *iter* so that the next invocation generates B_{k-2} .

void **bernoulli_rev_clear**(*bernoulli_rev_t* iter)

Frees all memory allocated internally by *iter*.

void **bernoulli_fmpz_vec_no_cache**(*fmpz_t* *res, *ulong* a, *slong* num)

Writes *num* consecutive Bernoulli numbers to *res* starting with B_a . This function is not currently optimized for a small count *num*. The entries are not read from or written to the Bernoulli number cache; if retrieving a vector of Bernoulli numbers is needed more than once, use *bernoulli_cache_compute()* followed by *bernoulli_fmpz_ui()* instead.

This function is a wrapper for the *rev* iterators. It can use multiple threads internally.

9.23.2 Caching

slong **bernoulli_cache_num**

fmpq ***bernoulli_cache**

Cache of Bernoulli numbers. Uses thread-local storage if enabled in FLINT.

void **bernoulli_cache_compute**(*slong* n)

Makes sure that the Bernoulli numbers up to at least B_{n-1} are cached. Calling `flint_cleanup()` frees the cache.

The cache is extended by calling `bernoulli_fmpq_vec_no_cache()` internally.

9.23.3 Bounding

slong **bernoulli_bound_2exp_si**(*ulong* n)

Returns an integer b such that $|B_n| \leq 2^b$. Uses a lookup table for small n , and for larger n uses the inequality $|B_n| < 4n!/(2\pi)^n < 4(n+1)^{n+1}e^{-n}/(2\pi)^n$. Uses integer arithmetic throughout, with the bound for the logarithm being looked up from a table. If $|B_n| = 0$, returns `LONG_MIN`. Otherwise, the returned exponent b is never more than one percent larger than the true magnitude.

This function is intended for use when n small enough that one might comfortably compute B_n exactly. It aborts if n is so large that internal overflow occurs.

9.23.4 Isolated Bernoulli numbers

ulong **bernoulli_mod_p_harvey**(*ulong* n, *ulong* p)

Returns the B_n modulo the prime number p , computed using Harvey's algorithm [Har2010]. The running time is linear in p . If p divides the numerator of B_n , `UWORD_MAX` is returned as an error code.

void **_bernoulli_fmpq_ui_zeta**(*fmpz_t* num, *fmpz_t* den, *ulong* n)

void **_bernoulli_fmpq_ui_multi_mod**(*fmpz_t* num, *fmpz_t* den, *ulong* n, double alpha)

Sets *num* and *den* to the reduced numerator and denominator of the Bernoulli number B_n .

The *zeta* version computes the denominator d using the von Staudt-Clausen theorem, numerically approximates B_n using `arb_bernoulli_ui_zeta()`, and then rounds dB_n to the correct numerator.

The *multi_mod* version reconstructs B_n by computing the high bits via the Riemann zeta function and the low bits via Harvey's multimodular algorithm. The tuning parameter *alpha* should be a fraction between 0 and 1 controlling the number of bits to compute by the multimodular algorithm. If set to a negative number, a default value will be used.

void **_bernoulli_fmpq_ui**(*fmpz_t* num, *fmpz_t* den, *ulong* n)

void **bernoulli_fmpq_ui**(*fmpz_t* b, *ulong* n)

Computes the Bernoulli number B_n as an exact fraction, for an isolated integer n . This function reads B_n from the global cache if the number is already cached, but does not automatically extend the cache by itself.

9.24 hypgeom.h – support for hypergeometric series

This module provides functions for high-precision evaluation of series of the form

$$\sum_{k=0}^{n-1} \frac{A(k)}{B(k)} z^k \prod_{j=1}^k \frac{P(j)}{Q(j)}$$

where A, B, P, Q are polynomials. The present version only supports $A, B, P, Q \in \mathbb{Z}[k]$ (represented using the FLINT `mpz_poly_t` type). This module also provides functions for high-precision evaluation of infinite series ($n \rightarrow \infty$), with automatic, rigorous error bounding.

Note that we can standardize to $A = B = 1$ by setting $\tilde{P}(k) = P(k)A(k)B(k-1)$, $\tilde{Q}(k) = Q(k)A(k-1)B(k)$. However, separating out A and B is convenient and improves efficiency during evaluation.

9.24.1 Strategy for error bounding

We wish to evaluate $S(z) = \sum_{k=0}^{\infty} T(k)z^k$ where $T(k)$ satisfies $T(0) = 1$ and

$$T(k) = R(k)T(k-1) = \left(\frac{P(k)}{Q(k)} \right) T(k-1)$$

for given polynomials

$$\begin{aligned} P(k) &= a_p k^p + a_{p-1} k^{p-1} + \dots a_0 \\ Q(k) &= b_q k^q + b_{q-1} k^{q-1} + \dots b_0. \end{aligned}$$

For convergence, we require $p < q$, or $p = q$ with $|z||a_p| < |b_q|$. We also assume that $P(k)$ and $Q(k)$ have no roots among the positive integers (if there are positive integer roots, the sum is either finite or undefined). With these conditions satisfied, our goal is to find a parameter $n \geq 0$ such that

$$\left| \sum_{k=n}^{\infty} T(k)z^k \right| \leq 2^{-d}.$$

We can rewrite the hypergeometric term ratio as

$$zR(k) = z \frac{P(k)}{Q(k)} = z \left(\frac{a_p}{b_q} \right) \frac{1}{k^{q-p}} F(k)$$

where

$$F(k) = \frac{1 + \tilde{a}_1/k + \tilde{a}_2/k^2 + \dots + \tilde{a}_q/k^p}{1 + \tilde{b}_1/k + \tilde{b}_2/k^2 + \dots + \tilde{b}_q/k^q} = 1 + O(1/k)$$

and where $\tilde{a}_i = a_{p-i}/a_p$, $\tilde{b}_i = b_{q-i}/b_q$. Next, we define

$$C = \max_{1 \leq i \leq p} |\tilde{a}_i|^{(1/i)}, \quad D = \max_{1 \leq i \leq q} |\tilde{b}_i|^{(1/i)}.$$

Now, if $k > C$, the magnitude of the numerator of $F(k)$ is bounded from above by

$$1 + \sum_{i=1}^p \left(\frac{C}{k} \right)^i \leq 1 + \frac{C}{k-C}$$

and if $k > 2D$, the magnitude of the denominator of $F(k)$ is bounded from below by

$$1 - \sum_{i=1}^q \left(\frac{D}{k} \right)^i \geq 1 - \frac{D}{D-k}.$$

Putting the inequalities together gives the following bound, valid for $k > K = \max(C, 2D)$:

$$|F(k)| \leq \frac{k(k-D)}{(k-C)(k-2D)} = \left(1 + \frac{C}{k-C}\right) \left(1 + \frac{D}{k-2D}\right).$$

Let $r = q - p$ and $\tilde{z} = |za_p/b_q|$. Assuming $k > \max(C, 2D, \tilde{z}^{1/r})$, we have

$$|zR(k)| \leq G(k) = \frac{\tilde{z}F(k)}{k^r}$$

where $G(k)$ is monotonically decreasing. Now we just need to find an n such that $G(n) < 1$ and for which $|T(n)|/(1 - G(n)) \leq 2^{-d}$. This can be done by computing a floating-point guess for n then trying successively larger values.

This strategy leaves room for some improvement. For example, if $\tilde{b}_1$ is positive and large, the bound B becomes very pessimistic (a larger positive $\tilde{b}_1$ causes faster convergence, not slower convergence).

9.24.2 Types, macros and constants

type `hypgeom_struct`

type `hypgeom_t`

Stores polynomials A , B , P , Q and precomputed bounds, representing a fixed hypergeometric series.

9.24.3 Memory management

void `hypgeom_init(hypgeom_t hyp)`

void `hypgeom_clear(hypgeom_t hyp)`

9.24.4 Error bounding

slong `hypgeom_estimate_terms(const mag_t z, int r, slong d)`

Computes an approximation of the largest n such that $|z|^n/(n!)^r = 2^{-d}$, giving a first-order estimate of the number of terms needed to approximate the sum of a hypergeometric series of weight $r \geq 0$ and argument z to an absolute precision of $d \geq 0$ bits. If $r = 0$, the direct solution of the equation is given by $n = (\log(1 - z) - d \log 2)/\log z$. If $r > 0$, using $\log n! \approx n \log n - n$ gives an equation that can be solved in terms of the Lambert W -function as $n = (d \log 2)/(r W(t))$ where $t = (d \log 2)/(erz^{1/r})$.

The evaluation is done using double precision arithmetic. The function aborts if the computed value of n is greater than or equal to `LONG_MAX / 2`.

slong `hypgeom_bound(mag_t error, int r, slong C, slong D, slong K, const mag_t TK, const mag_t z, slong prec)`

Computes a truncation parameter sufficient to achieve *prec* bits of absolute accuracy, according to the strategy described above. The input consists of r , C , D , K , precomputed bound for $T(K)$, and $\tilde{z} = z(a_p/b_q)$, such that for $k > K$, the hypergeometric term ratio is bounded by

$$\frac{\tilde{z}}{k^r} \frac{k(k-D)}{(k-C)(k-2D)}.$$

Given this information, we compute a ε and an integer n such that $|\sum_{k=n}^{\infty} T(k)| \leq \varepsilon \leq 2^{-\text{prec}}$. The output variable *error* is set to the value of ε , and n is returned.

void `hypgeom_precompute(hypgeom_t hyp)`

Precomputes the bounds data C , D , K and an upper bound for $T(K)$.

9.24.5 Summation

void **arb_hypgeom_sum**(*arb_t* P, *arb_t* Q, const *hypgeom_t* hyp, *slong* n, *slong* prec)

Computes P, Q such that $P/Q = \sum_{k=0}^{n-1} T(k)$ where $T(k)$ is defined by *hyp*, using binary splitting and a working precision of *prec* bits.

void **arb_hypgeom_infsum**(*arb_t* P, *arb_t* Q, *hypgeom_t* hyp, *slong* tol, *slong* prec)

Computes P, Q such that $P/Q = \sum_{k=0}^{\infty} T(k)$ where $T(k)$ is defined by *hyp*, using binary splitting and working precision of *prec* bits. The number of terms is chosen automatically to bound the truncation error by at most $2^{-\text{tol}}$. The bound for the truncation error is included in the output as part of P .

9.25 partitions.h – computation of the partition function

This module implements the asymptotically fast algorithm for evaluating the integer partition function $p(n)$ described in [Joh2012]. The idea is to evaluate a truncation of the Hardy-Ramanujan-Rademacher series using tight precision estimates, and symbolically factoring the occurring exponential sums.

An implementation based on floating-point arithmetic can also be found in FLINT. That version relies on some numerical subroutines that have not been proved correct.

The implementation provided here uses ball arithmetic throughout to guarantee a correct error bound for the numerical approximation of $p(n)$. Optionally, hardware double arithmetic can be used for low-precision terms. This gives a significant speedup for small (e.g. $n < 10^6$).

void **partitions_rademacher_bound**(*arf_t* b, const *fmpz_t* n, *ulong* N)

Sets b to an upper bound for

$$M(n, N) = \frac{44\pi^2}{225\sqrt{3}} N^{-1/2} + \frac{\pi\sqrt{2}}{75} \left(\frac{N}{n-1}\right)^{1/2} \sinh\left(\frac{\pi}{N} \sqrt{\frac{2n}{3}}\right).$$

This formula gives an upper bound for the truncation error in the Hardy-Ramanujan-Rademacher formula when the series is taken up to the term $t(n, N)$ inclusive.

void **partitions_hrr_sum_arb**(*arb_t* x, const *fmpz_t* n, *slong* N0, *slong* N, int use_doubles)

Evaluates the partial sum $\sum_{k=N_0}^N t(n, k)$ of the Hardy-Ramanujan-Rademacher series.

If *use_doubles* is nonzero, doubles and the system's standard library math functions are used to evaluate the smallest terms. This significantly speeds up evaluation for small n (e.g. $n < 10^6$), and gives a small speed improvement for larger n , but the result is not guaranteed to be correct. In practice, the error is estimated very conservatively, and unless the system's standard library is broken, use of doubles can be considered safe. Setting *use_doubles* to zero gives a fully guaranteed bound.

void **partitions_fmpz_fmpz**(*fmpz_t* p, const *fmpz_t* n, int use_doubles)

Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

If n is sufficiently large and a number of threads greater than 1 has been selected with *flint_set_num_threads()*, the computation time will be reduced by using two threads.

See *partitions_hrr_sum_arb()* for an explanation of the *use_doubles* option.

void **partitions_fmpz_ui**(*fmpz_t* p, *ulong* n)

Computes the partition function $p(n)$ using the Hardy-Ramanujan-Rademacher formula. This function computes a numerical ball containing $p(n)$ and verifies that the ball contains a unique integer.

void **partitions_fmpz_ui_using_doubles**(*fmpz_t* p, *ulong* n)

Computes the partition function $p(n)$, enabling the use of doubles internally. This significantly speeds up evaluation for small n (e.g. $n < 10^6$), but the error bounds are not certified (see remarks for `partitions_hrr_sum_arb()`).

void **partitions_leading_fmpz**(*arb_t* res, const *fmpz_t* n, *slong* prec)

Sets *res* to the leading term in the Hardy-Ramanujan series for $p(n)$ (without Rademacher's correction of this term, which is vanishingly small when n is large), that is, $\sqrt{12}(1 - 1/t)e^t/(24n - 1)$ where $t = \pi\sqrt{24n - 1}/6$.

9.26 arb_calc.h – calculus with real-valued functions

This module provides functions for operations of calculus over the real numbers (intended to include root-finding, optimization, integration, and so on). It is planned that the module will include two types of algorithms:

- Interval algorithms that give provably correct results. An example would be numerical integration on an interval by dividing the interval into small balls and evaluating the function on each ball, giving rigorous upper and lower bounds.
- Conventional numerical algorithms that use heuristics to estimate the accuracy of a result, without guaranteeing that it is correct. An example would be numerical integration based on pointwise evaluation, where the error is estimated by comparing the results with two different sets of evaluation points. Ball arithmetic then still tracks the accuracy of the function evaluations.

Any algorithms of the second kind will be clearly marked as such.

9.26.1 Types, macros and constants

type **arb_calc_func_t**

Typedef for a pointer to a function with signature:

```
int func(arb_ptr out, const arb_t inp, void * param, slong order, slong prec)
```

implementing a univariate real function $f(x)$. When called, *func* should write to *out* the first *order* coefficients in the Taylor series expansion of $f(x)$ at the point *inp*, evaluated at a precision of *prec* bits. The *param* argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that *out* and *inp* are not aliased and that *order* is positive.

ARB_CALC_SUCCESS

Return value indicating that an operation is successful.

ARB_CALC_IMPRECISE_INPUT

Return value indicating that the input to a function probably needs to be computed more accurately.

ARB_CALC_NO_CONVERGENCE

Return value indicating that an algorithm has failed to convergence, possibly due to the problem not having a solution, the algorithm not being applicable, or the precision being insufficient

9.26.2 Debugging

int **arb_calc_verbose**

If set, enables printing information about the calculation to standard output.

9.26.3 Subdivision-based root finding

type **arf_interval_struct**

type **arf_interval_t**

An *arf_interval_struct* consists of a pair of *arf_struct*, representing an interval used for subdivision-based root-finding. An *arf_interval_t* is defined as an array of length one of type *arf_interval_struct*, permitting an *arf_interval_t* to be passed by reference.

type **arf_interval_ptr**

Alias for *arf_interval_struct **, used for vectors of intervals.

type **arf_interval_srcptr**

Alias for *const arf_interval_struct **, used for vectors of intervals.

void **arf_interval_init**(*arf_interval_t* v)

void **arf_interval_clear**(*arf_interval_t* v)

arf_interval_ptr **_arf_interval_vec_init**(*slong* n)

void **_arf_interval_vec_clear**(*arf_interval_ptr* v, *slong* n)

void **arf_interval_set**(*arf_interval_t* v, *const arf_interval_t* u)

void **arf_interval_swap**(*arf_interval_t* v, *arf_interval_t* u)

void **arf_interval_get_arb**(*arb_t* x, *const arf_interval_t* v, *slong* prec)

void **arf_interval_printd**(*const arf_interval_t* v, *slong* n)

Helper functions for endpoint-based intervals.

void **arf_interval_fprintfd**(FILE *file, *const arf_interval_t* v, *slong* n)

Helper functions for endpoint-based intervals.

slong **arb_calc_isolate_roots**(*arf_interval_ptr* *found, int **flags, *arb_calc_func_t* func, void *param, *const arf_interval_t* interval, *slong* maxdepth, *slong* maxeval, *slong* maxfound, *slong* prec)

Rigorously isolates single roots of a real analytic function on the interior of an interval.

This routine writes an array of *n* interesting subintervals of *interval* to *found* and corresponding flags to *flags*, returning the integer *n*. The output has the following properties:

- The function has no roots on *interval* outside of the output subintervals.
- Subintervals are sorted in increasing order (with no overlap except possibly starting and ending with the same point).
- Subintervals with a flag of 1 contain exactly one (single) root.
- Subintervals with any other flag may or may not contain roots.

If no flags other than 1 occur, all roots of the function on *interval* have been isolated. If there are output subintervals on which the existence or nonexistence of roots could not be determined, the user may attempt further searches on those subintervals (possibly with increased precision and/or increased bounds for the breaking criteria). Note that roots of multiplicity higher than one and roots located exactly at endpoints cannot be isolated by the algorithm.

The following breaking criteria are implemented:

- At most *maxdepth* recursive subdivisions are attempted. The smallest details that can be distinguished are therefore about $2^{-\text{maxdepth}}$ times the width of *interval*. A typical, reasonable value might be between 20 and 50.
- If the total number of tested subintervals exceeds *maxeval*, the algorithm is terminated and any untested subintervals are added to the output. The total number of calls to *func* is thereby restricted to a small multiple of *maxeval* (the actual count can be slightly higher depending on implementation details). A typical, reasonable value might be between 100 and 100000.
- The algorithm terminates if *maxfound* roots have been isolated. In particular, setting *maxfound* to 1 can be used to locate just one root of the function even if there are numerous roots. To try to find all roots, *LONG_MAX* may be passed.

The argument *prec* denotes the precision used to evaluate the function. It is possibly also used for some other arithmetic operations performed internally by the algorithm. Note that it probably does not make sense for *maxdepth* to exceed *prec*.

Warning: it is assumed that subdivision points of *interval* can be represented exactly as floating-point numbers in memory. Do not pass $1 \pm 2^{-10^{100}}$ as input.

```
int arb_calc_refine_root_bisect(arb_interval_t r, arb_calc_func_t func, void *param, const
                               arb_interval_t start, slong iter, slong prec)
```

Given an interval *start* known to contain a single root of *func*, refines it using *iter* bisection steps. The algorithm can return a failure code if the sign of the function at an evaluation point is ambiguous. The output *r* is set to a valid isolating interval (possibly just *start*) even if the algorithm fails.

9.26.4 Newton-based root finding

```
void arb_calc_newton_conv_factor(arb_t conv_factor, arb_calc_func_t func, void *param, const
                                arb_t conv_region, slong prec)
```

Given an interval *I* specified by *conv_region*, evaluates a bound for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, where *f* is the function specified by *func* and *param*. The bound is obtained by evaluating *f'(I)* and *f''(I)* directly. If *f* is ill-conditioned, *I* may need to be extremely precise in order to get an effective, finite bound for *C*.

```
int arb_calc_newton_step(arb_t xnew, arb_calc_func_t func, void *param, const arb_t x, const
                        arb_t conv_region, const arb_t conv_factor, slong prec)
```

Performs a single step with an interval version of Newton's method. The input consists of the function *f* specified by *func* and *param*, a ball $x = [m - r, m + r]$ known to contain a single root of *f*, a ball *I* (*conv_region*) containing *x* with an associated bound (*conv_factor*) for $C = \sup_{t,u \in I} \frac{1}{2} |f''(t)|/|f'(u)|$, and a working precision *prec*.

The Newton update consists of setting $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. The expression $m - f(m)/f'(m)$ is evaluated using ball arithmetic at a working precision of *prec* bits, and the rounding error during this evaluation is accounted for in the output. We now check that $x' \in I$ and $r' < r$. If both conditions are satisfied, we set *xnew* to *x'* and return *ARB_CALC_SUCCESS*. If either condition fails, we set *xnew* to *x* and return *ARB_CALC_NO_CONVERGENCE*, indicating that no progress is made.

```
int arb_calc_refine_root_newton(arb_t r, arb_calc_func_t func, void *param, const arb_t start,
                               const arb_t conv_region, const arb_t conv_factor, slong
                               eval_extra_prec, slong prec)
```

Refines a precise estimate of a single root of a function to high precision by performing several Newton steps, using nearly optimally chosen doubling precision steps.

The inputs are defined as for *arb_calc_newton_step*, except for the precision parameters: *prec* is the target accuracy and *eval_extra_prec* is the estimated number of guard bits that need to be added to evaluate the function accurately close to the root (for example, if the function is a

polynomial with large coefficients of alternating signs and Horner's rule is used to evaluate it, the extra precision should typically be approximately the bit size of the coefficients).

This function returns `ARB_CALC_SUCCESS` if all attempted Newton steps are successful (note that this does not guarantee that the computed root is accurate to *prec* bits, which has to be verified by the user), only that it is more accurate than the starting ball.

On failure, `ARB_CALC_IMPRECISE_INPUT` or `ARB_CALC_NO_CONVERGENCE` may be returned. In this case, *r* is set to a ball for the root which is valid but likely does not have full accuracy (it can possibly just be equal to the starting ball).

9.27 acb_calc.h – calculus with complex-valued functions

This module provides functions for operations of calculus over the complex numbers (intended to include root-finding, integration, and so on). The numerical integration code is described in [Joh2018a].

9.27.1 Types, macros and constants

type `acb_calc_func_t`

Typedef for a pointer to a function with signature:

```
int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
```

implementing a univariate complex function $f(z)$. The *param* argument may be used to pass through additional parameters to the function. The return value is reserved for future use as an error code. It can be assumed that *out* and *inp* are not aliased.

When called with *order* = 0, *func* should write to *out* the value of $f(z)$ at the point *inp*, evaluated at a precision of *prec* bits. In this case, *f* can be an arbitrary complex function, which may have branch cuts or even be non-holomorphic.

When called with *order* = *n* for $n \geq 1$, *func* should write to *out* the first *n* coefficients in the Taylor series expansion of $f(z)$ at the point *inp*, evaluated at a precision of *prec* bits. In this case, the implementation of *func* must verify that *f* is holomorphic on the complex interval defined by *z*, and set the coefficients in *out* to non-finite values otherwise.

For algorithms that do not rely on derivatives, *func* will always get called with *order* = 0 or *order* = 1, in which case the user only needs to implement evaluation of the direct function value $f(z)$ (without derivatives). With *order* = 1, *func* must verify holomorphicity (unlike the *order* = 0 case).

If *f* is built from field operations and meromorphic functions, then no special action is necessary when *order* is positive since division by zero or evaluation of builtin functions at poles automatically produces infinite enclosures. However, manual action is needed for bounded functions with branch cuts. For example, when evaluating $\sqrt{z}$, the output must be set to a non-finite value if *z* overlaps with the branch cut $[-\infty, 0]$. The easiest way to accomplish this is to use versions of basic functions (`sqrt`, `log`, `pow`, etc.) that test holomorphicity of their arguments individually.

Some functions with branch cut detection are available as builtins: see `acb_sqrt_analytic()`, `acb_rsqrtn_analytic()`, `acb_log_analytic()`, `acb_pow_analytic()`. It is not difficult to write custom functions of this type, using the following pattern:

```
/* Square root function on C with detection of the branch cut. */
void sqrt_analytic(acb_t res, const acb_t z, int analytic, slong prec)
{
    if (analytic &&
        arb_contains_zero(acb_imagref(z)) &&
        arb_contains_nonpositive(acb_realref(z)))
```

(continues on next page)

(continued from previous page)

```

{
    acb_indeterminate(res);
}
else
{
    acb_sqrt(res, z, prec);
}
}

```

The built-in methods `acb_real_abs()`, `acb_real_sgn()`, `acb_real_heaviside()`, `acb_real_floor()`, `acb_real_ceil()`, `acb_real_max()`, `acb_real_min()` provide piecewise holomorphic functions that are useful for integrating piecewise-defined real functions.

For example, here we define a piecewise holomorphic extension of the function $f(z) = \sqrt{[z]}$ (for simplicity, without implementing derivatives):

```

int func(acb_ptr out, const acb_t inp, void * param, slong order, slong prec)
{
    if (order > 1) flint_abort(); /* derivatives not implemented */

    acb_real_floor(out, inp, order != 0, prec);
    acb_sqrt_analytic(out, out, order != 0, prec);
    return 0;
}

```

(Here, `acb_real_sqrtpos()` may be slightly better if it is known that z will be nonnegative on the path.)

See the demo program `examples/integrals.c` for more examples.

9.27.2 Integration

```

int acb_calc_integrate(acb_t res, acb_calc_func_t func, void *param, const acb_t a, const acb_t
    b, slong rel_goal, const mag_t abs_tol, const acb_calc_integrate_opt_t
    options, slong prec)

```

Computes a rigorous enclosure of the integral

$$I = \int_a^b f(t) dt$$

where f is specified by $(func, param)$, following a straight-line path between the complex numbers a and b . For finite results, a , b must be finite and f must be bounded on the path of integration. To compute improper integrals, the user should therefore truncate the path of integration manually (or make a regularizing change of variables, if possible). Returns `ARB_CALC_SUCCESS` if the integration converged to the target accuracy on all subintervals, and returns `ARB_CALC_NO_CONVERGENCE` otherwise.

By default, the integrand $func$ will only be called with $order = 0$ or $order = 1$; that is, derivatives are not required.

- The integrand will be called with $order = 0$ to evaluate f normally on the integration path (either at a single point or on a subinterval). In this case, f is treated as a pointwise defined function and can have arbitrary discontinuities.
- The integrand will be called with $order = 1$ to evaluate f on a domain surrounding a segment of the integration path for the purpose of bounding the error of a quadrature formula. In this case, $func$ must verify that f is holomorphic on this domain (and output a non-finite value if it is not).

The integration algorithm combines direct interval enclosures, Gauss-Legendre quadrature where f is holomorphic, and adaptive subdivision. This strategy supports integrands with discontinuities while providing exponential convergence for typical piecewise holomorphic integrands.

The following parameters control accuracy:

- *rel_goal* - relative accuracy goal as a number of bits, i.e. target a relative error less than $\varepsilon_{rel} = 2^{-r}$ where $r = rel\_goal$ (note the sign: *rel_goal* should be nonnegative).
- *abs_tol* - absolute accuracy goal as a *mag_t* describing the error tolerance, i.e. target an absolute error less than $\varepsilon_{abs} = abs\_tol$.
- *prec* - working precision. This is the working precision used to evaluate the integrand and manipulate interval endpoints. As currently implemented, the algorithm does not attempt to adjust the working precision by itself, and adaptive control of the working precision must be handled by the user.

For typical usage, set *rel_goal* = *prec* and *abs_tol* = 2^{-prec} . It usually only makes sense to have *rel_goal* between 0 and *prec*.

The algorithm attempts to achieve an error of $\max(\varepsilon_{abs}, M\varepsilon_{rel})$ on each subinterval, where M is the magnitude of the integral. These parameters are only guidelines; the cumulative error may be larger than both the prescribed absolute and relative error goals, depending on the number of subdivisions, cancellation between segments of the integral, and numerical errors in the evaluation of the integrand.

To compute tiny integrals with high relative accuracy, one should set $\varepsilon_{abs} \approx M\varepsilon_{rel}$ where M is a known estimate of the magnitude. Setting ε_{abs} to 0 is also allowed, forcing use of a relative instead of an absolute tolerance goal. This can be handy for exponentially small or large functions of unknown magnitude. It is recommended to avoid setting ε_{abs} very small if possible since the algorithm might need many extra subdivisions to estimate M automatically; if the approximate magnitude can be estimated by some external means (for example if a midpoint-width or endpoint-width estimate is known to be accurate), providing an appropriate $\varepsilon_{abs} \approx M\varepsilon_{rel}$ will be more efficient.

If the integral has very large magnitude, setting the absolute tolerance to a corresponding large value is recommended for best performance, but it is not necessary for convergence since the absolute tolerance is increased automatically during the execution of the algorithm if the partial integrals are found to have larger error.

Additional options for the integration can be provided via the *options* parameter (documented below). To use all defaults, *NULL* can be passed for *options*.

Options for integration

type **acb_calc_integrate_opt_struct**

type **acb_calc_integrate_opt_t**

This structure contains several fields, explained below. An *acb_calc_integrate_opt_t* is defined as an array of *acb_calc_integrate_opt_struct* of length 1, permitting it to be passed by reference. An *acb_calc_integrate_opt_t* must be initialized before use, which sets all fields to 0 or *NULL*. For fields that have not been set to other values, the integration algorithm will choose defaults automatically (based on the precision and accuracy goals). This structure will most likely be extended in the future to accommodate more options.

slong **deg_limit**

Maximum quadrature degree for each subinterval. If a zero or negative value is provided, the limit is set to a default value which currently equals $0.5 \cdot \min(prec, rel\_goal) + 60$ for Gauss-Legendre quadrature. A higher quadrature degree can be beneficial for functions that are holomorphic on a large domain around the integration path and yet behave irregularly, such as oscillatory entire functions. The drawback of increasing the degree is that the precomputation time for quadrature nodes increases.

***slong* eval_limit**

Maximum number of function evaluations. If a zero or negative value is provided, the limit is set to a default value which currently equals $1000 \cdot \text{prec} + \text{prec}^2$. This is the main parameter used to limit the amount of work before aborting due to possible slow convergence or non-convergence. A lower limit allows aborting faster. A higher limit may be needed for integrands with many discontinuities or many singularities close to the integration path. This limit is only taken as a rough guideline, and the actual number of function evaluations may be slightly higher depending on the actual subdivisions.

***slong* depth_limit**

Maximum search depth for adaptive subdivision. Technically, this is not the limit on the local bisection depth but the limit on the number of simultaneously queued subintervals. If a zero or negative value is provided, the limit is set to the default value $2 \cdot \text{prec}$. Warning: memory usage may increase in proportion to this limit.

int use_heap

By default (if set to 0), new subintervals generated by adaptive bisection will be appended to the top of a stack. If set to 1, a binary heap will be used to maintain a priority queue where the subintervals with larger error have higher priority. This sometimes gives better results in case of convergence failure, but can lead to a much larger array of subintervals (requiring a higher *depth_limit*) when many global bisections are needed.

int verbose

If set to 1, some information about the overall integration process is printed to standard output. If set to 2, information about each subinterval is printed.

void **acb_calc_integrate_opt_init**(*acb_calc_integrate_opt_t* options)

Initializes *options* for use, setting all fields to 0 indicating default values.

9.27.3 Local integration algorithms

int **acb_calc_integrate_g1_auto_deg**(*acb_t* res, *slong* *num_eval, *acb_calc_func_t* func, void *param, const *acb_t* a, const *acb_t* b, const *mag_t* tol, *slong* deg_limit, int flags, *slong* prec)

Attempts to compute $I = \int_a^b f(t)dt$ using a single application of Gauss-Legendre quadrature with automatic determination of the quadrature degree so that the error is smaller than *tol*. Returns *ARB_CALC_SUCCESS* if the integral has been evaluated successfully or *ARB_CALC_NO_CONVERGENCE* if the tolerance could not be met. The total number of function evaluations is written to *num_eval*.

For the interval $[-1, 1]$, the error of the n -point Gauss-Legendre rule is bounded by

$$\left| I - \sum_{k=0}^{n-1} w_k f(x_k) \right| \leq \frac{64M}{15(\rho - 1)\rho^{2n-1}}$$

if f is holomorphic with $|f(z)| \leq M$ inside the ellipse E with foci ± 1 and semiaxes X and $Y = \sqrt{X^2 - 1}$ such that $\rho = X + Y$ with $\rho > 1$ [Tre2008].

For an arbitrary interval, we use $\int_a^b f(t)dt = \int_{-1}^1 g(t)dt$ where $g(t) = \Delta f(\Delta t + m)$, $\Delta = \frac{1}{2}(b - a)$, $m = \frac{1}{2}(a + b)$. With $I = [\pm X] + [\pm Y]i$, this means that we evaluate $\Delta f(\Delta I + m)$ to get the bound M . (An improvement would be to reduce the wrapping effect of rotating the ellipse when the path is not rectilinear).

We search for an X that makes the error small by trying steps 2^{2^k} . Larger X will give smaller $1/\rho^{2n-1}$ but larger M . If we try successive larger values of k , we can abort when $M = \infty$ since this either means that we have hit a singularity or a branch cut or that overestimation in the evaluation of f is becoming too severe.

9.27.4 Integration (old)

void **acb_calc_cauchy_bound**(*arb_t* bound, *acb_calc_func_t* func, void *param, const *acb_t* x, const *arb_t* radius, *slong* maxdepth, *slong* prec)

Sets *bound* to a ball containing the value of the integral

$$C(x, r) = \frac{1}{2\pi r} \oint_{|z-x|=r} |f(z)| dz = \int_0^1 |f(x + re^{2\pi it})| dt$$

where f is specified by (*func*, *param*) and r is given by *radius*. The integral is computed using a simple step sum. The integration range is subdivided until the order of magnitude of b can be determined (i.e. its error bound is smaller than its midpoint), or until the step length has been cut in half *maxdepth* times. This function is currently implemented completely naively, and repeatedly subdivides the whole integration range instead of performing adaptive subdivisions.

int **acb_calc_integrate_taylor**(*acb_t* res, *acb_calc_func_t* func, void *param, const *acb_t* a, const *acb_t* b, const *arf_t* inner_radius, const *arf_t* outer_radius, *slong* accuracy_goal, *slong* prec)

Computes the integral

$$I = \int_a^b f(t) dt$$

where f is specified by (*func*, *param*), following a straight-line path between the complex numbers a and b which both must be finite.

The integral is approximated by piecewise centered Taylor polynomials. Rigorous truncation error bounds are calculated using the Cauchy integral formula. More precisely, if the Taylor series of f centered at the point m is $f(m+x) = \sum_{n=0}^{\infty} a_n x^n$, then

$$\int f(m+x) = \left(\sum_{n=0}^{N-1} a_n \frac{x^{n+1}}{n+1} \right) + \left(\sum_{n=N}^{\infty} a_n \frac{x^{n+1}}{n+1} \right).$$

For sufficiently small x , the second series converges and its absolute value is bounded by

$$\sum_{n=N}^{\infty} \frac{C(m, R)}{R^n} \frac{|x|^{n+1}}{N+1} = \frac{C(m, R)Rx}{(R-x)(N+1)} \left(\frac{x}{R} \right)^N.$$

It is required that any singularities of f are isolated from the path of integration by a distance strictly greater than the positive value *outer_radius* (which is the integration radius used for the Cauchy bound). Taylor series step lengths are chosen so as not to exceed *inner_radius*, which must be strictly smaller than *outer_radius* for convergence. A smaller *inner_radius* gives more rapid convergence of each Taylor series but means that more series might have to be used. A reasonable choice might be to set *inner_radius* to half the value of *outer_radius*, giving roughly one accurate bit per term.

The truncation point of each Taylor series is chosen so that the absolute truncation error is roughly 2^{-p} where p is given by *accuracy_goal* (in the future, this might change to a relative accuracy). Arithmetic operations and function evaluations are performed at a precision of *prec* bits. Note that due to accumulation of numerical errors, both values may have to be set higher (and the endpoints may have to be computed more accurately) to achieve a desired accuracy.

This function chooses the evaluation points uniformly rather than implementing adaptive subdivision.

9.28 arb_fpwrap.h – floating-point wrappers of Arb mathematical functions

This module provides wrappers of Arb functions intended users who want accurate floating-point mathematical functions without necessarily caring about ball arithmetic. The wrappers take floating-point input, give floating-point output, and automatically increase the internal working precision to ensure that the output is accurate (in the rare case of failure, they output NaN along with an error code).

Warning: This module is experimental (as of Arb 2.21). It has not been extensively tested, and interfaces may change in the future.

Supported types:

- `double` and `complex_double` (53-bit precision)

Limitations:

- The wrappers currently only handle finite input and points where function value is finite. For example, they do not know that $\log(0) = -\infty$ or that $\exp(-\infty) = 0$. Singular input or output result in `FPWRAP_UNABLE` and a NaN output value. Evaluation of limit values may be implemented in the future for some functions.
- The wrappers currently treat `-0.0` as `+0.0`. Users who need to distinguish signs of zero, e.g. on branch cuts, currently need to do so manually.
- When requesting *correct rounding*, the wrappers can fail to converge in asymptotic or exact cases (where special algorithms are required).
- If the value is computed accurately internally but is too small to represent as a floating-point number, the result will be `-0.0` or `+0.0` (on underflow) or `-Inf` or `+Inf` (on overflow). Since the underflowed or overflowed result is the best possible floating-point approximation of the true value, this outcome is considered correct and the flag `FPWRAP_SUCCESS` is returned. In the future, return status flags may be added to indicate that underflow or overflow has occurred.
- Different rounding modes are not yet implemented.

9.28.1 Option and return flags

Functions return an `int` flag indicating the status.

`FPWRAP_SUCCESS`

Indicates an accurate result. (Up to inevitable underflow or overflow in the final conversion to a floating-point result; see above.)

This flag has the numerical value 0.

`FPWRAP_UNABLE`

Indicates failure (unable to achieve to target accuracy, possibly because of a singularity). The output is set to NaN.

This flag has the numerical value 1.

Functions take a *flags* parameter specifying optional rounding and termination behavior. This can be set to 0 to use defaults.

`FPWRAP_ACCURATE_PARTS`

For complex output, compute both real and imaginary parts to full relative accuracy. By default (if this flag is not set), complex results are computed to at least 53-bit accuracy as a whole, but if either the real or imaginary part is much smaller than the other, that part can have a large relative error. Setting this flag can result in slower evaluation or failure to converge in some cases.

This flag has the numerical value 1.

FPWRAP_CORRECT_ROUNDING

Guarantees *correct rounding*. By default (if this flag is not set), real results are accurate up to the rounding of the last bit, but the last bit is not guaranteed to be rounded optimally. Setting this flag can result in slower evaluation or failure to converge in some cases. Correct rounding automatically applies to both real and imaginary parts of complex numbers, so it is unnecessary to set both this flag and *FPWRAP_ACCURATE_PARTS*.

This flag has the numerical value 2.

FPWRAP_WORK_LIMIT

Multiplied by an integer, specifies the maximum working precision to use before giving up. With $n * \text{FPWRAP\_WORK\_LIMIT}$ added to *flags*, n levels of precision will be used. The default $n = 0$ is equivalent to $n = 8$, which for `double` means trying with a working precision of 64, 128, 256, 512, 1024, 2048, 4096, 8192 bits. With `flags = 2 * FPWRAP_WORK_LIMIT`, we only try 64 and 128 bits, and with `flags = 16 * FPWRAP_WORK_LIMIT` we go up to 2097152 bits.

This flag has the numerical value 65536.

9.28.2 Types

Outputs are passed by reference so that we can return status flags and so that the interface is uniform for functions with multiple outputs.

type `complex_double`

A struct of two `double` components (`real` and `imag`), used to represent a machine-precision complex number. We use this custom type instead of the complex types defined in `<complex.h>` since Arb does not depend on C99. Users should easily be able to convert to the C99 complex type since the layout in memory is identical.

9.28.3 Functions

Elementary functions

```
int arb_fpwrap_double_exp(double *res, double x, int flags)
int arb_fpwrap_cdouble_exp(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_exp1(double *res, double x, int flags)
int arb_fpwrap_cdouble_exp1(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_log(double *res, double x, int flags)
int arb_fpwrap_cdouble_log(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_log1p(double *res, double x, int flags)
int arb_fpwrap_cdouble_log1p(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_pow(double *res, double x, double y, int flags)
int arb_fpwrap_cdouble_pow(complex_double *res, complex_double x, complex_double y, int flags)

int arb_fpwrap_double_sqrt(double *res, double x, int flags)
int arb_fpwrap_cdouble_sqrt(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_rsqrtdouble(double *res, double x, int flags)
int arb_fpwrap_cdouble_rsqrtdouble(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_cbqrt(double *res, double x, int flags)
int arb_fpwrap_cdouble_cbqrt(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_sin(double *res, double x, int flags)
```

```
int arb_fpwrap_cdouble_sin(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_cos(double *res, double x, int flags)
int arb_fpwrap_cdouble_cos(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_tan(double *res, double x, int flags)
int arb_fpwrap_cdouble_tan(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_cot(double *res, double x, int flags)
int arb_fpwrap_cdouble_cot(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_sec(double *res, double x, int flags)
int arb_fpwrap_cdouble_sec(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_csc(double *res, double x, int flags)
int arb_fpwrap_cdouble_csc(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_sinc(double *res, double x, int flags)
int arb_fpwrap_cdouble_sinc(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_sin_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_sin_pi(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_cos_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_cos_pi(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_tan_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_tan_pi(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_cot_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_cot_pi(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_sinc_pi(double *res, double x, int flags)
int arb_fpwrap_cdouble_sinc_pi(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_asin(double *res, double x, int flags)
int arb_fpwrap_cdouble_asin(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_acos(double *res, double x, int flags)
int arb_fpwrap_cdouble_acos(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_atan(double *res, double x, int flags)
int arb_fpwrap_cdouble_atan(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_atan2(double *res, double x1, double x2, int flags)
int arb_fpwrap_double_asinh(double *res, double x, int flags)
int arb_fpwrap_cdouble_asinh(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_acosh(double *res, double x, int flags)
int arb_fpwrap_cdouble_acosh(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_atanh(double *res, double x, int flags)
int arb_fpwrap_cdouble_atanh(complex_double *res, complex_double x, int flags)
int arb_fpwrap_double_lambertw(double *res, double x, slong branch, int flags)
int arb_fpwrap_cdouble_lambertw(complex_double *res, complex_double x, slong branch, int flags)
```


Gamma, zeta and related functions

int `arb_fpwrap_double_rising`(double *res, double x, double n, int flags)

int `arb_fpwrap_cdouble_rising`(*complex_double* *res, *complex_double* x, *complex_double* n, int flags)

Rising factorial.

int `arb_fpwrap_double_gamma`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_gamma`(*complex_double* *res, *complex_double* x, int flags)

Gamma function.

int `arb_fpwrap_double_rgamma`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_rgamma`(*complex_double* *res, *complex_double* x, int flags)

Reciprocal gamma function.

int `arb_fpwrap_double_lgamma`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_lgamma`(*complex_double* *res, *complex_double* x, int flags)

Log-gamma function.

int `arb_fpwrap_double_digamma`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_digamma`(*complex_double* *res, *complex_double* x, int flags)

Digamma function.

int `arb_fpwrap_double_zeta`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_zeta`(*complex_double* *res, *complex_double* x, int flags)

Riemann zeta function.

int `arb_fpwrap_double_hurwitz_zeta`(double *res, double s, double z, int flags)

int `arb_fpwrap_cdouble_hurwitz_zeta`(*complex_double* *res, *complex_double* s, *complex_double* z, int flags)

Hurwitz zeta function.

int `arb_fpwrap_double_lerch_phi`(double *res, double z, double s, double a, int flags)

int `arb_fpwrap_cdouble_lerch_phi`(*complex_double* *res, *complex_double* z, *complex_double* s, *complex_double* a, int flags)

Lerch transcendent.

int `arb_fpwrap_double_barnes_g`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_barnes_g`(*complex_double* *res, *complex_double* x, int flags)

Barnes G-function.

int `arb_fpwrap_double_log_barnes_g`(double *res, double x, int flags)

int `arb_fpwrap_cdouble_log_barnes_g`(*complex_double* *res, *complex_double* x, int flags)

Logarithmic Barnes G-function.

int `arb_fpwrap_double_polygamma`(double *res, double s, double z, int flags)

int `arb_fpwrap_cdouble_polygamma`(*complex_double* *res, *complex_double* s, *complex_double* z, int flags)

Polygamma function.

int `arb_fpwrap_double_polylog`(double *res, double s, double z, int flags)

int `arb_fpwrap_cdouble_polylog`(*complex_double* *res, *complex_double* s, *complex_double* z, int flags)

Polylogarithm.

int `arb_fpwrap_cdouble_dirichlet_eta`(*complex_double* *res, *complex_double* s, int flags)

```

int arb_fpwrap_cdouble_riemann_xi(complex_double *res, complex_double s, int flags)
int arb_fpwrap_cdouble_hardy_theta(complex_double *res, complex_double z, int flags)
int arb_fpwrap_cdouble_hardy_z(complex_double *res, complex_double z, int flags)
int arb_fpwrap_cdouble_zeta_zero(complex_double *res, ulong n, int flags)

```

Error functions and exponential integrals

```

int arb_fpwrap_double_erf(double *res, double x, int flags)
int arb_fpwrap_cdouble_erf(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_erfc(double *res, double x, int flags)
int arb_fpwrap_cdouble_erfc(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_erfi(double *res, double x, int flags)
int arb_fpwrap_cdouble_erfi(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_erfinv(double *res, double x, int flags)
int arb_fpwrap_double_erfcinv(double *res, double x, int flags)

int arb_fpwrap_double_fresnel_s(double *res, double x, int normalized, int flags)
int arb_fpwrap_cdouble_fresnel_s(complex_double *res, complex_double x, int normalized, int
    flags)

int arb_fpwrap_double_fresnel_c(double *res, double x, int normalized, int flags)
int arb_fpwrap_cdouble_fresnel_c(complex_double *res, complex_double x, int normalized, int
    flags)

int arb_fpwrap_double_gamma_upper(double *res, double s, double z, int regularized, int flags)
int arb_fpwrap_cdouble_gamma_upper(complex_double *res, complex_double s, complex_double z, int
    regularized, int flags)

int arb_fpwrap_double_gamma_lower(double *res, double s, double z, int regularized, int flags)
int arb_fpwrap_cdouble_gamma_lower(complex_double *res, complex_double s, complex_double z, int
    regularized, int flags)

int arb_fpwrap_double_beta_lower(double *res, double a, double b, double z, int regularized, int
    flags)
int arb_fpwrap_cdouble_beta_lower(complex_double *res, complex_double a, complex_double b,
    complex_double z, int regularized, int flags)

int arb_fpwrap_double_exp_integral_e(double *res, double s, double z, int flags)
int arb_fpwrap_cdouble_exp_integral_e(complex_double *res, complex_double s, complex_double z,
    int flags)

int arb_fpwrap_double_exp_integral_ei(double *res, double x, int flags)
int arb_fpwrap_cdouble_exp_integral_ei(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_sin_integral(double *res, double x, int flags)
int arb_fpwrap_cdouble_sin_integral(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_cos_integral(double *res, double x, int flags)
int arb_fpwrap_cdouble_cos_integral(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_sinh_integral(double *res, double x, int flags)

```

```
int arb_fpwrap_cdouble_sinh_integral(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_cosh_integral(double *res, double x, int flags)
int arb_fpwrap_cdouble_cosh_integral(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_log_integral(double *res, double x, int offset, int flags)
int arb_fpwrap_cdouble_log_integral(complex_double *res, complex_double x, int offset, int flags)

int arb_fpwrap_double_dilog(double *res, double x, int flags)
int arb_fpwrap_cdouble_dilog(complex_double *res, complex_double x, int flags)
```

Bessel, Airy and Coulomb functions

```
int arb_fpwrap_double_bessel_j(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_j(complex_double *res, complex_double nu, complex_double x, int
    flags)

int arb_fpwrap_double_bessel_y(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_y(complex_double *res, complex_double nu, complex_double x, int
    flags)

int arb_fpwrap_double_bessel_i(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_i(complex_double *res, complex_double nu, complex_double x, int
    flags)

int arb_fpwrap_double_bessel_k(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_k(complex_double *res, complex_double nu, complex_double x, int
    flags)

int arb_fpwrap_double_bessel_k_scaled(double *res, double nu, double x, int flags)
int arb_fpwrap_cdouble_bessel_k_scaled(complex_double *res, complex_double nu,
    complex_double x, int flags)

int arb_fpwrap_double_airy_ai(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_ai(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_ai_prime(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_ai_prime(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_bi(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_bi(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_bi_prime(double *res, double x, int flags)
int arb_fpwrap_cdouble_airy_bi_prime(complex_double *res, complex_double x, int flags)

int arb_fpwrap_double_airy_ai_zero(double *res, ulong n, int flags)
int arb_fpwrap_double_airy_ai_prime_zero(double *res, ulong n, int flags)
int arb_fpwrap_double_airy_bi_zero(double *res, ulong n, int flags)
int arb_fpwrap_double_airy_bi_prime_zero(double *res, ulong n, int flags)

int arb_fpwrap_double_coulomb_f(double *res, double l, double eta, double x, int flags)
int arb_fpwrap_cdouble_coulomb_f(complex_double *res, complex_double l, complex_double eta,
    complex_double x, int flags)

int arb_fpwrap_double_coulomb_g(double *res, double l, double eta, double x, int flags)
```

```

int arb_fpwrap_cdouble_coulomb_g(complex_double *res, complex_double l, complex_double eta,
                                complex_double x, int flags)

int arb_fpwrap_cdouble_coulomb_hpos(complex_double *res, complex_double l, complex_double eta,
                                   complex_double x, int flags)

int arb_fpwrap_cdouble_coulomb_hneg(complex_double *res, complex_double l, complex_double eta,
                                   complex_double x, int flags)

```

Orthogonal polynomials

```

int arb_fpwrap_double_chebyshev_t(double *res, double n, double x, int flags)
int arb_fpwrap_cdouble_chebyshev_t(complex_double *res, complex_double n, complex_double x,
                                   int flags)

int arb_fpwrap_double_chebyshev_u(double *res, double n, double x, int flags)
int arb_fpwrap_cdouble_chebyshev_u(complex_double *res, complex_double n, complex_double x,
                                   int flags)

int arb_fpwrap_double_jacobi_p(double *res, double n, double a, double b, double x, int flags)
int arb_fpwrap_cdouble_jacobi_p(complex_double *res, complex_double n, complex_double a,
                                complex_double b, complex_double x, int flags)

int arb_fpwrap_double_gegenbauer_c(double *res, double n, double m, double x, int flags)
int arb_fpwrap_cdouble_gegenbauer_c(complex_double *res, complex_double n, complex_double m,
                                    complex_double x, int flags)

int arb_fpwrap_double_laguerre_l(double *res, double n, double m, double x, int flags)
int arb_fpwrap_cdouble_laguerre_l(complex_double *res, complex_double n, complex_double m,
                                  complex_double x, int flags)

int arb_fpwrap_double_hermite_h(double *res, double n, double x, int flags)
int arb_fpwrap_cdouble_hermite_h(complex_double *res, complex_double n, complex_double x, int
                                flags)

int arb_fpwrap_double_legendre_p(double *res, double n, double m, double x, int type, int flags)
int arb_fpwrap_cdouble_legendre_p(complex_double *res, complex_double n, complex_double m,
                                  complex_double x, int type, int flags)

int arb_fpwrap_double_legendre_q(double *res, double n, double m, double x, int type, int flags)
int arb_fpwrap_cdouble_legendre_q(complex_double *res, complex_double n, complex_double m,
                                  complex_double x, int type, int flags)

int arb_fpwrap_double_legendre_root(double *res1, double *res2, ulong n, ulong k, int flags)
    Sets res1 to the index k root of the Legendre polynomial  $P_n(x)$ , and simultaneously sets res2 to
    the corresponding weight for Gauss-Legendre quadrature.

int arb_fpwrap_cdouble_spherical_y(complex_double *res, slong n, slong m, complex_double x1,
                                   complex_double x2, int flags)

```

Hypergeometric functions

```
int arb_fpwrap_double_hypgeom_0f1(double *res, double a, double x, int regularized, int flags)
int arb_fpwrap_cdouble_hypgeom_0f1(complex_double *res, complex_double a, complex_double x,
    int regularized, int flags)

int arb_fpwrap_double_hypgeom_1f1(double *res, double a, double b, double x, int regularized, int
    flags)
int arb_fpwrap_cdouble_hypgeom_1f1(complex_double *res, complex_double a, complex_double b,
    complex_double x, int regularized, int flags)

int arb_fpwrap_double_hypgeom_u(double *res, double a, double b, double x, int flags)
int arb_fpwrap_cdouble_hypgeom_u(complex_double *res, complex_double a, complex_double b,
    complex_double x, int flags)

int arb_fpwrap_double_hypgeom_2f1(double *res, double a, double b, double c, double x, int
    regularized, int flags)
int arb_fpwrap_cdouble_hypgeom_2f1(complex_double *res, complex_double a, complex_double b,
    complex_double c, complex_double x, int regularized, int flags)

int arb_fpwrap_double_hypgeom_pfq(double *res, const double *a, slong p, const double *b, slong q,
    double z, int regularized, int flags)
int arb_fpwrap_cdouble_hypgeom_pfq(complex_double *res, const complex_double *a, slong p, const
    complex_double *b, slong q, complex_double z, int regularized,
    int flags)
```

Elliptic integrals, elliptic functions and modular forms

```
int arb_fpwrap_double_agm(double *res, double x, double y, int flags)
int arb_fpwrap_cdouble_agm(complex_double *res, complex_double x, complex_double y, int flags)
    Arithmetic-geometric mean.

int arb_fpwrap_cdouble_elliptic_k(complex_double *res, complex_double m, int flags)
int arb_fpwrap_cdouble_elliptic_e(complex_double *res, complex_double m, int flags)
int arb_fpwrap_cdouble_elliptic_pi(complex_double *res, complex_double n, complex_double m,
    int flags)
int arb_fpwrap_cdouble_elliptic_f(complex_double *res, complex_double phi, complex_double m,
    int pi, int flags)
int arb_fpwrap_cdouble_elliptic_e_inc(complex_double *res, complex_double phi, complex_double
    m, int pi, int flags)
int arb_fpwrap_cdouble_elliptic_pi_inc(complex_double *res, complex_double n, complex_double
    phi, complex_double m, int pi, int flags)
    Complete and incomplete elliptic integrals.

int arb_fpwrap_cdouble_elliptic_rf(complex_double *res, complex_double x, complex_double y,
    complex_double z, int option, int flags)
int arb_fpwrap_cdouble_elliptic_rg(complex_double *res, complex_double x, complex_double y,
    complex_double z, int option, int flags)
int arb_fpwrap_cdouble_elliptic_rj(complex_double *res, complex_double x, complex_double y,
    complex_double z, complex_double w, int option, int flags)
    Carlson symmetric elliptic integrals.
```

```
int arb_fpwrap_cdouble_elliptic_p(complex_double *res, complex_double z, complex_double tau,
                                int flags)
```

```
int arb_fpwrap_cdouble_elliptic_p_prime(complex_double *res, complex_double z, complex_double
                                         tau, int flags)
```

```
int arb_fpwrap_cdouble_elliptic_inv_p(complex_double *res, complex_double z, complex_double
                                       tau, int flags)
```

```
int arb_fpwrap_cdouble_elliptic_zeta(complex_double *res, complex_double z, complex_double
                                      tau, int flags)
```

```
int arb_fpwrap_cdouble_elliptic_sigma(complex_double *res, complex_double z, complex_double
                                       tau, int flags)
```

Weierstrass elliptic functions.

```
int arb_fpwrap_cdouble_jacobi_theta_1(complex_double *res, complex_double z, complex_double
                                       tau, int flags)
```

```
int arb_fpwrap_cdouble_jacobi_theta_2(complex_double *res, complex_double z, complex_double
                                       tau, int flags)
```

```
int arb_fpwrap_cdouble_jacobi_theta_3(complex_double *res, complex_double z, complex_double
                                       tau, int flags)
```

```
int arb_fpwrap_cdouble_jacobi_theta_4(complex_double *res, complex_double z, complex_double
                                       tau, int flags)
```

Jacobi theta functions.

```
int arb_fpwrap_cdouble_dedekind_eta(complex_double *res, complex_double tau, int flags)
```

```
int arb_fpwrap_cdouble_modular_j(complex_double *res, complex_double tau, int flags)
```

```
int arb_fpwrap_cdouble_modular_lambda(complex_double *res, complex_double tau, int flags)
```

```
int arb_fpwrap_cdouble_modular_delta(complex_double *res, complex_double tau, int flags)
```

9.28.4 Calling from C

The program `examples/fpwrap.c` provides a usage example:

```
#include "arb_fpwrap.h"

int main()
{
    double x, y;
    complex_double cx, cy;
    int flags = 0;    /* default options */

    x = 2.0;
    cx.real = 0.5;
    cx.imag = 123.0;

    arb_fpwrap_double_zeta(&y, x, flags);
    arb_fpwrap_cdouble_zeta(&cy, cx, flags);

    printf("zeta(%g) = %.16g\n", x, y);
    printf("zeta(%g + %gi) = %.16g + %.16gi\n", cx.real, cx.imag, cy.real, cy.imag);
}
```

(continues on next page)

(continued from previous page)

```
flint_cleanup();  
return 0;  
}
```

This should print:

```
> build/examples/fpwrap  
zeta(2) = 1.644934066848226  
zeta(0.5 + 123i) = 0.006252861175594465 + 0.08206030514520983i
```

Note that this program does not check the return flag to perform error handling.

9.28.5 Interfacing from Python

This illustrates how to call functions from Python using `ctypes`:

```
import ctypes  
import ctypes.util  
  
libarb_path = ctypes.util.find_library('arb')  
libarb = ctypes.CDLL(libarb_path)  
  
class _complex_double(ctypes.Structure):  
    _fields_ = [('real', ctypes.c_double),  
                ('imag', ctypes.c_double)]  
  
def wrap_double_fun(fun):  
    def f(x):  
        y = ctypes.c_double()  
        if fun(ctypes.byref(y), ctypes.c_double(x), 0):  
            raise ValueError(f"unable to evaluate function accurately at {x}")  
        return y.value  
    return f  
  
def wrap_cdouble_fun(fun):  
    def f(x):  
        x = complex(x)  
        cx = _complex_double()  
        cy = _complex_double()  
        cx.real = x.real  
        cx.imag = x.imag  
        if fun(ctypes.byref(cy), cx, 0):  
            raise ValueError(f"unable to evaluate function accurately at {x}")  
        return complex(cy.real, cy.imag)  
    return f  
  
zeta = wrap_double_fun(libarb.arb_fpwrap_double_zeta)  
czeta = wrap_cdouble_fun(libarb.arb_fpwrap_cdouble_zeta)  
  
print(zeta(2.0))  
print(czeta(0.5+1e9j))  
print(zeta(1.0))      # pole, where wrapper throws exception
```

This should print:


```
1.6449340668482264
(-2.761748029838061-1.6775122409894598j)
Traceback (most recent call last):
...
ValueError: unable to evaluate function accurately at 1.0
```

9.28.6 Interfacing from Julia

This illustrates how to call functions from Julia using `ccall`:

```
using Libdl

dlopen("/home/fredrik/src/arb/libarb.so")

function zeta(x::Float64)
    cy = Ref{Float64}{}
    if Bool(ccall((:arb_fpwrap_double_zeta, :libarb), Cint, (Ptr{Float64}, Float64, Cint), cy, x, 0))
        error("unable to evaluate accurately at ", x)
    end
    return cy[]
end

function zeta(x::Complex{Float64})
    cy = Ref{Complex{Float64}}{}
    if Bool(ccall((:arb_fpwrap_cdoube_zeta, :libarb), Cint, (Ptr{Complex{Float64}}, Complex{Float64}, Cint), cy, x, 0))
        error("unable to evaluate accurately at ", x)
    end
    return cy[]
end

println(zeta(2.0))
println(zeta(0.5 + 1e9im))
println(zeta(1.0))           # pole, where wrapper throws exception
```

This should print:

```
1.6449340668482264
-2.761748029838061 - 1.6775122409894598im
ERROR: unable to evaluate accurately at 1.0
Stacktrace:
...
```

9.29 fmpz_extras.h – extra methods for FLINT integers

This module implements a few utility methods for the FLINT multiprecision integer type (*fmpz_t*). It is mainly intended for internal use.

9.29.1 Memory-related methods

slong **fmpz_allocated_bytes**(const *fmpz_t* x)

Returns the total number of bytes heap-allocated internally by this object. The count excludes the size of the structure itself. Add `sizeof(fmpz)` to get the size of the object as a whole.

9.29.2 Convenience methods

void **fmpz_adiv_q_2exp**(*fmpz_t* z, const *fmpz_t* x, *flint_bitcnt_t* exp)

Sets z to $x/2^{\text{exp}}$, rounded away from zero.

void **fmpz_ui_mul_ui**(*fmpz_t* x, *ulong* a, *ulong* b)

Sets x to a times b .

void **fmpz_max**(*fmpz_t* z, const *fmpz_t* x, const *fmpz_t* y)

void **fmpz_min**(*fmpz_t* z, const *fmpz_t* x, const *fmpz_t* y)

Sets z to the maximum (respectively minimum) of x and y .

9.29.3 Inlined arithmetic

The *fmpz_t* bignum type uses an immediate representation for small integers, specifically when the absolute value is at most $2^{62} - 1$ (on 64-bit machines) or $2^{30} - 1$ (on 32-bit machines). The following methods completely inline the case where all operands (and possibly some intermediate values in the calculation) are known to be small. This is faster in code where all values *almost certainly will be much smaller than a full word*. In particular, these methods are used within Arb for manipulating exponents of floating-point numbers. Inlining slows down the general case, and increases code size, so these methods should not be used gratuitously.

void **fmpz_add_inline**(*fmpz_t* z, const *fmpz_t* x, const *fmpz_t* y)

void **fmpz_add_si_inline**(*fmpz_t* z, const *fmpz_t* x, *slong* y)

void **fmpz_add_ui_inline**(*fmpz_t* z, const *fmpz_t* x, *ulong* y)

Sets z to the sum of x and y .

void **fmpz_sub_si_inline**(*fmpz_t* z, const *fmpz_t* x, *slong* y)

Sets z to the difference of x and y .

void **fmpz_add2_fmpz_si_inline**(*fmpz_t* z, const *fmpz_t* x, const *fmpz_t* y, *slong* c)

Sets z to the sum of x , y , and c .

slong **_fmpz_size**(const *fmpz_t* x)

Returns the number of limbs required to represent x .

slong **_fmpz_sub_small**(const *fmpz_t* x, const *fmpz_t* y)

Computes the difference of x and y and returns the result as an *slong*. The result is clamped between $-WORD\_MAX$ and $WORD\_MAX$, i.e. between $\pm(2^{63} - 1)$ inclusive on a 64-bit machine.

void **_fmpz_set_si_small**(*fmpz_t* x, *slong* v)

Sets x to the integer v which is required to be a value between $COEFF\_MIN$ and $COEFF\_MAX$ so that promotion to a bignum cannot occur.

9.29.4 Low-level conversions

FMPZ_GET_MPN_READONLY(*zsign*, *zn*, *zptr*, *ztmp*, *zv*)

Given an *fmpz_t* *zv*, this macro sets *zptr* to a pointer to the limbs of *zv*, *zn* to the number of limbs, and *zsign* to a sign bit (0 if nonnegative, 1 if negative). The variable *ztmp* must be a single *ulong*, which is used as a buffer. If *zv* is a small value, *zv* itself contains no limb array that *zptr* could point to, so the single limb is copied to *ztmp* and *zptr* is set to point to *ztmp*. The case where *zv* is zero is not handled specially, and *zn* is set to 1.

void **fmpz_lshift_mpn**(*fmpz_t* *z*, *nn_srcptr* *src*, *slong* *n*, int *negative*, *flint_bitcnt_t* *shift*)

Sets *z* to the integer represented by the *n* limbs in the array *src*, or minus this value if *negative* is 1, shifted left by *shift* bits. Requires *n* ≥ 1 and that the top limb of *src* is nonzero.

9.30 General formulas and bounds

This section collects some results from real and complex analysis that are useful when deriving error bounds. Beware of typos.

9.30.1 Error propagation

We want to bound the error when $f(x+a)$ is approximated by $f(x)$. Specifically, the goal is to bound $f(x+a) - f(x)$ in terms of r for the set of values a with $|a| \leq r$. Most bounds will be monotone increasing with $|a|$ (assuming that x is fixed), so for brevity we simply express the bounds in terms of $|a|$.

Theorem (generic first-order bound):

$$|f(x+a) - f(x)| \leq \min(2C_0, C_1|a|)$$

where

$$C_0 = \sup_{|t| \leq |a|} |f(x+t)|, \quad C_1 = \sup_{|t| \leq |a|} |f'(x+t)|.$$

The statement is valid with either $a, t \in \mathbb{R}$ or $a, t \in \mathbb{C}$.

Theorem (product): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$,

$$|(x+a)(y+b) - xy| \leq |xb| + |ya| + |ab|.$$

Theorem (quotient): For $x, y \in \mathbb{C}$ and $a, b \in \mathbb{C}$ with $|b| < |y|$,

$$\left| \frac{x}{y} - \frac{x+a}{y+b} \right| \leq \frac{|xb| + |ya|}{|y|(|y| - |b|)}.$$

Theorem (square root): For $x, a \in \mathbb{R}$ with $0 \leq |a| \leq x$,

$$|\sqrt{x+a} - \sqrt{x}| \leq \sqrt{x} \left(1 - \sqrt{1 - \frac{|a|}{x}} \right) \leq \frac{\sqrt{x}}{2} \left(\frac{|a|}{x} + \frac{|a|^2}{x^2} \right)$$

where the first inequality is an equality if $a \leq 0$. (When $x = a = 0$, the limiting value is 0.)

Theorem (reciprocal square root): For $x, a \in \mathbb{R}$ with $0 \leq |a| < x$,

$$\left| \frac{1}{\sqrt{x+a}} - \frac{1}{\sqrt{x}} \right| \leq \frac{|a|}{2(x - |a|)^{3/2}}.$$

Theorem (k-th root): For $k > 1$ and $x, a \in \mathbb{R}$ with $0 \leq |a| \leq x$,

$$\left| (x+a)^{1/k} - x^{1/k} \right| \leq x^{1/k} \min \left(1, \frac{1}{k} \log \left(1 + \frac{|a|}{x - |a|} \right) \right).$$

Proof: The error is largest when $a = -r$ is negative, and

$$\begin{aligned} x^{1/k} - (x-r)^{1/k} &= x^{1/k} [1 - (1 - r/x)^{1/k}] \\ &= x^{1/k} [1 - \exp(\log(1 - r/x)/k)] \leq x^{1/k} \min(1, -\log(1 - r/x)/k) \\ &= x^{1/k} \min(1, \log(1 + r/(x-r))/k). \end{aligned}$$

Theorem (sine, cosine): For $x, a \in \mathbb{R}$, $|\sin(x+a) - \sin(x)| \leq \min(2, |a|)$.

Theorem (logarithm): For $x, a \in \mathbb{R}$ with $0 \leq |a| < x$,

$$|\log(x+a) - \log(x)| \leq \log \left(1 + \frac{|a|}{x-|a|} \right),$$

with equality if $a \leq 0$.

Theorem (exponential): For $x, a \in \mathbb{R}$, $|e^{x+a} - e^x| = e^x(e^a - 1) \leq e^x(e^{|a|} - 1)$, with equality if $a \geq 0$.

Theorem (inverse tangent): For $x, a \in \mathbb{R}$,

$$|\operatorname{atan}(x+a) - \operatorname{atan}(x)| \leq \min(\pi, C_1|a|).$$

where

$$C_1 = \sup_{|t| \leq |a|} \frac{1}{1 + (x+t)^2}.$$

If $|a| < |x|$, then $C_1 = (1 + (|x| - |a|)^2)^{-1}$ gives a monotone bound.

An exact bound: if $|a| < |x|$ or $|x(x+a)| < 1$, then

$$|\operatorname{atan}(x+a) - \operatorname{atan}(x)| = \operatorname{atan} \left(\frac{|a|}{1 + x(x+a)} \right).$$

In the last formula, a case distinction has to be made depending on the signs of x and a .

9.30.2 Sums and series

Theorem (geometric bound): If $|c_k| \leq C$ and $|z| \leq D < 1$, then

$$\left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1-D}.$$

Theorem (integral bound): If $f(x)$ is nonnegative and monotone decreasing, then

$$\int_N^{\infty} f(x) \leq \sum_{k=N}^{\infty} f(k) \leq f(N) + \int_N^{\infty} f(x) dx.$$

9.30.3 Complex analytic functions

Theorem (Cauchy's integral formula): If $f(z) = \sum_{k=0}^{\infty} c_k z^k$ is analytic (on an open subset of $\mathbb{C}$ containing the disk $D = \{z : |z| \leq R\}$ in its interior, where $R > 0$), then

$$c_k = \frac{1}{2\pi i} \int_{|z|=R} \frac{f(z)}{z^{k+1}} dz.$$

Corollary (derivative bound):

$$|c_k| \leq \frac{C}{R^k}, \quad C = \max_{|z|=R} |f(z)|.$$

Corollary (Taylor series tail): If $0 \leq r < R$ and $|z| \leq r$, then

$$\left| \sum_{k=N}^{\infty} c_k z^k \right| \leq \frac{CD^N}{1-D}, \quad D = \left| \frac{r}{R} \right|.$$

9.30.4 Euler-Maclaurin formula

Theorem (Euler-Maclaurin): If $f(t)$ is $2M$ -times differentiable, then

$$\begin{aligned}\sum_{k=L}^U f(k) &= S + I + T + R \\ S &= \sum_{k=L}^{N-1} f(k), \quad I = \int_N^U f(t) dt, \\ T &= \frac{1}{2} (f(N) + f(U)) + \sum_{k=1}^M \frac{B_{2k}}{(2k)!} \left(f^{(2k-1)}(U) - f^{(2k-1)}(N) \right), \\ R &= - \int_N^U \frac{B_{2M}(t - \lfloor t \rfloor)}{(2M)!} f^{(2M)}(t) dt.\end{aligned}$$

Lemma (Bernoulli polynomials): $|B_n(t - \lfloor t \rfloor)| \leq 4n!/(2\pi)^n$.

Theorem (remainder bound):

$$|R| \leq \frac{4}{(2\pi)^{2M}} \int_N^U |f^{(2M)}(t)| dt.$$

Theorem (parameter derivatives): If $f(t) = f(t, x) = \sum_{k=0}^{\infty} a_k(t)x^k$ and $R = R(x) = \sum_{k=0}^{\infty} c_k x^k$ are analytic functions of x , then

$$|c_k| \leq \frac{4}{(2\pi)^{2M}} \int_N^U |a_k^{(2M)}(t)| dt.$$

9.31 Algorithms for mathematical constants

Most mathematical constants are evaluated using the generic hypergeometric summation code.

9.31.1 Pi

π is computed using the Chudnovsky series

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)! (k!)^3 640320^{3k+3/2}}$$

which is hypergeometric and adds roughly 14 digits per term. Methods based on the arithmetic-geometric mean seem to be slower by a factor three in practice.

A small trick is to compute $1/\sqrt{640320}$ instead of $\sqrt{640320}$ at the end.

9.31.2 Logarithms of integers

The standalone constant $\log(2)$ is computed using Zuniga's series [Zun2023b]

$$\log(2) = \frac{1}{2} \sum_{n=1}^{\infty} \frac{1}{3888^n} \frac{(1794n - 297)}{n(2n-1)} \frac{n! (\frac{1}{2})_n}{(\frac{1}{6})_n (\frac{5}{6})_n}.$$

Logarithms of other small integers are in certain situations computed using Machin-like formulas, e.g.:

$$\log(10) = 46 \operatorname{atanh}(1/31) + 34 \operatorname{atanh}(1/49) + 20 \operatorname{atanh}(1/161)$$

9.31.3 Euler's constant

Euler's constant γ is computed using the Brent-McMillan formula ([BM1980], [MPFR2012])

$$\gamma = \frac{S_0(2n) - K_0(2n)}{I_0(2n)} - \log(n)$$

in which n is a free parameter and

$$S_0(x) = \sum_{k=0}^{\infty} \frac{H_k}{(k!)^2} \left(\frac{x}{2}\right)^{2k}, \quad I_0(x) = \sum_{k=0}^{\infty} \frac{1}{(k!)^2} \left(\frac{x}{2}\right)^{2k}$$

$$2xI_0(x)K_0(x) \sim \sum_{k=0}^{\infty} \frac{[(2k)!]^3}{(k!)^4 8^{2k} x^{2k}}.$$

All series are evaluated using binary splitting. The first two series are evaluated simultaneously, with the summation taken up to $k = N - 1$ inclusive where $N \geq \alpha n + 1$ and $\alpha \approx 4.9706257595442318644$ satisfies $\alpha(\log \alpha - 1) = 3$. The third series is taken up to $k = 2n - 1$ inclusive. With these parameters, it is shown in [BJ2013] that the error is bounded by $24e^{-8n}$.

9.31.4 Catalan's constant

Catalan's constant is computed using the hypergeometric series

$$C = \frac{1}{768} \sum_{k=1}^{\infty} \frac{(-4096)^k P(k)}{k^3(2k-1)(3k-1)(3k-2)(6k-1)(6k-5) \binom{5k}{k} \binom{10k}{5k} \binom{12k}{6k}}$$

where

$$P(k) = -43203456k^6 + 92809152k^5 - 76613904k^4 + 30494304k^3 - 6004944k^2 + 536620k - 17325,$$

discovered by Zuniga [Zun2023]. It was previously computed using a series given in [PP2010].

9.31.5 Apéry's constant

Apéry's constant $\zeta(3)$ is computed using the hypergeometric series

$$\zeta(3) = \frac{1}{48} \sum_{k=1}^{\infty} \frac{(-1)^{k-1} P(k)}{k^5(2k-1)^3(3k-1)(3k-2)(4k-1)(4k-3)(6k-1)(6k-5) \binom{5k}{k} \binom{5k}{2k} \binom{9k}{4k} \binom{10k}{5k} \binom{12k}{6k}}$$

where

$$P(k) = 1565994397644288k^{11} - 6719460725627136k^{10} + 12632254526031264k^9 - 13684352515879536k^8 + 9451223531851808k^7 - 4348596587040104k^6 + 1352700034136826k^5 - 282805786014979k^4 + 38721705264979k^3 - 3292502315430k^2 + 156286859400k - 3143448000,$$

discovered by Zuniga [Zun2023].

9.31.6 Khinchin's constant

Khinchin's constant K_0 is computed using the formula

$$\log K_0 = \frac{1}{\log 2} \left[\sum_{k=2}^{N-1} \log \left(\frac{k-1}{k} \right) \log \left(\frac{k+1}{k} \right) + \sum_{n=1}^{\infty} \frac{\zeta(2n, N)}{n} \sum_{k=1}^{2n-1} \frac{(-1)^{k+1}}{k} \right]$$

where $N \geq 2$ is a free parameter that can be used for tuning [BBC1997]. If the infinite series is truncated after $n = M$, the remainder is smaller in absolute value than

$$\begin{aligned} \sum_{n=M+1}^{\infty} \zeta(2n, N) &= \sum_{n=M+1}^{\infty} \sum_{k=0}^{\infty} (k+N)^{-2n} \leq \sum_{n=M+1}^{\infty} \left(N^{-2n} + \int_0^{\infty} (t+N)^{-2n} dt \right) \\ &= \sum_{n=M+1}^{\infty} \frac{1}{N^{2n}} \left(1 + \frac{N}{2n-1} \right) \leq \sum_{n=M+1}^{\infty} \frac{N+1}{N^{2n}} = \frac{1}{N^{2M}(N-1)} \leq \frac{1}{N^{2M}}. \end{aligned}$$

Thus, for an error of at most 2^{-p} in the series, it is sufficient to choose $M \geq p/(2 \log_2 N)$.

9.31.7 Glaisher's constant

Glaisher's constant $A = \exp(1/12 - \zeta'(-1))$ is computed directly from this formula. We don't use the reflection formula for the zeta function, as the arithmetic in Euler-Maclaurin summation is faster at $s = -1$ than at $s = 2$.

9.31.8 Reciprocal Fibonacci constant

We use Gosper's series ([Gos1974], corrected in [Arn2012])

$$\sum_{n=1}^{\infty} \frac{1}{F_n} = \sum_{n=0}^{\infty} \frac{(-1)^{n(n-1)/2} (F_{4n+3} + (-1)^n F_{2n+2})}{F_{2n+1} F_{2n+2} L_1 L_3 \cdots L_{2n+1}}$$

where $L_n = 2F_{n-1} + F_n$ denotes a Lucas number. The truncation error after $N \geq 1$ terms is bounded by $(1/\phi)^{N^2}$. The series is not of hypergeometric type, but we can evaluate it in quasilinear time using binary splitting; factoring out a multiplicative recurrence for $L_1 L_3 \cdots$ allows computing the series as a product of $O(\sqrt{p})$ matrices with $O(\sqrt{p})$ -bit entries.

9.32 Algorithms for the gamma function

9.32.1 The Stirling series

In general, the gamma function is computed via the Stirling series

$$\log \Gamma(z) = \left(z - \frac{1}{2} \right) \log z - z + \frac{\ln 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)z^{2k-1}} + R(n, z)$$

where ([Olv1997] pp. 293-295) the remainder term is exactly

$$R_n(z) = \int_0^{\infty} \frac{B_{2n} - \tilde{B}_{2n}(x)}{2n(x+z)^{2n}} dx.$$

To evaluate the gamma function of a power series argument, we substitute $z \rightarrow z + t \in \mathbb{C}[[t]]$.

Using the bound for $|x + z|$ given by [Olv1997] and the fact that the numerator of the integrand is bounded in absolute value by $2|B_{2n}|$, the remainder can be shown to satisfy the bound

$$|[t^k]R_n(z+t)| \leq 2|B_{2n}| \frac{\Gamma(2n+k-1)}{\Gamma(k+1)\Gamma(2n+1)} |z| \left(\frac{b}{|z|} \right)^{2n+k}$$

where $b = 1/\cos(\arg(z)/2)$. Note that by trigonometric identities, assuming that $z = x + yi$, we have $b = \sqrt{1+u^2}$ where

$$u = \frac{y}{\sqrt{x^2+y^2}+x} = \frac{\sqrt{x^2+y^2}-x}{y}.$$

To use the Stirling series at p -bit precision, we select parameters r, n such that the remainder $R(n, z)$ approximately is bounded by 2^{-p} . If $|z|$ is too small for the Stirling series to give sufficient accuracy directly, we first translate to $z + r$ using the formula $\Gamma(z) = \Gamma(z + r)/(z(z + 1)(z + 2) \cdots (z + r - 1))$.

To obtain a remainder smaller than 2^{-p} , we must choose an r such that, in the real case, $z + r > \beta p$, where $\beta > \log(2)/(2\pi) \approx 0.11$. In practice, a slightly larger factor $\beta \approx 0.2$ more closely balances n and r . A much larger β (e.g. $\beta = 1$) could be used to reduce the number of Bernoulli numbers that have to be precomputed, at the expense of slower repeated evaluation.

9.32.2 Rational arguments

We use efficient methods to compute $y = \Gamma(p/q)$ where q is one of 1, 2, 3, 4, 6 and p is a small integer.

The cases $\Gamma(1) = 1$ and $\Gamma(1/2) = \sqrt{\pi}$ are trivial. We reduce all remaining cases to $\Gamma(1/3)$ or $\Gamma(1/4)$ using the following relations:

$$\begin{aligned}\Gamma(2/3) &= \frac{2\pi}{3^{1/2}\Gamma(1/3)}, & \Gamma(3/4) &= \frac{2^{1/2}\pi}{\Gamma(1/4)}, \\ \Gamma(1/6) &= \frac{\Gamma(1/3)^2}{(\pi/3)^{1/2}2^{1/3}}, & \Gamma(5/6) &= \frac{2\pi(\pi/3)^{1/2}2^{1/3}}{\Gamma(1/3)^2}.\end{aligned}$$

We compute $\Gamma(1/3)$ and $\Gamma(1/4)$ rapidly to high precision using

$$\frac{\pi}{\Gamma(1/3)^3} = \frac{1}{960 \cdot 10^{1/4}} \sum_{k=0}^{\infty} \left(-\frac{3^2}{2^9 \cdot 5^3} \right)^k (9108k + 279) \frac{(1/12)_k (5/12)_k}{(k!)^2}, \quad \Gamma(1/4) = \sqrt{\frac{(2\pi)^{3/2}}{\text{agm}(1, \sqrt{2})}}.$$

where the infinite series is due to Guillera. An alternative formula which could be used for $\Gamma(1/3)$ is

$$\Gamma(1/3) = \frac{2^{4/9}\pi^{2/3}}{3^{1/12} \left(\text{agm} \left(1, \frac{1}{2}\sqrt{2 + \sqrt{3}} \right) \right)^{1/3}},$$

but this appears to be slightly slower in practice.

9.33 Algorithms for the Hurwitz zeta function

9.33.1 Euler-Maclaurin summation

The Euler-Maclaurin formula allows evaluating the Hurwitz zeta function and its derivatives for general complex input. The algorithm is described in [Joh2013].

9.33.2 Parameter Taylor series

To evaluate $\zeta(s, a)$ for several nearby parameter values, the following Taylor expansion is useful:

$$\zeta(s, a + x) = \sum_{k=0}^{\infty} (-x)^k \frac{(s)_k}{k!} \zeta(s + k, a)$$

We assume that $a \geq 1$ is real and that $\sigma = \text{re}(s)$ with $K + \sigma > 1$. The tail is bounded by

$$\sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \zeta(\sigma + k, a) \leq \sum_{k=K}^{\infty} |x|^k \frac{|(s)_k|}{k!} \left[\frac{1}{a^{\sigma+k}} + \frac{1}{(\sigma + k - 1)a^{\sigma+k-1}} \right].$$

Denote the term on the right by $T(k)$. Then

$$\left| \frac{T(k+1)}{T(k)} \right| = \frac{|x|}{a} \frac{(k + \sigma - 1)}{(k + \sigma)} \frac{(k + \sigma + a)}{(k + \sigma + a - 1)} \frac{|k + s|}{(k + 1)} \leq \frac{|x|}{a} \left(1 + \frac{1}{K + \sigma + a - 1} \right) \left(1 + \frac{|s - 1|}{K + 1} \right) = C$$

and if $C < 1$,

$$\sum_{k=K}^{\infty} T(k) \leq \frac{T(K)}{1-C}.$$

9.34 Algorithms for polylogarithms

The polylogarithm is defined for $s, z \in \mathbb{C}$ with $|z| < 1$ by

$$\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} \frac{z^k}{k^s}$$

and for $|z| \geq 1$ by analytic continuation, except for the singular point $z = 1$.

9.34.1 Computation for small z

The power sum converges rapidly when $|z| \ll 1$. To compute the series expansion with respect to s , we substitute $s \rightarrow s + x \in \mathbb{C}[[x]]$ and obtain

$$\mathrm{Li}_{s+x}(z) = \sum_{d=0}^{\infty} x^d \frac{(-1)^d}{d!} \sum_{k=1}^{\infty} T(k)$$

where

$$T(k) = \frac{z^k \log^d(k)}{k^s}.$$

The remainder term $|\sum_{k=N}^{\infty} T(k)|$ is bounded via the following strategy, implemented in `mag_polylog_tail()`.

Denote the terms by $T(k)$. We pick a nonincreasing function $U(k)$ such that

$$\frac{T(k+1)}{T(k)} = z \left(\frac{k}{k+1} \right)^s \left(\frac{\log(k+1)}{\log(k)} \right)^d \leq U(k).$$

Then, as soon as $U(N) < 1$,

$$\sum_{k=N}^{\infty} T(k) \leq T(N) \sum_{k=0}^{\infty} U(N)^k = \frac{T(N)}{1-U(N)}.$$

In particular, we take

$$U(k) = z B(k, \max(0, -s)) B(k \log(k), d)$$

where $B(m, n) = (1 + 1/m)^n$. This follows from the bounds

$$\left(\frac{k}{k+1} \right)^s \leq \begin{cases} 1 & \text{if } s \geq 0 \\ (1 + 1/k)^{-s} & \text{if } s < 0. \end{cases}$$

and

$$\left(\frac{\log(k+1)}{\log(k)} \right)^d \leq \left(1 + \frac{1}{k \log(k)} \right)^d.$$

9.34.2 Expansion for general z

For general complex s, z , we write the polylogarithm as a sum of two Hurwitz zeta functions

$$\mathrm{Li}_s(z) = \frac{\Gamma(v)}{(2\pi)^v} \left[i^v \zeta \left(v, \frac{1}{2} + \frac{\log(-z)}{2\pi i} \right) + i^{-v} \zeta \left(v, \frac{1}{2} - \frac{\log(-z)}{2\pi i} \right) \right]$$

in which $s = 1 - v$. With the principal branch of $\log(-z)$, we obtain the conventional analytic continuation of the polylogarithm with a branch cut on $z \in (1, +\infty)$.

To compute the series expansion with respect to v , we substitute $v \rightarrow v + x \in \mathbb{C}[[x]]$ in this formula (at the end of the computation, we map $x \rightarrow -x$ to obtain the power series for $\mathrm{Li}_{s+x}(z)$). The right hand side becomes

$$\Gamma(v + x)[E_1 Z_1 + E_2 Z_2]$$

where $E_1 = (i/(2\pi))^{v+x}$, $Z_1 = \zeta(v + x, \dots)$, $E_2 = (1/(2\pi i))^{v+x}$, $Z_2 = \zeta(v + x, \dots)$.

When $v = 1$, the Z_1 and Z_2 terms become Laurent series with a leading $1/x$ term. In this case, we compute the deflated series $\tilde{Z}_1, \tilde{Z}_2 = \zeta(x, \dots) - 1/x$. Then

$$E_1 Z_1 + E_2 Z_2 = (E_1 + E_2)/x + E_1 \tilde{Z}_1 + E_2 \tilde{Z}_2.$$

Note that $(E_1 + E_2)/x$ is a power series, since the constant term in $E_1 + E_2$ is zero when $v = 1$. So we simply compute one extra derivative of both E_1 and E_2 , and shift them one step. When $v = 0, -1, -2, \dots$, the $\Gamma(v + x)$ prefactor has a pole. In this case, we proceed analogously and formally multiply $x \Gamma(v + x)$ with $[E_1 Z_1 + E_2 Z_2]/x$.

Note that the formal cancellation only works when the order s (or v) is an exact integer: it is not currently possible to use this method when s is a small ball containing any of $0, 1, 2, \dots$ (then the result becomes indeterminate).

The Hurwitz zeta method becomes inefficient when $|z| \rightarrow 0$ (it gives an indeterminate result when $z = 0$). This is not a problem since we just use the defining series for the polylogarithm in that region. It also becomes inefficient when $|z| \rightarrow \infty$, for which an asymptotic expansion would better.

9.35 Algorithms for hypergeometric functions

The algorithms used to compute hypergeometric functions are described in [Joh2016]. Here, we state the most important error bounds.

9.35.1 Convergent series

Let

$$T(k) = \frac{\prod_{i=0}^{p-1} (a_i)_k}{\prod_{i=0}^{q-1} (b_i)_k} z^k.$$

We compute a factor C such that

$$\left| \sum_{k=n}^{\infty} T(k) \right| \leq C |T(n)|.$$

We check that $\mathrm{Re}(b + n) > 0$ for all lower parameters b . If this does not hold, C is set to infinity. Otherwise, we cancel out pairs of parameters a and b against each other. We have

$$\left| \frac{a + k}{b + k} \right| = \left| 1 + \frac{a - b}{b + k} \right| \leq 1 + \frac{|a - b|}{|b + n|}$$

and

$$\left| \frac{1}{b+k} \right| \leq \frac{1}{|b+n|}$$

for all $k \geq n$. This gives us a constant D such that $T(k+1) \leq DT(k)$ for all $k \geq n$. If $D \geq 1$, we set C to infinity. Otherwise, we take $C = \sum_{k=0}^{\infty} D^k = (1-D)^{-1}$.

9.35.2 Convergent series of power series

The same principle is used to get tail bounds for with $a_i, b_i, z \in \mathbb{C}[[x]]$, or more precisely, bounds for each coefficient in $\sum_{k=N}^{\infty} T(k) \in \mathbb{C}[[x]]/\langle x^n \rangle$ given $a_i, b_i, z \in \mathbb{C}[[x]]/\langle x^n \rangle$. First, we fix some notation, assuming that A and B are power series:

- $A_{[k]}$ denotes the coefficient of x^k in A , and $A_{[m:n]}$ denotes the power series $\sum_{k=m}^{n-1} A_{[k]}x^k$.
- $|A|$ denotes $\sum_{k=0}^{\infty} |A_{[k]}|x^k$ (this can be viewed as an element of $\mathbb{R}_{\geq 0}[[x]]$).
- $A \leq B$ signifies that $|A|_{[k]} \leq |B|_{[k]}$ holds for all k .
- We define $\mathcal{R}(B) = |B_{[0]}| - |B_{[1:\infty]}|$.

Using the formulas

$$(AB)_{[k]} = \sum_{j=0}^k A_{[j]}B_{[k-j]}, \quad (1/B)_{[k]} = \frac{1}{B_{[0]}} \sum_{j=1}^k -B_{[j]}(1/B)_{[k-j]},$$

it is easy to prove the following bounds for the coefficients of sums, products and quotients of formal power series:

$$|A+B| \leq |A| + |B|, \quad |AB| \leq |A||B|, \quad |A/B| \leq |A|/\mathcal{R}(B).$$

If $p \leq q$ and $\operatorname{Re}(b_{i[0]} + N) > 0$ for all b_i , then we may take

$$D = |z| \prod_{i=1}^p \left(1 + \frac{|a_i - b_i|}{\mathcal{R}(b_i + N)} \right) \prod_{i=p+1}^q \frac{1}{\mathcal{R}(b_i + N)}.$$

If $D_{[0]} < 1$, then $(1-D)^{-1}|T(n)|$ gives the error bound.

Note when adding and multiplying power series with (complex) interval coefficients, we can use point-valued upper bounds for the absolute values instead of performing interval arithmetic throughout. For $\mathcal{R}(B)$, we must then pick a lower bound for $|B_{[0]}|$ and upper bounds for the coefficients of $|B_{[1:\infty]}|$.

9.35.3 Asymptotic series for the confluent hypergeometric function

Let $U(a, b, z)$ denote the confluent hypergeometric function of the second kind with the principal branch cut, and let $U^* = z^a U(a, b, z)$. For all $z \neq 0$ and $b \notin \mathbb{Z}$ (but valid for all b as a limit), we have (DLMF 13.2.42)

$$U(a, b, z) = \frac{\Gamma(1-b)}{\Gamma(a-b+1)} M(a, b, z) + \frac{\Gamma(b-1)}{\Gamma(a)} z^{1-b} M(a-b+1, 2-b, z).$$

Moreover, for all $z \neq 0$ we have

$$\frac{{}_1F_1(a, b, z)}{\Gamma(b)} = \frac{(-z)^{-a}}{\Gamma(b-a)} U^*(a, b, z) + \frac{z^{a-b} e^z}{\Gamma(a)} U^*(b-a, b, -z)$$

which is equivalent to DLMF 13.2.41 (but simpler in form).

We have the asymptotic expansion

$$U^*(a, b, z) \sim {}_2F_0(a, a-b+1, -1/z)$$

where ${}_2F_0(a, b, z)$ denotes a formal hypergeometric series, i.e.

$$U^*(a, b, z) = \sum_{k=0}^{n-1} \frac{(a)_k (a-b+1)_k}{k! (-z)^k} + \varepsilon_n(z).$$

The error term $\varepsilon_n(z)$ is bounded according to DLMF 13.7. A case distinction is made depending on whether z lies in one of three regions which we index by R . Our formula for the error bound increases with the value of R , so we can always choose the larger out of two indices if z lies in the union of two regions.

Let $r = |b - 2a|$. If $\operatorname{Re}(z) \geq r$, set $R = 1$. Otherwise, if $\operatorname{Im}(z) \geq r$ or $\operatorname{Re}(z) \geq 0 \wedge |z| \geq r$, set $R = 2$. Otherwise, if $|z| \geq 2r$, set $R = 3$. Otherwise, the bound is infinite. If the bound is finite, we have

$$|\varepsilon_n(z)| \leq 2\alpha C_n \left| \frac{(a)_n (a-b+1)_n}{n! z^n} \right| \exp(2\alpha \rho C_1 / |z|)$$

in terms of the following auxiliary quantities

$$\begin{aligned} \sigma &= |(b - 2a)/z| \\ C_n &= \begin{cases} 1 & \text{if } R = 1 \\ \chi(n) & \text{if } R = 2 \\ (\chi(n) + \sigma \nu^2 n) \nu^n & \text{if } R = 3 \end{cases} \\ \nu &= \left(\frac{1}{2} + \frac{1}{2} \sqrt{1 - 4\sigma^2} \right)^{-1/2} \leq 1 + 2\sigma^2 \\ \chi(n) &= \sqrt{\pi} \Gamma(\tfrac{1}{2}n + 1) / \Gamma(\tfrac{1}{2}n + \tfrac{1}{2}) \\ \sigma' &= \begin{cases} \sigma & \text{if } R \neq 3 \\ \nu\sigma & \text{if } R = 3 \end{cases} \\ \alpha &= (1 - \sigma')^{-1} \\ \rho &= \tfrac{1}{2} |2a^2 - 2ab + b| + \sigma' (1 + \tfrac{1}{4}\sigma') (1 - \sigma')^{-2} \end{aligned}$$

9.35.4 Asymptotic series for Airy functions

Error bounds are based on Olver (DLMF section 9.7). For $\arg(z) < \pi$ and $\zeta = (2/3)z^{3/2}$, we have

$$\begin{aligned} \operatorname{Ai}(z) &= \frac{e^{-\zeta}}{2\sqrt{\pi} z^{1/4}} [S_n(\zeta) + R_n(z)], \quad \operatorname{Ai}'(z) = -\frac{z^{1/4} e^{-\zeta}}{2\sqrt{\pi}} [(S'_n(\zeta) + R'_n(z))] \\ S_n(\zeta) &= \sum_{k=0}^{n-1} (-1)^k \frac{u(k)}{\zeta^k}, \quad S'_n(\zeta) = \sum_{k=0}^{n-1} (-1)^k \frac{v(k)}{\zeta^k} \\ u(k) &= \frac{(1/6)_k (5/6)_k}{2^k k!}, \quad v(k) = \frac{6k+1}{1-6k} u(k). \end{aligned}$$

Assuming that n is positive, the error terms are bounded by

$$|R_n(z)| \leq C |u(n)| |\zeta|^{-n}, \quad |R'_n(z)| \leq C |v(n)| |\zeta|^{-n}$$

where

$$C = \begin{cases} 2 \exp(7/(36|\zeta|)) & |\arg(z)| \leq \pi/3 \\ 2\chi(n) \exp(7\pi/(72|\zeta|)) & \pi/3 \leq |\arg(z)| \leq 2\pi/3 \\ 4\chi(n) \exp(7\pi/(36|\operatorname{re}(\zeta)|)) |\cos(\arg(\zeta))|^{-n} & 2\pi/3 \leq |\arg(z)| < \pi. \end{cases}$$

For computing Bi when z is roughly in the positive half-plane, we use the connection formulas

$$\begin{aligned}\text{Bi}(z) &= -i(2w^{+1} \text{Ai}(zw^{-2}) - \text{Ai}(z)) \\ \text{Bi}(z) &= +i(2w^{-1} \text{Ai}(zw^{+2}) - \text{Ai}(z))\end{aligned}$$

where $w = \exp(\pi i/3)$. Combining roots of unity gives

$$\begin{aligned}\text{Bi}(z) &= \frac{1}{2\sqrt{\pi}z^{1/4}}[2X + iY] \\ \text{Bi}(z) &= \frac{1}{2\sqrt{\pi}z^{1/4}}[2X - iY] \\ X &= \exp(+\zeta)[S_n(-\zeta) + R_n(zw^{\mp 2})], \quad Y = \exp(-\zeta)[S_n(\zeta) + R_n(z)]\end{aligned}$$

where the upper formula is valid for $-\pi/3 < \arg(z) < \pi$ and the lower formula is valid for $-\pi < \arg(z) < \pi/3$. We proceed analogously for the derivative of Bi.

In the negative half-plane, we use the connection formulas

$$\begin{aligned}\text{Ai}(z) &= e^{+\pi i/3} \text{Ai}(z_1) + e^{-\pi i/3} \text{Ai}(z_2) \\ \text{Bi}(z) &= e^{-\pi i/6} \text{Ai}(z_1) + e^{+\pi i/6} \text{Ai}(z_2)\end{aligned}$$

where $z_1 = -ze^{+\pi i/3}$, $z_2 = -ze^{-\pi i/3}$. Provided that $|\arg(-z)| < 2\pi/3$, we have $|\arg(z_1)|, |\arg(z_2)| < \pi$, and thus the asymptotic expansion for Ai can be used. As before, we collect roots of unity to obtain

$$\begin{aligned}\text{Ai}(z) &= A_1[S_n(i\zeta) + R_n(z_1)] + A_2[S_n(-i\zeta) + R_n(z_2)] \\ \text{Bi}(z) &= A_3[S_n(i\zeta) + R_n(z_1)] + A_4[S_n(-i\zeta) + R_n(z_2)]\end{aligned}$$

where $\zeta = (2/3)(-z)^{3/2}$ and

$$A_1 = \frac{\exp(-i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_2 = \frac{\exp(+i(\zeta - \pi/4))}{2\sqrt{\pi}(-z)^{1/4}}, \quad A_3 = -iA_1, \quad A_4 = +iA_2.$$

The differentiated formulas are analogous.

9.35.5 Corner case of the Gauss hypergeometric function

In the corner case where z is near $\exp(\pm\pi i/3)$, none of the linear fractional transformations is effective. In this case, we use Taylor series to analytically continue the solution of the hypergeometric differential equation from the origin. The function $f(z) = {}_2F_1(a, b, c, z_0 + z)$ satisfies

$$f''(z) = -\frac{((z_0 + z)(a + b + 1) - c)}{(z_0 + z)(z_0 - 1 + z)}f'(z) - \frac{ab}{(z_0 + z)(z_0 - 1 + z)}f(z).$$

Knowing $f(0), f'(0)$, we can compute the consecutive derivatives recursively, and evaluating the truncated Taylor series allows us to compute $f(z), f'(z)$ to high accuracy for sufficiently small z . Some experimentation showed that two continuation steps

$$0 \rightarrow 0.375 \pm 0.625i \rightarrow 0.5 \pm 0.8125i \rightarrow z$$

gives good performance. Error bounds for the truncated Taylor series are obtained using the Cauchy-Kovalevskaya majorant method, following the outline in [Hoe2001]. The differential equation is majorized by

$$g''(z) = \frac{N+1}{2} \left(\frac{\nu}{1-\nu z} \right) g'(z) + \frac{(N+1)N}{2} \left(\frac{\nu}{1-\nu z} \right)^2 g(z)$$

provided that N and $\nu \geq \max(1/|z_0|, 1/|z_0 - 1|)$ are chosen sufficiently large. It follows that we can compute explicit numbers A, N, ν such that the simple solution $g(z) = A(1 - \nu z)^{-N}$ of the differential equation provides the bound

$$|f_{[k]}| \leq g_{[k]} = A \binom{N+k}{k} \nu^k.$$

9.36 Algorithms for the arithmetic-geometric mean

With complex variables, it is convenient to work with the univariate function $M(z) = \text{agm}(1, z)$. The general case is given by $\text{agm}(a, b) = aM(1, b/a)$.

9.36.1 Functional equation

If the real part of z initially is not completely nonnegative, we apply the functional equation $M(z) = (z+1)M(u)/2$ where $u = \sqrt{z}/(z+1)$.

Note that u has nonnegative real part, absent rounding error. It is not a problem for correctness if rounding makes the interval contain negative points, as this just inflates the final result.

For the derivative, the functional equation becomes $M'(z) = [M(u) - (z-1)M'(u)/((1+z)\sqrt{z})]/2$.

9.36.2 AGM iteration

Once z is in the right half plane, we can apply the AGM iteration ($2a_{n+1} = a_n + b_n, b_{n+1}^2 = a_n b_n$) directly. The correct square root is given by $\sqrt{a}\sqrt{b}$, which is computed as $\sqrt{ab}, i\sqrt{-ab}, -i\sqrt{-ab}, \sqrt{a}\sqrt{b}$ respectively if both a and b have positive real part, nonnegative imaginary part, nonpositive imaginary part, or otherwise.

The iteration should be terminated when a_n and b_n are close enough. For positive real variables, we can simply take lower and upper bounds to get a correct enclosure at this point. For complex variables, it is shown in [Dup2006], p. 87 that, for z with nonnegative real part, $|M(z) - a_n| \leq |a_n - b_n|$, giving a convenient error bound.

Rather than running the AGM iteration until a_n and b_n agree to p bits, it is slightly more efficient to iterate until they agree to about $p/10$ bits and finish with a series expansion. With $z = (a-b)/(a+b)$, we have

$$\text{agm}(a, b) = \frac{(a+b)\pi}{4K(z^2)},$$

valid at least when $|z| < 1$ and a, b have nonnegative real part, and

$$\frac{\pi}{4K(z^2)} = \frac{1}{2} - \frac{1}{8}z^2 - \frac{5}{128}z^4 - \frac{11}{512}z^6 - \frac{469}{32768}z^8 + \dots$$

where the tail is bounded by $\sum_{k=10}^{\infty} |z|^k / 64$.

9.36.3 First derivative

Assuming that z is exact and that $|\arg(z)| \leq 3\pi/4$, we compute $(M(z), M'(z))$ simultaneously using a finite difference.

The basic inequality we need is $|M(z)| \leq \max(1, |z|)$, which is an immediate consequence of the AGM iteration.

By Cauchy's integral formula, $|M^{(k)}(z)/k!| \leq CD^k$ where $C = \max(1, |z| + r)$ and $D = 1/r$, for any $0 < r < |z|$ (we choose r to be of the order $|z|/4$). Taylor expansion now gives

$$\begin{aligned} \left| \frac{M(z+h) - M(z)}{h} - M'(z) \right| &\leq \frac{CD^2h}{1-Dh} \\ \left| \frac{M(z+h) - M(z-h)}{2h} - M'(z) \right| &\leq \frac{CD^3h^2}{1-Dh} \\ \left| \frac{M(z+h) + M(z-h)}{2} - M(z) \right| &\leq \frac{CD^2h^2}{1-Dh} \end{aligned}$$

assuming that h is chosen so that it satisfies $hD < 1$.

The forward finite difference would require two function evaluations at doubled precision. We use the central difference as it only requires 1.5 times the precision.

When z is not exact, we evaluate at the midpoint as above and bound the propagated error using derivatives. Again by Cauchy's integral formula, we have

$$\begin{aligned} |M'(z + \varepsilon)| &\leq \frac{\max(1, |z| + |\varepsilon| + r)}{r} \\ |M''(z + \varepsilon)| &\leq \frac{2 \max(1, |z| + |\varepsilon| + r)}{r^2} \end{aligned}$$

assuming that the circle centered on z with radius $|\varepsilon| + r$ does not cross the negative half axis. We choose r of order $|z|/2$ and verify that all assumptions hold.

9.36.4 Higher derivatives

The function $W(z) = 1/M(z)$ is D-finite. The coefficients of $W(z + x) = \sum_{k=0}^{\infty} c_k x^k$ satisfy

$$-2z(z^2 - 1)c_2 = (3z^2 - 1)c_1 + zc_0,$$

$$-(k+2)(k+3)z(z^2 - 1)c_{k+3} = (k+2)^2(3z^2 - 1)c_{k+2} + (3k(k+3) + 7)zc_{k+1} + (k+1)^2c_k$$

in general, and

$$-(k+2)^2c_{k+2} = (3k(k+3) + 7)c_{k+1} + (k+1)^2c_k$$

when $z = 1$.

EXACT REAL AND COMPLEX NUMBERS

10.1 Introduction

10.1.1 Exact numbers in Calcium

The core idea behind Calcium is to represent real and complex numbers as elements of extension fields

$$\mathbb{Q}(a_1, \dots, a_n)$$

of the rational numbers, where the extension numbers a_k are described by symbolic expressions (which may depend on other fields recursively). The system constructs such fields automatically as needed to represent the results of computations. Any extension field is isomorphic to a formal field

$$\mathbb{Q}(a_1, \dots, a_n) \cong K_{\text{formal}} := \text{Frac}(\mathbb{Q}[X_1, \dots, X_n]/I)$$

where I is the ideal of algebraic relations among the extension numbers. The relations may involve algebraic numbers (for example: $i^2 + 1 = 0$), transcendental numbers (for example: $e^{-\pi} \cdot e^{\pi} = 1$), or combinations thereof.

Computation in the formal field depends (in general) on multivariate polynomial arithmetic together with use of a Gröbner basis for the ideal. The map from the formal field to the true complex field is maintained using arbitrary-precision ball arithmetic where necessary.

As an important special case, Calcium can be used for arithmetic in algebraic number fields (embedded explicitly in $\mathbb{C}$)

$$\mathbb{Q}(a) \cong \mathbb{Q}[X]/\langle f(X) \rangle$$

with excellent performance thanks to internal use of the Antic library.

It will not always work perfectly: although Calcium by design should never give a mathematically erroneous answer, it may be unable to simplify a result as much as expected and it may be unable to decide a predicate (in which case it can return “Unknown”). Equality is at least decidable over the algebraic numbers $\overline{\mathbb{Q}}$ (for practical degrees and bit sizes of the numbers!), and in certain cases involving transcendentals. We hope to improve Calcium’s capabilities gradually through enhancements to its built-in algorithms and through customization options.

Usage details

To understand how Calcium works more concretely, see *Calcium example programs* and the documentation for the main Calcium number type (`ca_t`):

- *ca.h* – exact real and complex numbers

Implementation details for extension numbers and formal fields can be found in the documentation of the corresponding modules:

- *ca_ext.h* – real and complex extension numbers

- *ca_field.h – extension fields*

The following modules are used internally for arithmetic in transcendental number fields (rational function fields) $\mathbb{Q}(x_1, \dots, x_n)$ and over the field of algebraic numbers $\overline{\mathbb{Q}}$, respectively. They may be of independent interest:

- *fmpz_mpoly_q.h – multivariate rational functions over $\mathbb{Q}$*
- *qqbar.h – algebraic numbers represented by minimal polynomials*

10.1.2 FAQ

Isn't $x = 0$ undecidable?

In general, yes: equality over the reals is undecidable. In practice, much of calculus and elementary number theory can be done with numbers that are simple algebraic combinations of well-known elementary and special functions, and there are heuristics that work quite well for deciding predicates about such numbers. Calcium will be able to give a definitive answer at least in simple cases (for example, proving $16 \operatorname{atan}(\frac{1}{5}) - 4 \operatorname{atan}(\frac{1}{239}) = \pi$ or $\sqrt{5 + 2\sqrt{6}} = \sqrt{2} + \sqrt{3}$), and will simply answer “Unknown” when its heuristics are not powerful enough.

How does Calcium compare to ordinary numerical computing?

Calcium is far too slow to replace floating-point numbers for 99.93% of scientific computing. The target is symbolic and algebraic computation. Nevertheless, Calcium may well be useful as a tool to test and enhance the capabilities of numerical programs.

How does Calcium compare to Arb arithmetic?

The main advantage of Calcium over ball arithmetic alone is the ability to do exact comparisons. The automatic precision management in Calcium can also be convenient.

Calcium will usually be slower than Arb arithmetic. If a computation is mostly numerical, it is probably better to try using Arb first, and fall back on an exact calculation with Calcium only if that fails because an exact comparison is needed.

How does Calcium compare to symbolic computation systems (Mathematica, SymPy, etc.)?

Calculating with constant values is only a small part of what such systems have to do, but it is one of the most complex parts. Existing computer algebra systems sometimes manage this very well, and sometimes fail horribly. The most common problems are 1) getting numerical error bounds or branch cuts wrong, and 2) slowing down too much when the expressions get large. Calcium is intended to address both problems (through rigorous numerical evaluation and use of fast polynomial arithmetic).

Ultimately, Calcium will no doubt handle some problems better and others worse, and it should be considered a complement to existing computer algebra systems rather than a replacement. A symbolic expression simplifier may use Calcium evaluation as one of its tools, but this probably needs to be done selectively and in combination with many other heuristics.

Why is Calcium written in C?

The main advantage of developing Calcium as a C library is that it will not be tied to a particular programming language ecosystem: C is uniquely easy to interface from almost any other language. The second most important reason is familiarity: Calcium follows the design of FLINT and Arb (coding style, naming, module layout, memory management, test code, etc.) which has proved to work quite well for libraries of this type.

There is also the performance argument. Some core functions will benefit from optimizations that are natural in C such as in-place operations and fine-grained manual memory management. However, the performance aspect should not be overemphasized: Calcium will spend most of its time in FLINT and Arb kernel functions and this would probably still be true even if it were written in a slower language.

There are certainly types of mathematical functionality that will be too inconvenient to implement in C. Our intention is indeed to leave such functionality to projects written in Python, Julia, etc. which may then opt to depend on Calcium for basic operations.

What is the development status of Calcium?

Calcium is presently in early development and should be considered experimental software. The interfaces are subject to change and many important functions and optimizations have not been implemented. A more stable and functional release can be expected in late 2021.

10.2 Calcium example programs

See *Examples* for general information about example programs. Running:

```
make examples
```

will compile the programs and place the binaries in `build/examples`. The examples related to the Calcium module are documented below.

10.2.1 elementary.c

This program evaluates several elementary expressions. For some inputs, Calcium's arithmetic should produce a simplified result automatically. Some inputs do not yet automatically simplify as much as one might hope. Calcium may still be able to prove that such a number is zero or nonzero; the output of `ca_check_is_zero()` is then `T_TRUE` or `T_FALSE`.

Sample output:

```
> build/examples/elementary
>>> Exp(Pi*I) + 1
0

>>> Log(-1) / (Pi*I)
1

>>> Log(-I) / (Pi*I)
-0.500000 {-1/2}

>>> Log(1 / 10^123) / Log(100)
-61.5000 {-123/2}

>>> Log(1 + Sqrt(2)) / Log(3 + 2*Sqrt(2))
0.500000 {1/2}

>>> Sqrt(2)*Sqrt(3) - Sqrt(6)
0

>>> Exp(1+Sqrt(2)) * Exp(1-Sqrt(2)) / (Exp(1)^2)
1

>>> I^I - Exp(-Pi/2)
0

>>> Exp(Sqrt(3))^2 - Exp(Sqrt(12))
0

>>> 2*Log(Pi*I) - 4*Log(Sqrt(Pi)) - Pi*I
0

>>> -I*Pi/8*Log(2/3-2*I/3)^2 + I*Pi/8*Log(2/3+2*I/3)^2 + Pi^2/12*Log(-1-I) + Pi^2/
↪ 12*Log(-1+I) + Pi^2/12*Log(1/3-I/3) + Pi^2/12*Log(1/3+I/3) - Pi^2/48*Log(18)
0

>>> Sqrt(5 + 2*Sqrt(6)) - Sqrt(2) - Sqrt(3)
0e-1126 {a-c-d where a = 3.14626 [Sqrt(9.89898 {2*b+5})], b = 2.44949 [b^2-6=0], c =
↪ 1.73205 [c^2-3=0], d = 1.41421 [d^2-2=0]}

>>> Is zero?
```

(continues on next page)

(continued from previous page)

```

T_TRUE

>>> Sqrt(I) - (1+I)/Sqrt(2)
0e-1126 + 0e-1126*I {(2*a-b*c-b)/2 where a = 0.707107 + 0.707107*I [Sqrt(1.00000*I {c}
↪)], b = 1.41421 [b^2-2=0], c = I [c^2+1=0]}
>>> Is zero?
T_TRUE

>>> Exp(Pi*Sqrt(163)) - (640320^3 + 744)
-7.49927e-13 {a-262537412640768744 where a = 2.62537e+17 [Exp(40.1092 {b*c})], b = 3.
↪14159 [Pi], c = 12.7671 [c^2-163=0]}

>>> Erf(2*Log(Sqrt(1/2-Sqrt(2)/4))+Log(4)) - Erf(Log(2-Sqrt(2)))
0

cpu/wall(s): 0.022 0.022
virt/peak/res/peak(MB): 36.45 36.47 9.37 9.37

```

10.2.2 binet.c

This program computes the n -th Fibonacci number using Binet's formula $F_n = (\varphi^n - (1 - \varphi)^n)/\sqrt{5}$ where $\varphi = \frac{1}{2}(1 + \sqrt{5})$. The program takes n as input.

Sample output:

```

> build/examples/binet 250
7.89633e+51 {7896325826131730509282738943634332893686268675876375}

cpu/wall(s): 0.002 0.001
virt/peak/res/peak(MB): 36.14 36.14 5.81 5.81

```

This illustrates exact arithmetic in algebraic number fields. The program also illustrates another aspect of Calcium arithmetic: evaluation limits. For example, trying to compute the index $n = 10^6$ Fibonacci number hits an evaluation limit, so the value is not expanded to an explicit integer:

```

> build/examples/binet 1000000
1.95328e+208987 {(a*c-b*c)/5 where a = 4.36767e+208987 [Pow(1.61803 {(c+1)/2}, 1.
↪00000e+6 {1000000})], b = 2.28955e-208988 [Pow(-0.618034 {(c+1)/2}, 1.00000e+6
↪{1000000})], c = 2.23607 [c^2-5=0]}

cpu/wall(s): 0.006 0.005
virt/peak/res/peak(MB): 36.14 36.14 9.05 9.05

```

Calling the program with `-limit B n` raises the bit evaluation limit to B . Setting this large enough allows F_{10^6} to expand to an integer (the following output has been truncated to avoid reproducing all 208988 digits):

```

> build/examples/binet -limit 10000000 1000000
1.95328e+208987 {1953282128...8242546875}

cpu/wall(s): 0.229 0.242
virt/peak/res/peak(MB): 36.79 37.29 7.13 7.13

```

The exact mechanisms and interfaces for evaluation limits are still a work in progress.

10.2.3 machin.c

This program checks several variations of Machin's formula

$$\frac{\pi}{4} = 4 \operatorname{atan}\left(\frac{1}{5}\right) - \operatorname{atan}\left(\frac{1}{239}\right)$$

expressing π or logarithms of small integers in terms of arctangents or hyperbolic arctangents of rational numbers. The program actually evaluates $4 \operatorname{atan}\left(\frac{1}{5}\right) - \operatorname{atan}\left(\frac{1}{239}\right) - \frac{\pi}{4}$ (etc.) and prints the result, which should be precisely 0, proving the identity. Inverse trigonometric functions are not yet implemented in Calcium, so the example program evaluates them using logarithms.

Sample output:

```
> build/examples/machin
[(1)*Atan(1/1) - Pi/4] = 0
[(1)*Atan(1/2) + (1)*Atan(1/3) - Pi/4] = 0
[(2)*Atan(1/2) + (-1)*Atan(1/7) - Pi/4] = 0
[(2)*Atan(1/3) + (1)*Atan(1/7) - Pi/4] = 0
[(4)*Atan(1/5) + (-1)*Atan(1/239) - Pi/4] = 0
[(1)*Atan(1/2) + (1)*Atan(1/5) + (1)*Atan(1/8) - Pi/4] = 0
[(1)*Atan(1/3) + (1)*Atan(1/4) + (1)*Atan(1/7) + (1)*Atan(1/13) - Pi/4] = 0
[(12)*Atan(1/49) + (32)*Atan(1/57) + (-5)*Atan(1/239) + (12)*Atan(1/110443) - Pi/4] ↵
↪ = 0

[(14)*Atanh(1/31) + (10)*Atanh(1/49) + (6)*Atanh(1/161) - Log(2)] = 0
[(22)*Atanh(1/31) + (16)*Atanh(1/49) + (10)*Atanh(1/161) - Log(3)] = 0
[(32)*Atanh(1/31) + (24)*Atanh(1/49) + (14)*Atanh(1/161) - Log(5)] = 0
[(144)*Atanh(1/251) + (54)*Atanh(1/449) + (-38)*Atanh(1/4801) + (62)*Atanh(1/8749) ↵
↪ Log(2)] = 0
[(228)*Atanh(1/251) + (86)*Atanh(1/449) + (-60)*Atanh(1/4801) + (98)*Atanh(1/8749) ↵
↪ Log(3)] = 0
[(334)*Atanh(1/251) + (126)*Atanh(1/449) + (-88)*Atanh(1/4801) + (144)*Atanh(1/8749) ↵
↪ Log(5)] = 0
[(404)*Atanh(1/251) + (152)*Atanh(1/449) + (-106)*Atanh(1/4801) + (174)*Atanh(1/8749) ↵
↪ - Log(7)] = 0

cpu/wall(s): 0.016 0.016
virt/peak/res/peak(MB): 35.57 35.57 8.80 8.80
```

10.2.4 swinnerton_dyer_poly.c

This program computes the coefficients of the Swinnerton-Dyer polynomial

$$S_n = \prod (x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \dots \pm \sqrt{p_n})$$

where p_n denotes the n -th prime number and all combinations of signs are taken. This polynomial has degree 2^n . The polynomial is expanded from its roots using naive polynomial multiplication over `ca_t` coefficients. There are far more efficient ways to construct this polynomial; this program simply illustrates that arithmetic in multivariate number fields works smoothly.

The program prints the coefficients of S_n , from the constant term to the coefficient of x^{2^n} .

Sample output:

```
> build/examples/swinnerton_dyer_poly 3
576
0
-960
```

(continues on next page)

(continued from previous page)

```

0
352
0
-40
0
1

cpu/wall(s): 0.002 0.002
virt/peak/res/peak(MB): 35.07 35.11 5.40 5.40

```

A big benchmark problem (output truncated):

```

> build/examples/swinnerton_dyer_poly 10
4.35675e+809 {43567450015...212890625}
0
...
0
1

cpu/wall(s): 9.296 9.307
virt/peak/res/peak(MB): 38.95 38.95 10.01 10.01

```

10.2.5 huge_expr.c

This program proves equality of two complicated algebraic numbers. More precisely, the program verifies that $N = -(1 - |M|^2)^2$ where N and M are given by huge symbolic expressions involving nested square roots (about 7000 operations in total).

By default, the program runs the computation using `qqbar_t` arithmetic:

```

> build/examples/huge_expr
Evaluating N...
cpu/wall(s): 7.205 7.206
Evaluating M...
cpu/wall(s): 0.933 0.934
Evaluating E = -(1-|M|^2)^2...
cpu/wall(s): 0.391 0.391
N ~ -0.16190853053311203695842869991458578203473645660641
E ~ -0.16190853053311203695842869991458578203473645660641
Testing E = N...
cpu/wall(s): 0.001 0

Equal = T_TRUE

Total: cpu/wall(s): 8.53 8.531
virt/peak/res/peak(MB): 54.50 64.56 24.64 34.61

```

To run the computation using `ca_t` arithmetic instead, pass the `-ca` flag:

```

> build/examples/huge_expr -ca
Evaluating N...
cpu/wall(s): 0.193 0.193
Evaluating M...
cpu/wall(s): 0.024 0.024
Evaluating E = -(1-|M|^2)^2...
cpu/wall(s): 0.008 0.009

```

(continues on next page)

(continued from previous page)

```

N ~ -0.16190853053311203695842869991458578203473645660641
E ~ -0.16190853053311203695842869991458578203473645660641
Testing E = N...
cpu/wall(s): 8.017 8.019

Equal = T_TRUE

Total: cpu/wall(s): 8.243 8.246
virt/peak/res/peak(MB): 61.67 65.29 33.97 37.54

```

This simplification problem was posted in a help request for Sage (<https://ask.sagemath.org/question/52653>). The C code has been generated from the symbolic expressions using a Python script.

10.2.6 hilbert_matrix.c

This program constructs the Hilbert matrix $H_n = (1/(i+j-1))_{i=1,j=1}^n$, computes its eigenvalues $\lambda_1, \dots, \lambda_n$, as exact algebraic numbers, and verifies the exact trace and determinant formulas

$$\lambda_1 + \lambda_2 + \dots + \lambda_n = \text{tr}(H_n), \quad \lambda_1 \lambda_2 \cdots \lambda_n = \det(H_n).$$

Sample output:

```

> build/examples/hilbert_matrix 6
Trace:
1.87821 {6508/3465}
1.87821 {6508/3465}
Equal: T_TRUE

Det:
5.36730e-18 {1/186313420339200000}
5.36730e-18 {1/186313420339200000}
Equal: T_TRUE

cpu/wall(s): 0.07 0.069
virt/peak/res/peak(MB): 36.56 36.66 9.69 9.69

```

The program accepts the following optional arguments:

- With `-vieta`, force use of Vieta's formula internally (by default, Calcium uses Vieta's formulas when working with algebraic conjugates, but only up to some bound on the degree).
- With `-novieta`, force Calcium not to use Vieta's formulas internally.
- With `-qqbar`, do a similar computation using `qqbar_t` arithmetic.

10.2.7 dft.c

This program demonstrates the discrete Fourier transform (DFT) in exact arithmetic. For the input vector $\mathbf{x} = (x_n)_{n=0}^{N-1}$, it verifies the identity

$$\mathbf{x} - \text{DFT}^{-1}(\text{DFT}(\mathbf{x})) = 0$$

where

$$\text{DFT}(\mathbf{x})_n = \sum_{k=0}^{N-1} \omega^{-kn} x_k, \quad \text{DFT}^{-1}(\mathbf{x})_n = \frac{1}{N} \sum_{k=0}^{N-1} \omega^{kn} x_k, \quad \omega = e^{2\pi i/N}.$$

The program computes the DFT by naive $O(N^2)$ summation (not using FFT). It uses repeated multiplication of ω to precompute an array of roots of unity $1, \omega, \omega^2, \dots, \omega^{2N-1}$ for use in both the DFT and the inverse DFT.

Usage:

```
build/examples/dft [-verbose] [-input i] [-limit B] [-timing T] N
```

The required parameter `N` selects the length of the vector.

The optional flag `-verbose` chooses whether to print the arrays.

The optional parameter `-timing T` selects a timing method (default = 0).

- 0: run the computation once and time it
- 1: run the computation repeatedly if needed to get an accurate timing, creating a new context object for each iteration so that fields are not cached
- 2: run the computation once, then run the computation at least one more time (repeatedly if needed to get an accurate timing), recycling the same context object to measure the performance with cached fields

The optional parameter `-input i` selects an input sequence (default = 0).

- 0: $x_n = n + 2$
- 1: $x_n = \sqrt{n + 2}$
- 2: $x_n = \log(n + 2)$
- 3: $x_n = e^{2\pi i/(n+2)}$

The optional parameter `-limit B` sets the internal degree limit for algebraic numbers.

Sample output:

```
> build/examples/dft 4 -input 1 -verbose
DFT benchmark, length N = 4

[x] =
1.41421 {a where a = 1.41421 [a^2-2=0]}
1.73205 {a where a = 1.73205 [a^2-3=0]}
2
2.23607 {a where a = 2.23607 [a^2-5=0]}

DFT([x]) =
7.38233 {a+b+c+2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0]}
-0.585786 + 0.504017*I {a*d-b*d+c-2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
-0.553905 {-a-b+c+2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
-0.585786 - 0.504017*I {-a*d+b*d+c-2 where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}

IDFT(DFT([x])) =
1.41421 {c where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
1.73205 {b where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
2
2.23607 {a where a = 2.23607 [a^2-5=0], b = 1.73205 [b^2-3=0], c = 1.41421 [c^2-2=0], d = I [d^2+1=0]}
```

(continues on next page)

(continued from previous page)

```
[x] - IDFT(DFT([x])) =  
0      (= 0   T_TRUE)  
0      (= 0   T_TRUE)  
0      (= 0   T_TRUE)  
0      (= 0   T_TRUE)  
  
cpu/wall(s): 0.009 0.009  
virt/peak/res/peak(MB): 36.28 36.28 9.14 9.14
```

10.3 calcium.h – global definitions

10.3.1 Version

`const char *calcium_version(void)`

Returns a pointer to the version of the library as a string `X.Y.Z`.

10.3.2 Triple-valued logic

The Calcium modules use two kinds of predicate functions:

- Predicates with signature `int foo_is_X(const foo_t x)` return the usual C boolean values 1 for true and 0 for false, unless otherwise documented. Some functions may return 0 also when truth cannot be certified (this will be documented explicitly).
- Predicates with signature `truth_t foo_check_is_X(const foo_t x)` check a mathematical property that may not be decidable (or may be too costly to decide). The return value is a `truth_t` (`T_TRUE`, `T_FALSE` or `T_UNKNOWN`).

10.3.3 FLINT, Arb and Antic extras

Here we collect various utility methods for FLINT, Arb and Antic types that are missing in those libraries. Some of these functions may be migrated upstream in the future.

`ulong calcium_fmpz_hash(const fmpz_t x)`

Hash function for integers. The algorithm may change; presently, this simply extracts the low word (with sign).

10.3.4 Input and output

`type calcium_stream_struct`

`type calcium_stream_t`

A stream object which can hold either a file pointer or a string (with automatic resizing).

`void calcium_stream_init_file(calcium_stream_t out, FILE *fp)`

Initializes the stream *out* for writing to the file *fp*. The file can be *stdout*, *stderr*, or any file opened for writing by the user.

`void calcium_stream_init_str(calcium_stream_t out)`

Initializes the stream *out* for writing to a string in memory. When finished, the user should free the string (the *s* member of *out* with `flint_free()`).

`void calcium_write(calcium_stream_t out, const char *s)`

Writes the string *s* to *out*.

`void calcium_write_free(calcium_stream_t out, char *s)`

Writes *s* to *out* and then frees *s* by calling `flint_free()`.

`void calcium_write_si(calcium_stream_t out, slong x)`

`void calcium_write_fmpz(calcium_stream_t out, const fmpz_t x)`

Writes the integer *x* to *out*.

`void calcium_write_arb(calcium_stream_t out, const arb_t z, slong digits, ulong flags)`

`void calcium_write_acb(calcium_stream_t out, const acb_t z, slong digits, ulong flags)`

Writes the Arb number *z* to *out*, showing *digits* digits and with the display style specified by *flags* (`ARB_STR_NO_RADIUS`, etc.).

10.4 ca.h – exact real and complex numbers

A `ca_t` represents a real or complex number in a form suitable for exact field arithmetic or comparison. Exceptionally, a `ca_t` may represent a special nonnumerical value, such as an infinity.

10.4.1 Introduction: numbers

A *Calcium number* is a real or complex number represented as an element of a formal field $K = \mathbb{Q}(a_1, \dots, a_n)$ where the symbols a_k denote fixed algebraic or transcendental numbers called *extension numbers*. For example, $e^{-2\pi} - 3i$ may be represented as $(1 - 3a_2^2 a_1)/a_2^2$ in the field $\mathbb{Q}(a_1, a_2)$ with $a_1 = i, a_2 = e^\pi$. Extension numbers and fields are documented in the following separate modules:

- `ca_ext.h` – real and complex extension numbers
- `ca_field.h` – extension fields

The user does not need to construct extension numbers or formal extension fields explicitly: each `ca_t` contains an internal pointer to its formal field, and operations on Calcium numbers generate and cache fields automatically as needed to express the results.

This representation is not canonical (in general). A given complex number can be represented in different ways depending on the choice of formal field K . Even within a fixed field K , a number can have different representations if there are algebraic relations between the extension numbers. Two numbers x and y can be tested for inequality using numerical evaluation; to test for equality, it may be necessary to eliminate dependencies between extension numbers. One of the central goals of Calcium will be to implement heuristics for such elimination.

Together with each formal field K , Calcium stores a *reduction ideal* $I = \{g_1, \dots, g_m\}$ with $g_i \in \mathbb{Z}[a_1, \dots, a_n]$, defining a set of algebraic relations $g_i(a_1, \dots, a_n) = 0$. Relations can be absolute, say $g_i = a_j^2 + 1$, or relative, say $g_i = 2a_j - 4a_k - a_i a_m$. The reduction ideal effectively partitions K into equivalence classes of complex numbers (e.g. $i^2 = -1$ or $2 \log(\pi i) = 4 \log(\sqrt{\pi}) + \pi i$), enabling simplifications and equality proving.

Extension numbers are always sorted $a_1 \succ a_2 \succ \dots \succ a_n$ where $\succ$ denotes a structural ordering (see `ca_cmp_repr()`). If the reduction ideal is triangular and the multivariate polynomial arithmetic uses lexicographic ordering, reduction by I eliminates numbers a_i with higher complexity in the sense of $\succ$.

The reduction ideal is an imperfect computational crutch: it is not guaranteed to capture *all* algebraic relations, and reduction is not guaranteed to produce uniquely defined representatives. However, in the specific case of an absolute number field $K = \mathbb{Q}(a)$ where a is a `qqbar_t` extension, the reduction ideal (consisting of a single minimal polynomial) is canonical and field elements of K can be chosen canonically.

10.4.2 Introduction: special values

In order to provide a closed arithmetic system and express limiting cases of operators and special functions, a `ca_t` can hold any of the following special values besides ordinary numbers:

- *Unsigned infinity*, a formal object $\tilde{\infty}$ representing the value of $1/0$. More generally, this is the value of meromorphic functions at poles.
- *Signed infinity*, a formal object $a \cdot \infty$ where the sign a is a Calcium number with $|a| = 1$. The most common values are $+\infty, -\infty, +i\infty, -i\infty$. Signed infinities are used to denote directional limits and logarithmic singularities (for example, $\log(0) = -\infty$).
- *Undefined*, a formal object representing the value of indeterminate forms such as $0/0$ and essential singularities such as $\exp(\tilde{\infty})$, where a number or infinity would not make sense as an answer.
- *Unknown*, a meta-value used to signal that the actual desired value could not be computed, either because Calcium does not (yet) have a data structure or algorithm for that case, or because doing so would be unreasonably expensive. This occurs, for example, if Calcium performs a division and is unable to decide whether the result is a number, unsigned infinity or undefined (because testing

for zero fails). Wrappers may want to check output variables for *Unknown* and throw an exception (e.g. *NotImplementedError* in Python).

The distinction between *Calcium numbers* (which must represent elements of $\mathbb{C}$) and the different kinds of nonnumerical values (infinities, Undefined or Unknown) is essential. Nonnumerical values may not be used as field extension numbers a_k , and the denominator of a formal field element must always represent a nonzero complex number. Accordingly, for any given Calcium value x that is not *Unknown*, it is exactly known whether x represents A) a number, B) unsigned infinity, C) a signed infinity, or D) Undefined.

10.4.3 Number objects

For all types, a *type_t* is defined as an array of length one of type *type_struct*, permitting a *type_t* to be passed by reference.

type **ca_struct**

type **ca_t**

A *ca_t* contains an index to a field K , and data representing an element x of K . The data is either an inline rational number (*fmpq_t*), an inline Antic number field element (*nf_elem_t*) when K is an absolute algebraic number field $\mathbb{Q}(a)$, or a pointer to a heap-allocated *fmpz_poly_q_t* representing an element of a generic field $\mathbb{Q}(a_1, \dots, a_n)$. Special values are encoded using magic bits in the field index.

type **ca_ptr**

Alias for *ca_struct **, used for vectors of numbers.

type **ca_srcptr**

Alias for *const ca_struct **, used for vectors of numbers when passed as constant input to functions.

10.4.4 Context objects

type **ca_ctx_struct**

type **ca_ctx_t**

A *ca_ctx_t* context object holds a cache of fields K and constituent extension numbers a_k . The field index in an individual *ca_t* instance represents a shallow reference to the object defining the field K within the context object, so creating many elements of the same field is cheap.

Since context objects are mutable (and may be mutated even when performing read-only operations on *ca_t* instances), they must not be accessed simultaneously by different threads: in multithreaded environments, the user must use a separate context object for each thread.

void **ca_ctx_init**(*ca_ctx_t* ctx)

Initializes the context object *ctx* for use. Any evaluation options stored in the context object are set to default values.

void **ca_ctx_clear**(*ca_ctx_t* ctx)

Clears the context object *ctx*, freeing any memory allocated internally. This function should only be called after all *ca_t* instances referring to this context have been cleared.

void **ca_ctx_print**(*ca_ctx_t* ctx)

Prints a description of the context *ctx* to standard output. This will give a complete listing of the cached fields in *ctx*.

10.4.5 Memory management for numbers

void **ca_init**(*ca_t* x, *ca_ctx_t* ctx)

Initializes the variable *x* for use, associating it with the context object *ctx*. The value of *x* is set to the rational number 0.

void **ca_clear**(*ca_t* x, *ca_ctx_t* ctx)

Clears the variable *x*.

void **ca_swap**(*ca_t* x, *ca_t* y, *ca_ctx_t* ctx)

Efficiently swaps the variables *x* and *y*.

10.4.6 Symbolic expressions

void **ca_get_fexpr**(*fexpr_t* res, const *ca_t* x, *ulong* flags, *ca_ctx_t* ctx)

Sets *res* to a symbolic expression representing *x*.

int **ca_set_fexpr**(*ca_t* res, const *fexpr_t* expr, *ca_ctx_t* ctx)

Sets *res* to the value represented by the symbolic expression *expr*. Returns 1 on success and 0 on failure. This function essentially just traverses the expression tree using **ca** arithmetic; it does not provide advanced symbolic evaluation. It is guaranteed to at least be able to parse the output of **ca_get_fexpr()**.

10.4.7 Printing

The style of printed output is controlled by `ctx->options[CA_OPT_PRINT_FLAGS]` (see *Context options*) which can be set to any combination of the following flags:

CA_PRINT_N

Print a decimal approximation of the number. The approximation is guaranteed to be correctly rounded to within one unit in the last place.

If combined with **CA_PRINT_REPR**, numbers appearing within the symbolic representation will also be printed with decimal approximations.

Warning: printing a decimal approximation requires a computation, which can be expensive. It can also mutate cached data (numerical enclosures of extension numbers), affecting subsequent computations.

CA_PRINT_DIGITS

Multiplied by a positive integer, specifies the number of decimal digits to show with **CA_PRINT_N**. If not given, the default precision is six digits.

CA_PRINT_REPR

Print the symbolic representation of the number (including its recursive elements). If used together with **CA_PRINT_N**, field elements will print as `decimal {symbolic}` while extension numbers will print as `decimal [symbolic]`.

All extension numbers appearing in the field defining *x* and in the inner constructions of those extension numbers will be given local labels *a*, *b*, etc. for this printing.

CA_PRINT_FIELD

For each field element, explicitly print its formal field along with its reduction ideal if present, e.g. `QQ` or `QQ(a,b,c) / <a-b, c^2+1>`.

CA_PRINT_DEFAULT

The default print style. Equivalent to `CA_PRINT_N | CA_PRINT_REPR`.

10.4.8 Special values

void **ca_zero**(*ca_t* res, *ca_ctx_t* ctx)

void **ca_one**(*ca_t* res, *ca_ctx_t* ctx)

void **ca_neg_one**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to the integer 0, 1 or -1. This creates a canonical representation of this number as an element of the trivial field $\mathbb{Q}$.

void **ca_i**(*ca_t* res, *ca_ctx_t* ctx)

void **ca_neg_i**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to the imaginary unit $i = \sqrt{-1}$, or its negation $-i$. This creates a canonical representation of i as the generator of the algebraic number field $\mathbb{Q}(i)$.

void **ca_pi**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to the constant π . This creates an element of the transcendental number field $\mathbb{Q}(\pi)$.

void **ca_pi_i**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to the constant πi . This creates an element of the composite field $\mathbb{Q}(i, \pi)$ rather than representing πi (or even $2\pi i$, which for some purposes would be more elegant) as an atomic quantity.

void **ca_euler**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to Euler's constant γ . This creates an element of the (transcendental?) number field $\mathbb{Q}(\gamma)$.

void **ca_unknown**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to the meta-value *Unknown*.

void **ca_undefined**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to *Undefined*.

void **ca_uinf**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to unsigned infinity ∞ .

void **ca_pos_inf**(*ca_t* res, *ca_ctx_t* ctx)

void **ca_neg_inf**(*ca_t* res, *ca_ctx_t* ctx)

void **ca_pos_i_inf**(*ca_t* res, *ca_ctx_t* ctx)

void **ca_neg_i_inf**(*ca_t* res, *ca_ctx_t* ctx)

Sets *res* to the signed infinity $+\infty$, $-\infty$, $+i\infty$ or $-i\infty$.

10.4.9 Assignment and conversion

void **ca_set**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to a copy of *x*.

void **ca_set_si**(*ca_t* res, *slong* v, *ca_ctx_t* ctx)

void **ca_set_ui**(*ca_t* res, *ulong* v, *ca_ctx_t* ctx)

void **ca_set_fmpz**(*ca_t* res, const *fmpz_t* v, *ca_ctx_t* ctx)

void **ca_set_fmpq**(*ca_t* res, const *fmpq_t* v, *ca_ctx_t* ctx)

Sets *res* to the integer or rational number *v*. This creates a canonical representation of this number as an element of the trivial field $\mathbb{Q}$.

void **ca_set_d**(*ca_t* res, double x, *ca_ctx_t* ctx)

void **ca_set_d_d**(*ca_t* res, double x, double y, *ca_ctx_t* ctx)

Sets *res* to the value of *x*, or the complex value $x + yi$. NaN is interpreted as *Unknown* (not *Undefined*).

void **ca_transfer**(*ca_t* res, *ca_ctx_t* res_ctx, const *ca_t* src, *ca_ctx_t* src_ctx)

Sets *res* to *src* where the corresponding context objects *res_ctx* and *src_ctx* may be different.

This operation preserves the mathematical value represented by *src*, but may result in a different internal representation depending on the settings of the context objects.

10.4.10 Conversion of algebraic numbers

void **ca_set_qqbar**(*ca_t* res, const *qqbar_t* x, *ca_ctx_t* ctx)

Sets *res* to the algebraic number *x*.

If *x* is rational, *res* is set to the canonical representation as an element in the trivial field $\mathbb{Q}$.

If *x* is irrational, this function always sets *res* to an element of a univariate number field $\mathbb{Q}(a)$. It will not, for example, identify $\sqrt{2} + \sqrt{3}$ as an element of $\mathbb{Q}(\sqrt{2}, \sqrt{3})$. However, it may attempt to find a simpler number field than that generated by *x* itself. For example:

- If *x* is quadratic, it will be expressed as an element of $\mathbb{Q}(\sqrt{N})$ where *N* has no small repeated factors (obtained by performing a smooth factorization of the discriminant).
- TODO: if possible, coerce *x* to a low-degree cyclotomic field.

int **ca_get_fmpz**(*fmpz_t* res, const *ca_t* x, *ca_ctx_t* ctx)

int **ca_get_fmpq**(*fmpq_t* res, const *ca_t* x, *ca_ctx_t* ctx)

int **ca_get_qqbar**(*qqbar_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Attempts to evaluate *x* to an explicit integer, rational or algebraic number. If successful, sets *res* to this number and returns 1. If unsuccessful, returns 0.

The conversion certainly fails if *x* does not represent an integer, rational or algebraic number (respectively), but can also fail if *x* is too expensive to compute under the current evaluation limits. In particular, the evaluation will be aborted if an intermediate algebraic number (or more precisely, the resultant polynomial prior to factorization) exceeds **CA_OPT_QQBAR_DEG_LIMIT** or the coefficients exceed some multiple of **CA_OPT_PREC_LIMIT**. Note that evaluation may hit those limits even if the minimal polynomial for *x* itself is small. The conversion can also fail if no algorithm has been implemented for the functions appearing in the construction of *x*.

int **ca_can_evaluate_qqbar**(const *ca_t* x, *ca_ctx_t* ctx)

Checks if **ca_get_qqbar()** has a chance to succeed. In effect, this checks if all extension numbers are manifestly algebraic numbers (without doing any evaluation).

10.4.11 Random generation

void **ca_randtest_rational**(*ca_t* res, *flint_rand_t* state, *slong* bits, *ca_ctx_t* ctx)

Sets *res* to a random rational number with numerator and denominator up to *bits* bits in size.

void **ca_randtest**(*ca_t* res, *flint_rand_t* state, *slong* depth, *slong* bits, *ca_ctx_t* ctx)

Sets *res* to a random number generated by evaluating a random expression. The algorithm randomly selects between generating a “simple” number (a random rational number or quadratic field element with coefficients up to *bits* in size, or a random builtin constant), or if *depth* is nonzero, applying a random arithmetic operation or function to operands produced through recursive calls with *depth* - 1. The output is guaranteed to be a number, not a special value.

void **ca_randtest_special**(*ca_t* res, *flint_rand_t* state, *slong* depth, *slong* bits, *ca_ctx_t* ctx)

Randomly generates either a special value or a number.

void **ca_randtest_same_nf**(*ca_t* res, *flint_rand_t* state, const *ca_t* x, *slong* bits, *slong* den_bits, *ca_ctx_t* ctx)

Sets *res* to a random element in the same number field as *x*, with numerator coefficients up to *bits* in size and denominator up to *den_bits* in size. This function requires that *x* is an element of an absolute number field.

10.4.12 Representation properties

The following functions deal with the representation of a `ca_t` and hence can always be decided quickly and unambiguously. The return value for predicates is 0 for false and 1 for true.

int `ca_equal_repr`(const `ca_t` x, const `ca_t` y, `ca_ctx_t` ctx)

Returns whether x and y have identical representation. For field elements, this checks if x and y belong to the same formal field (with generators having identical representation) and are represented by the same rational function within that field.

For special values, this tests equality of the special values, with *Unknown* handled as if it were a value rather than a meta-value: that is, $Unknown = Unknown$ gives 1, and $Unknown = y$ gives 0 for any other kind of value y . If neither x nor y is *Unknown*, then representation equality implies that x and y describe the same mathematical value, but if either operand is *Unknown*, the result is meaningless for mathematical comparison.

int `ca_cmp_repr`(const `ca_t` x, const `ca_t` y, `ca_ctx_t` ctx)

Compares the representations of x and y in a canonical sort order, returning -1, 0 or 1. This only performs a lexicographic comparison of the representations of x and y ; the return value does not say anything meaningful about the numbers represented by x and y .

ulong `ca_hash_repr`(const `ca_t` x, `ca_ctx_t` ctx)

Hashes the representation of x .

int `ca_is_unknown`(const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is *Unknown*.

int `ca_is_special`(const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is a special value or metavalue (not a field element).

int `ca_is_qq_elem`(const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is represented as an element of the rational field $\mathbb{Q}$.

int `ca_is_qq_elem_zero`(const `ca_t` x, `ca_ctx_t` ctx)

int `ca_is_qq_elem_one`(const `ca_t` x, `ca_ctx_t` ctx)

int `ca_is_qq_elem_integer`(const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is represented as the element 0, 1 or any integer in the rational field $\mathbb{Q}$.

int `ca_is_nf_elem`(const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is represented as an element of a univariate algebraic number field $\mathbb{Q}(a)$.

int `ca_is_cyclotomic_nf_elem`(*slong* *p, *ulong* *q, const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is represented as an element of a univariate cyclotomic field, i.e. $\mathbb{Q}(a)$ where a is a root of unity. If p and q are not *NULL* and x is represented as an element of a cyclotomic field, this also sets p and q to the minimal integers with $0 \leq p < q$ such that the generating root of unity is $a = e^{2\pi i p/q}$. Note that the answer 0 does not prove that x is not a cyclotomic number, and the order q is also not necessarily the generator of the *smallest* cyclotomic field containing x . For the purposes of this function, only nontrivial cyclotomic fields count; the return value is 0 if x is represented as a rational number.

int `ca_is_generic_elem`(const `ca_t` x, `ca_ctx_t` ctx)

Returns whether x is represented as a generic field element; i.e. it is not a special value, not represented as an element of the rational field, and not represented as an element of a univariate algebraic number field.

10.4.13 Value predicates

The following predicates check a mathematical property which might not be effectively decidable. The result is a *truth_t* to allow representing an unknown outcome.

truth_t **ca_check_is_number**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is a number. The result is **T_TRUE** if x is a field element (and hence a complex number), **T_FALSE** if x is an infinity or *Undefined*, and **T_UNKNOWN** if x is *Unknown*.

truth_t **ca_check_is_zero**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_one**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_neg_one**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_i**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_neg_i**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is equal to the number 0, 1, -1 , i , or $-i$.

truth_t **ca_check_is_algebraic**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_rational**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_integer**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is respectively an algebraic number, a rational number, or an integer.

truth_t **ca_check_is_real**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is a real number. Warning: this returns **T_FALSE** if x is an infinity with real sign.

truth_t **ca_check_is_negative_real**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is a negative real number. Warning: this returns **T_FALSE** if x is negative infinity.

truth_t **ca_check_is_imaginary**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is an imaginary number. Warning: this returns **T_FALSE** if x is an infinity with imaginary sign.

truth_t **ca_check_is_undefined**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is the special value *Undefined*.

truth_t **ca_check_is_infinity**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is any infinity (unsigned or signed).

truth_t **ca_check_is_uinf**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is unsigned infinity ∞ .

truth_t **ca_check_is_signed_inf**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is any signed infinity.

truth_t **ca_check_is_pos_inf**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_neg_inf**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_pos_i_inf**(const *ca_t* x, *ca_ctx_t* ctx)

truth_t **ca_check_is_neg_i_inf**(const *ca_t* x, *ca_ctx_t* ctx)

Tests if x is equal to the signed infinity $+\infty$, $-\infty$, $+i\infty$, $-i\infty$, respectively.

10.4.14 Comparisons

`truth_t ca_check_equal(const ca_t x, const ca_t y, ca_ctx_t ctx)`

Tests $x = y$ as a mathematical equality. The result is `T_UNKNOWN` if either operand is *Unknown*. The result may also be `T_UNKNOWN` if x and y are numerically indistinguishable and cannot be proved equal or unequal by an exact computation.

`truth_t ca_check_lt(const ca_t x, const ca_t y, ca_ctx_t ctx)`

`truth_t ca_check_le(const ca_t x, const ca_t y, ca_ctx_t ctx)`

`truth_t ca_check_gt(const ca_t x, const ca_t y, ca_ctx_t ctx)`

`truth_t ca_check_ge(const ca_t x, const ca_t y, ca_ctx_t ctx)`

Compares x and y , implementing the respective operations $x < y$, $x \leq y$, $x > y$, $x \geq y$. Only real numbers and $-\infty$ and $+\infty$ are considered comparable. The result is `T_FALSE` (not `T_UNKNOWN`) if either operand is not comparable (being a nonreal complex number, unsigned infinity, or undefined).

10.4.15 Field structure operations

`void ca_merge_fields(ca_t resx, ca_t resy, const ca_t x, const ca_t y, ca_ctx_t ctx)`

Sets *resx* and *resy* to copies of x and y coerced to a common field. Both x and y must be field elements (not special values).

In the present implementation, this simply merges the lists of generators, avoiding duplication. In the future, it will be able to eliminate generators satisfying algebraic relations.

`void ca_condense_field(ca_t res, ca_ctx_t ctx)`

Attempts to demote the value of *res* to a trivial subfield of its current field by removing unused generators. In particular, this demotes any obviously rational value to the trivial field $\mathbb{Q}$.

This function is applied automatically in most operations (arithmetic operations, etc.).

`ca_ext_ptr ca_is_gen_as_ext(const ca_t x, ca_ctx_t ctx)`

If x is a generator of its formal field, $x = a_k \in \mathbb{Q}(a_1, \dots, a_n)$, returns a pointer to the extension number defining a_k . If x is not a generator, returns `NULL`.

10.4.16 Arithmetic

`void ca_neg(ca_t res, const ca_t x, ca_ctx_t ctx)`

Sets *res* to the negation of x . For numbers, this operation amounts to a direct negation within the formal field. For a signed infinity $c\infty$, negation gives $(-c)\infty$; all other special values are unchanged.

`void ca_add_fmpq(ca_t res, const ca_t x, const fmpq_t y, ca_ctx_t ctx)`

`void ca_add_fmpz(ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)`

`void ca_add_ui(ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)`

`void ca_add_si(ca_t res, const ca_t x, slong y, ca_ctx_t ctx)`

`void ca_add(ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)`

Sets *res* to the sum of x and y . For special values, the following rules apply ($c\infty$ denotes a signed infinity, $|c| = 1$):

- $c\infty + d\infty = c\infty$ if $c = d$
- $c\infty + d\infty = \text{Undefined}$ if $c \neq d$
- $\tilde{\infty} + c\infty = \tilde{\infty} + \tilde{\infty} = \text{Undefined}$
- $c\infty + z = c\infty$ if $z \in \mathbb{C}$
- $\tilde{\infty} + z = \tilde{\infty}$ if $z \in \mathbb{C}$
- $z + \text{Undefined} = \text{Undefined}$ for any value z (including *Unknown*)

In any other case involving special values, or if the specific case cannot be distinguished, the result is *Unknown*.

```
void ca_sub_fmpq(ca_t res, const ca_t x, const fmpq_t y, ca_ctx_t ctx)
void ca_sub_fmpz(ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_sub_ui(ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)
void ca_sub_si(ca_t res, const ca_t x, slong y, ca_ctx_t ctx)
void ca_fmpq_sub(ca_t res, const fmpq_t x, const ca_t y, ca_ctx_t ctx)
void ca_fmpz_sub(ca_t res, const fmpz_t x, const ca_t y, ca_ctx_t ctx)
void ca_ui_sub(ca_t res, ulong x, const ca_t y, ca_ctx_t ctx)
void ca_si_sub(ca_t res, slong x, const ca_t y, ca_ctx_t ctx)
void ca_sub(ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)
```

Sets *res* to the difference of *x* and *y*. This is equivalent to computing $x + (-y)$.

```
void ca_mul_fmpq(ca_t res, const ca_t x, const fmpq_t y, ca_ctx_t ctx)
void ca_mul_fmpz(ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_mul_ui(ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)
void ca_mul_si(ca_t res, const ca_t x, slong y, ca_ctx_t ctx)
void ca_mul(ca_t res, const ca_t x, const ca_t y, ca_ctx_t ctx)
```

Sets *res* to the product of *x* and *y*. For special values, the following rules apply ($c\infty$ denotes a signed infinity, $|c| = 1$):

- $c\infty \cdot d\infty = cd\infty$
- $c\infty \cdot \tilde{\infty} = \tilde{\infty}$
- $\tilde{\infty} \cdot \tilde{\infty} = \tilde{\infty}$
- $c\infty \cdot z = \text{sgn}(z)c\infty$ if $z \in \mathbb{C} \setminus \{0\}$
- $c\infty \cdot 0 = \text{Undefined}$
- $\tilde{\infty} \cdot 0 = \text{Undefined}$
- $z \cdot \text{Undefined} = \text{Undefined}$ for any value *z* (including *Unknown*)

In any other case involving special values, or if the specific case cannot be distinguished, the result is *Unknown*.

```
void ca_inv(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the multiplicative inverse of *x*. In a univariate algebraic number field, this always produces a rational denominator, but the denominator might not be rationalized in a multivariate field. For special values and zero, the following rules apply:

- $1/(c\infty) = 1/\tilde{\infty} = 0$
- $1/0 = \tilde{\infty}$
- $1/\text{Undefined} = \text{Undefined}$
- $1/\text{Unknown} = \text{Unknown}$

If it cannot be determined whether *x* is zero or nonzero, the result is *Unknown*.

```
void ca_fmpq_div(ca_t res, const fmpq_t x, const ca_t y, ca_ctx_t ctx)
void ca_fmpz_div(ca_t res, const fmpz_t x, const ca_t y, ca_ctx_t ctx)
void ca_ui_div(ca_t res, ulong x, const ca_t y, ca_ctx_t ctx)
void ca_si_div(ca_t res, slong x, const ca_t y, ca_ctx_t ctx)
void ca_div_fmpq(ca_t res, const ca_t x, const fmpq_t y, ca_ctx_t ctx)
void ca_div_fmpz(ca_t res, const ca_t x, const fmpz_t y, ca_ctx_t ctx)
void ca_div_ui(ca_t res, const ca_t x, ulong y, ca_ctx_t ctx)
void ca_div_si(ca_t res, const ca_t x, slong y, ca_ctx_t ctx)
```

void **ca_div**(*ca_t* res, const *ca_t* x, const *ca_t* y, *ca_ctx_t* ctx)

Sets *res* to the quotient of *x* and *y*. This is equivalent to computing $x \cdot (1/y)$. For special values and division by zero, this implies the following rules ($c\infty$ denotes a signed infinity, $|c| = 1$):

- $(c\infty)/(d\infty) = (c\infty)/\tilde{\infty} = \tilde{\infty}/(c\infty) = \tilde{\infty}/\tilde{\infty} = \text{Undefined}$
- $c\infty/z = (c/\text{sgn}(z))\infty$ if $z \in \mathbb{C} \setminus \{0\}$
- $c\infty/0 = \tilde{\infty}/0 = \tilde{\infty}$
- $z/(c\infty) = z/\tilde{\infty} = 0$ if $z \in \mathbb{C}$
- $z/0 = \tilde{\infty}$ if $z \in \mathbb{C} \setminus \{0\}$
- $0/0 = \text{Undefined}$
- $z/\text{Undefined} = \text{Undefined}$ for any value *z* (including *Unknown*)
- $\text{Undefined}/z = \text{Undefined}$ for any value *z* (including *Unknown*)

In any other case involving special values, or if the specific case cannot be distinguished, the result is *Unknown*.

void **ca_dot**(*ca_t* res, const *ca_t* initial, int subtract, *ca_srcptr* x, *slong* xstep, *ca_srcptr* y, *slong* ystep, *slong* len, *ca_ctx_t* ctx)

Computes the dot product of the vectors *x* and *y*, setting *res* to $s + (-1)^{\text{subtract}} \sum_{i=0}^{\text{len}-1} x_i y_i$.

The initial term *s* is optional and can be omitted by passing *NULL* (equivalently, $s = 0$). The parameter *subtract* must be 0 or 1. The length *len* is allowed to be negative, which is equivalent to a length of zero. The parameters *xstep* or *ystep* specify a step length for traversing subsequences of the vectors *x* and *y*; either can be negative to step in the reverse direction starting from the initial pointer. Aliasing is allowed between *res* and *s* but not between *res* and the entries of *x* and *y*.

void **ca_fmpz_poly_evaluate**(*ca_t* res, const *fmpz_poly_t* poly, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_fmpz_poly_evaluate**(*ca_t* res, const *fmpz_poly_t* poly, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the polynomial *poly* evaluated at *x*.

void **ca_fmpz_mpoly_evaluate_horner**(*ca_t* res, const *fmpz_mpoly_t* f, *ca_srcptr* x, const *fmpz_mpoly_ctx_t* mctx, *ca_ctx_t* ctx)

void **ca_fmpz_mpoly_evaluate_iter**(*ca_t* res, const *fmpz_mpoly_t* f, *ca_srcptr* x, const *fmpz_mpoly_ctx_t* mctx, *ca_ctx_t* ctx)

void **ca_fmpz_mpoly_evaluate**(*ca_t* res, const *fmpz_mpoly_t* f, *ca_srcptr* x, const *fmpz_mpoly_ctx_t* mctx, *ca_ctx_t* ctx)

Sets *res* to the multivariate polynomial *f* evaluated at the vector of arguments *x*.

void **ca_fmpz_mpoly_q_evaluate**(*ca_t* res, const *fmpz_mpoly_q_t* f, *ca_srcptr* x, const *fmpz_mpoly_ctx_t* mctx, *ca_ctx_t* ctx)

Sets *res* to the multivariate rational function *f* evaluated at the vector of arguments *x*.

void **ca_fmpz_mpoly_q_evaluate_no_division_by_zero**(*ca_t* res, const *fmpz_mpoly_q_t* f, *ca_srcptr* x, const *fmpz_mpoly_ctx_t* mctx, *ca_ctx_t* ctx)

void **ca_inv_no_division_by_zero**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

These functions behave like the normal arithmetic functions, but assume (and do not check) that division by zero cannot occur. Division by zero will result in undefined behavior.

10.4.17 Powers and roots

void **ca_sqr**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the square of *x*.

void **ca_pow_fmpq**(*ca_t* res, const *ca_t* x, const *fmpq_t* y, *ca_ctx_t* ctx)

void **ca_pow_fmpz**(*ca_t* res, const *ca_t* x, const *fmpz_t* y, *ca_ctx_t* ctx)

void **ca_pow_ui**(*ca_t* res, const *ca_t* x, *ulong* y, *ca_ctx_t* ctx)

void **ca_pow_si**(*ca_t* res, const *ca_t* x, *slong* y, *ca_ctx_t* ctx)

void **ca_pow**(*ca_t* res, const *ca_t* x, const *ca_t* y, *ca_ctx_t* ctx)

Sets *res* to *x* raised to the power *y*. Handling of special values is not yet implemented.

void **ca_pow_si_arithmetic**(*ca_t* res, const *ca_t* x, *slong* n, *ca_ctx_t* ctx)

Sets *res* to *x* raised to the power *n*. Whereas **ca_pow()**, **ca_pow_si()** etc. may create x^n as an extension number if *n* is large, this function always perform the exponentiation using field arithmetic.

void **ca_sqrt_inert**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_sqrt_nofactor**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_sqrt_factor**(*ca_t* res, const *ca_t* x, *ulong* flags, *ca_ctx_t* ctx)

void **ca_sqrt**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the principal square root of *x*.

For special values, the following definitions apply:

- $\sqrt{c\infty} = \sqrt{c}\infty$
- $\sqrt{\infty} = \infty$.
- Both *Undefined* and *Unknown* map to themselves.

The *inert* version outputs the generator in the formal field $\mathbb{Q}(\sqrt{x})$ without simplifying.

The *factor* version writes $x = A^2B$ in *K* where *K* is the field of *x*, and outputs $A\sqrt{B}$ or $-A\sqrt{B}$ (whichever gives the correct sign) as an element of $K(\sqrt{B})$ or some subfield thereof. This factorization is only a heuristic and is not guaranteed to make *B* minimal. Factorization options can be passed through to *flags*: see **ca_factor()** for details.

The *nofactor* version will not perform a general factorization, but may still perform other simplifications. It may in particular attempt to simplify $\sqrt{x}$ to a single element in $\overline{\mathbb{Q}}$.

void **ca_sqrt_ui**(*ca_t* res, *ulong* n, *ca_ctx_t* ctx)

Sets *res* to the principal square root of *n*.

10.4.18 Complex parts

void **ca_abs**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the absolute value of *x*.

For special values, the following definitions apply:

- $|c\infty| = |\infty| = +\infty$.
- Both *Undefined* and *Unknown* map to themselves.

This function will attempt to simplify its argument through an exact computation. It may in particular attempt to simplify $|x|$ to a single element in $\overline{\mathbb{Q}}$.

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(|x|)$.

void **ca_sgn**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the sign of *x*, defined by

$$\operatorname{sgn}(x) = \begin{cases} 0 & x = 0 \\ \frac{x}{|x|} & x \neq 0 \end{cases}$$

for numbers. For special values, the following definitions apply:

- $\operatorname{sgn}(c\infty) = c$.
- $\operatorname{sgn}(\infty) = \text{Undefined}$.
- Both *Undefined* and *Unknown* map to themselves.

This function will attempt to simplify its argument through an exact computation. It may in particular attempt to simplify $\operatorname{sgn}(x)$ to a single element in $\overline{\mathbb{Q}}$.

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(\operatorname{sgn}(x))$.

void **ca_csgn**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the extension of the real sign function taking the value 1 for *z* strictly in the right half plane, -1 for *z* strictly in the left half plane, and the sign of the imaginary part when *z* is on the imaginary axis. Equivalently, $\operatorname{csgn}(z) = z/\sqrt{z^2}$ except that the value is 0 when *z* is exactly zero. This function gives *Undefined* for unsigned infinity and $\operatorname{csgn}(\operatorname{sgn}(c\infty)) = \operatorname{csgn}(c)$ for signed infinities.

void **ca_arg**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the complex argument (phase) of *x*, normalized to the range $(-\pi, +\pi]$. The argument of 0 is defined as 0. For special values, the following definitions apply:

- $\operatorname{arg}(c\infty) = \operatorname{arg}(c)$.
- $\operatorname{arg}(\infty) = \text{Undefined}$.
- Both *Undefined* and *Unknown* map to themselves.

void **ca_re**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the real part of *x*. The result is *Undefined* if *x* is any infinity (including a real infinity).

void **ca_im**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the imaginary part of *x*. The result is *Undefined* if *x* is any infinity (including an imaginary infinity).

void **ca_conj_deep**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_conj_shallow**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_conj**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the complex conjugate of *x*. The *shallow* version creates a new extension element $\bar{x}$ unless *x* can be trivially conjugated in-place in the existing field. The *deep* version recursively conjugates the extension numbers in the field of *x*.

void **ca_floor**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the floor function of *x*. The result is *Undefined* if *x* is any infinity (including a real infinity). For complex numbers, this is presently defined to take the floor of the real part.

void **ca_ceil**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the ceiling function of *x*. The result is *Undefined* if *x* is any infinity (including a real infinity). For complex numbers, this is presently defined to take the ceiling of the real part.

10.4.19 Exponentials and logarithms

void **ca_exp**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the exponential function of *x*.

For special values, the following definitions apply:

- $e^{+\infty} = +\infty$
- $e^{c\infty} = \infty$ if $0 < \operatorname{Re}(c) < 1$.
- $e^{c\infty} = 0$ if $\operatorname{Re}(c) < 0$.
- $e^{c\infty} = \text{Undefined}$ if $\operatorname{Re}(c) = 0$.
- $e^{\infty} = \text{Undefined}$.
- Both *Undefined* and *Unknown* map to themselves.

The following symbolic simplifications are performed automatically:

- $e^0 = 1$
- $e^{\log(z)} = z$
- $e^{(p/q)\log(z)} = z^{p/q}$ (for rational p/q)
- $e^{(p/q)\pi i} = \text{algebraic root of unity}$ (for small rational p/q)

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(e^x)$.

void **ca_log**(*ca_t* res, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res* to the natural logarithm of *x*.

For special values and at the origin, the following definitions apply:

- For any infinity, $\log(c\infty) = \log(\infty) = +\infty$.
- $\log(0) = -\infty$. The result is *Unknown* if deciding $x = 0$ fails.
- Both *Undefined* and *Unknown* map to themselves.

The following symbolic simplifications are performed automatically:

- $\log(1) = 0$
- $\log(e^z) = z + 2\pi i k$
- $\log(\sqrt{z}) = \frac{1}{2} \log(z) + 2\pi i k$
- $\log(z^a) = a \log(z) + 2\pi i k$
- $\log(x) = \log(-x) + \pi i$ for negative real *x*

In the generic case, this function outputs an element of the formal field $\mathbb{Q}(\log(x))$.

10.4.20 Trigonometric functions

void **ca_sin_cos_exponential**(*ca_t* res1, *ca_t* res2, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_sin_cos_direct**(*ca_t* res1, *ca_t* res2, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_sin_cos_tangent**(*ca_t* res1, *ca_t* res2, const *ca_t* x, *ca_ctx_t* ctx)

void **ca_sin_cos**(*ca_t* res1, *ca_t* res2, const *ca_t* x, *ca_ctx_t* ctx)

Sets *res1* to the sine of *x* and *res2* to the cosine of *x*. Either *res1* or *res2* can be *NULL* to compute only the other function. Various representations are implemented:

- The *exponential* version expresses the sine and cosine in terms of complex exponentials. Simple algebraic values will simplify to rational numbers or elements of cyclotomic fields.

- The *direct* method expresses the sine and cosine in terms of the original functions (perhaps after applying some symmetry transformations, which may interchange sin and cos). Extremely simple algebraic values will automatically simplify to elements of real algebraic number fields.
- The *tangent* version expresses the sine and cosine in terms of $\tan(x/2)$, perhaps after applying some symmetry transformations. Extremely simple algebraic values will automatically simplify to elements of real algebraic number fields.

By default, the standard function uses the *exponential* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the `CA_OPT_TRIG_FORM` context setting.

For special values, the following definitions apply:

- $\sin(\pm i\infty) = \pm i\infty$
- $\cos(\pm i\infty) = +\infty$
- All other infinities give Undefined

```
void ca_sin(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_cos(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the sine or cosine of *x*. These functions are shortcuts for `ca_sin_cos()`.

```
void ca_tan_sine_cosine(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_tan_exponential(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_tan_direct(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_tan(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the tangent of *x*. The *sine_cosine* version evaluates the tangent as a quotient of a sine and cosine, the *direct* version evaluates it directly as a tangent (possibly after transforming the variable), and the *exponential* version evaluates it in terms of complex exponentials. Simple algebraic values will automatically simplify to elements of trigonometric or cyclotomic number fields.

By default, the standard function uses the *exponential* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the `CA_OPT_TRIG_FORM` context setting.

For special values, the following definitions apply:

- At poles, $\tan((n + \frac{1}{2})\pi) = \infty$
- $\tan(e^{i\theta}\infty) = +i, \quad 0 < \theta < \pi$
- $\tan(e^{i\theta}\infty) = -i, \quad -\pi < \theta < 0$
- $\tan(\pm\infty) = \tan(\infty) = \text{Undefined}$

```
void ca_cot(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the cotangent *x*. This is equivalent to computing the reciprocal of the tangent.

```
void ca_atan_logarithm(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_atan_direct(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_atan(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the inverse tangent of *x*.

The *direct* version expresses the result as an inverse tangent (possibly after transforming the variable). The *logarithm* version expresses it in terms of complex logarithms. Simple algebraic inputs will automatically simplify to rational multiples of π .

By default, the standard function uses the *logarithm* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be

changed using the `CA_OPT_TRIG_FORM` context setting (exponential mode results in logarithmic forms).

For special values, the following definitions apply:

- $\operatorname{atan}(\pm i) = \pm i\infty$
- $\operatorname{atan}(c\infty) = \operatorname{csgn}(c)\pi/2$
- $\operatorname{atan}(\infty) = \text{Undefined}$

```
void ca_asin_logarithm(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_acos_logarithm(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_asin_direct(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_acos_direct(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_asin(ca_t res, const ca_t x, ca_ctx_t ctx)
```

```
void ca_acos(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the inverse sine (respectively, cosine) of *x*.

The *direct* version expresses the result as an inverse sine or cosine (possibly after transforming the variable). The *logarithm* version expresses it in terms of complex logarithms. Simple algebraic inputs will automatically simplify to rational multiples of π .

By default, the standard function uses the *logarithm* representation as this typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. The behavior of the standard function can be changed using the `CA_OPT_TRIG_FORM` context setting (exponential mode results in logarithmic forms).

The inverse cosine is presently implemented as $\operatorname{acos}(x) = \pi/2 - \operatorname{asin}(x)$.

10.4.21 Special functions

```
void ca_gamma(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the gamma function of *x*.

```
void ca_erf(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the error function of *x*.

```
void ca_erfc(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the complementary error function of *x*.

```
void ca_erfi(ca_t res, const ca_t x, ca_ctx_t ctx)
```

Sets *res* to the imaginary error function of *x*.

10.4.22 Numerical evaluation

```
void ca_get_acb_raw(acb_t res, const ca_t x, slong prec, ca_ctx_t ctx)
```

Sets *res* to an enclosure of the numerical value of *x*. A working precision of *prec* bits is used internally for the evaluation, without adaptive refinement. If *x* is any special value, *res* is set to *acb_indeterminate*.

```
void ca_get_acb(acb_t res, const ca_t x, slong prec, ca_ctx_t ctx)
```

```
void ca_get_acb_accurate_parts(acb_t res, const ca_t x, slong prec, ca_ctx_t ctx)
```

Sets *res* to an enclosure of the numerical value of *x*. The working precision is increased adaptively to try to ensure *prec* accurate bits in the output. The *accurate_parts* version tries to ensure *prec* accurate bits for both the real and imaginary part separately.

The refinement is stopped if the working precision exceeds `CA_OPT_PREC_LIMIT` (or twice the initial precision, if this is larger). The user may call `acb_rel_accuracy_bits` to check if the calculation was successful.

The output is not rounded down to *prec* bits (to avoid unnecessary double rounding); the user may call `acb_set_round` when rounding is desired.

char `*ca_get_decimal_str`(const *ca_t* x, *slong* digits, *ulong* flags, *ca_ctx_t* ctx)

Returns a decimal approximation of *x* with precision up to *digits*. The output is guaranteed to be correct within 1 ulp in the returned digits, but the number of returned digits may be smaller than *digits* if the numerical evaluation does not succeed.

If *flags* is set to 1, attempts to achieve full accuracy for both the real and imaginary parts separately.

If *x* is not finite or a finite enclosure cannot be produced, returns the string “?”.

The user should free the returned string with `flint_free`.

10.4.23 Rewriting and simplification

void `ca_rewrite_complex_normal_form`(*ca_t* res, const *ca_t* x, int deep, *ca_ctx_t* ctx)

Sets *res* to *x* rewritten using standardizing transformations over the complex numbers:

- Elementary functions are rewritten in terms of (complex) exponentials, roots and logarithms
- Complex parts are rewritten using logarithms, square roots, and (deep) complex conjugates
- Algebraic numbers are rewritten in terms of cyclotomic fields where applicable

If *deep* is set, the rewriting is applied recursively to the tower of extension numbers; otherwise, the rewriting is only applied to the top-level extension numbers.

The result is not a normal form in the strong sense (the same number can have many possible representations even after applying this transformation), but in practice this is a powerful heuristic for simplification.

10.4.24 Factorization

type `ca_factor_struct`

type `ca_factor_t`

Represents a real or complex number in factored form $b_1^{e_1} b_2^{e_2} \dots b_n^{e_n}$ where b_i and e_i are *ca_t* numbers (the exponents need not be integers).

void `ca_factor_init`(*ca_factor_t* fac, *ca_ctx_t* ctx)

Initializes *fac* and sets it to the empty factorization (equivalent to the number 1).

void `ca_factor_clear`(*ca_factor_t* fac, *ca_ctx_t* ctx)

Clears the factorization structure *fac*.

void `ca_factor_one`(*ca_factor_t* fac, *ca_ctx_t* ctx)

Sets *fac* to the empty factorization (equivalent to the number 1).

void `ca_factor_print`(const *ca_factor_t* fac, *ca_ctx_t* ctx)

Prints a description of *fac* to standard output.

void `ca_factor_insert`(*ca_factor_t* fac, const *ca_t* base, const *ca_t* exp, *ca_ctx_t* ctx)

Inserts b^e into *fac* where *b* is given by *base* and *e* is given by *exp*. If a base element structurally identical to *base* already exists in *fac*, the corresponding exponent is incremented by *exp*; otherwise, this factor is appended.

void **ca_factor_get_ca**(*ca_t* res, const *ca_factor_t* fac, *ca_ctx_t* ctx)

Expands *fac* back to a single *ca_t* by evaluating the powers and multiplying out the result.

void **ca_factor**(*ca_factor_t* res, const *ca_t* x, *ulong* flags, *ca_ctx_t* ctx)

Sets *res* to a factorization of *x* of the form $x = b_1^{e_1} b_2^{e_2} \cdots b_n^{e_n}$. Requires that *x* is not a special value. The type of factorization is controlled by *flags*, which can be set to a combination of constants in the following section.

Factorization options

The following flags select the structural polynomial factorization to perform over formal fields $\mathbb{Q}(a_1, \dots, a_n)$. Each flag in the list strictly encompasses the factorization power of the preceding flag, so it is unnecessary to pass more than one flag.

CA_FACTOR_POLY_NONE

No polynomial factorization at all.

CA_FACTOR_POLY_CONTENT

Only extract the rational content.

CA_FACTOR_POLY_SQF

Perform a squarefree factorization in addition to extracting the rational content.

CA_FACTOR_POLY_FULL

Perform a full multivariate polynomial factorization.

The following flags select the factorization to perform over $\mathbb{Z}$. Integer factorization is applied if *x* is an element of $\mathbb{Q}$, and to the extracted rational content of polynomials. Each flag in the list strictly encompasses the factorization power of the preceding flag, so it is unnecessary to pass more than one flag.

CA_FACTOR_ZZ_NONE

No integer factorization at all.

CA_FACTOR_ZZ_SMOOTH

Perform a smooth factorization to extract small prime factors (heuristically up to **CA_OPT_SMOOTH_LIMIT** bits) in addition to identifying perfect powers.

CA_FACTOR_ZZ_FULL

Perform a complete integer factorization into prime numbers. This is prohibitively slow for general integers exceeding 70-80 digits.

10.4.25 Context options

The *options* member of a *ca_ctx_t* object is an array of *ulong* values controlling simplification behavior and various other settings. The values of the array at the following indices can be changed by the user (example: `ctx->options[CA_OPT_PREC_LIMIT] = 65536`).

It is recommended to set options controlling evaluation only at the time when a context object is created. Changing such options later should normally be harmless, but since the update will not apply retroactively to objects that have already been computed and cached, one might not see the expected behavior. Superficial options (printing) can be changed at any time.

CA_OPT_VERBOSE

Whether to print debug information. Default value: 0.

CA_OPT_PRINT_FLAGS

Printing style. See [Printing](#) for details. Default value: **CA_PRINT_DEFAULT**.

CA_OPT_MPOLY_ORD

Monomial ordering to use for multivariate polynomials. Possible values are `ORD_LEX`, `ORD_DEGLEX` and `ORD_DEGREVLEX`. Default value: `ORD_LEX`. This option must be set before doing any computations.

CA_OPT_PREC_LIMIT

Maximum precision to use internally for numerical evaluation with Arb, and in some cases for the magnitude of exact coefficients. This parameter affects the possibility to prove inequalities and find simplifications between related extension numbers. This is not a strict limit; some calculations may use higher precision when there is a good reason to do so. Default value: 4096.

CA_OPT_QQBAR_DEG_LIMIT

Maximum degree of `qqbar_t` elements allowed internally during simplification of algebraic numbers. This limit may be exceeded when the user provides explicit `qqbar_t` input of higher degree. Default value: 120.

CA_OPT_LOW_PREC

Numerical precision to use for fast checks (typically, before attempting more expensive operations). Default value: 64.

CA_OPT_SMOOTH_LIMIT

Size in bits for factors in smooth integer factorization. Default value: 32.

CA_OPT_LLL_PREC

Precision to use to find integer relations using LLL. Default value: 128.

CA_OPT_POW_LIMIT

Largest exponent to expand powers automatically. This only applies in multivariate and transcendental fields: in number fields, `CA_OPT_PREC_LIMIT` applies instead. Default value: 20.

CA_OPT_USE_GROEBNER

Boolean flag for whether to use Gröbner basis computation. This flag and the following limits affect the ability to prove multivariate identities. Default value: 1.

CA_OPT_GROEBNER_LENGTH_LIMIT

Maximum length of ideal basis allowed in Buchberger's algorithm. Default value: 100.

CA_OPT_GROEBNER_POLY_LENGTH_LIMIT

Maximum length of polynomials allowed in Buchberger's algorithm. Default value: 1000.

CA_OPT_GROEBNER_POLY_BITS_LIMIT

Maximum coefficient size in bits of polynomials allowed in Buchberger's algorithm. Default value: 10000.

CA_OPT_VIETA_LIMIT

Maximum degree n of algebraic numbers for which to add Vieta's formulas to the reduction ideal. This must be set relatively low since the number of terms in Vieta's formulas is $O(2^n)$ and the resulting Gröbner basis computations can be expensive. Default value: 6.

CA_OPT_TRIG_FORM

Default representation of trigonometric functions. The following values are possible:

CA_TRIG_DIRECT

Use the direct functions (with some exceptions).

CA_TRIG_EXPONENTIAL

Use complex exponentials.

CA_TRIG_SINE_COSINE

Use sines and cosines.

CA_TRIG_TANGENT

Use tangents.

Default value: CA_TRIG_EXPONENTIAL.

The *exponential* representation is currently used by default as typically works best for field arithmetic and simplifications, although it has the disadvantage of introducing complex numbers where real numbers would be sufficient. This may change in the future.

10.4.26 Internal representation

CA_FMPQ(x)**CA_FMPQ_NUMREF(x)****CA_FMPQ_DENREF(x)**

Assuming that x holds an element of the trivial field $\mathbb{Q}$, this macro returns a pointer which can be used as an *fmpq_t*, or respectively to the numerator or denominator as an *fmpz_t*.

CA_MPOLY_Q(x)

Assuming that x holds a generic field element as data, this macro returns a pointer which can be used as an *fmpz_poly_q_t*.

CA_NF_ELEM(x)

Assuming that x holds an Antic number field element as data, this macro returns a pointer which can be used as an *nf_elem_t*.

void **_ca_make_field_element**(*ca_t* x, *ca_field_srcptr* new_index, *ca_ctx_t* ctx)

Changes the internal representation of x to that of an element of the field with index *new_index* in the context object *ctx*. This may destroy the value of x .

void **_ca_make_fmpq**(*ca_t* x, *ca_ctx_t* ctx)

Changes the internal representation of x to that of an element of the trivial field $\mathbb{Q}$. This may destroy the value of x .

10.5 `ca_vec.h` – vectors of real and complex numbers

A `ca_vec_t` represents a vector of real or complex numbers, implemented as an array of coefficients of type `ca_struct`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients (taking `ca_ptr` and `ca_srcptr` arguments), and a non-underscore method which takes `ca_vec_t` input and performs automatic memory management.

Unlike `ca_poly_t`, a `ca_vec_t` is not normalised by removing zero coefficients; it retains the exact length assigned by the user.

10.5.1 Types, macros and constants

type `ca_vec_struct`

type `ca_vec_t`

Contains a pointer to an array of entries (*coeffs*), the used length (*length*), and the allocated size of the array (*alloc*).

A `ca_vec_t` is defined as an array of length one of type `ca_vec_struct`, permitting an `ca_vec_t` to be passed by reference.

`ca_vec_entry`(*vec*, *i*)

Macro returning a pointer to entry *i* in the vector *vec*. The index must be in bounds.

10.5.2 Memory management

`ca_ptr_ca_vec_init`(*slong* *len*, `ca_ctx_t` *ctx*)

Returns a pointer to an array of *len* coefficients initialized to zero.

void `ca_vec_init`(`ca_vec_t` *vec*, *slong* *len*, `ca_ctx_t` *ctx*)

Initializes *vec* to a length *len* vector. All entries are set to zero.

void `_ca_vec_clear`(`ca_ptr` *vec*, *slong* *len*, `ca_ctx_t` *ctx*)

Clears all *len* entries in *vec* and frees the pointer *vec* itself.

void `ca_vec_clear`(`ca_vec_t` *vec*, `ca_ctx_t` *ctx*)

Clears the vector *vec*.

void `_ca_vec_swap`(`ca_ptr` *vec1*, `ca_ptr` *vec2*, *slong* *len*, `ca_ctx_t` *ctx*)

Swaps the entries in *vec1* and *vec2* efficiently.

void `ca_vec_swap`(`ca_vec_t` *vec1*, `ca_vec_t` *vec2*, `ca_ctx_t` *ctx*)

Swaps the vectors *vec1* and *vec2* efficiently.

10.5.3 Length

slong `ca_vec_length`(const `ca_vec_t` *vec*, `ca_ctx_t` *ctx*)

Returns the length of *vec*.

void `_ca_vec_fit_length`(`ca_vec_t` *vec*, *slong* *len*, `ca_ctx_t` *ctx*)

Allocates space in *vec* for *len* elements.

void `ca_vec_set_length`(`ca_vec_t` *vec*, *slong* *len*, `ca_ctx_t` *ctx*)

Sets the length of *vec* to *len*. If *vec* is shorter on input, it will be zero-extended. If *vec* is longer on input, it will be truncated.

10.5.4 Assignment

void `_ca_vec_set`(*ca_ptr* res, *ca_srcptr* src, *slong* len, *ca_ctx_t* ctx)

Sets *res* to a copy of *src* of length *len*.

void `ca_vec_set`(*ca_vec_t* res, const *ca_vec_t* src, *ca_ctx_t* ctx)

Sets *res* to a copy of *src*.

10.5.5 Special vectors

void `_ca_vec_zero`(*ca_ptr* res, *slong* len, *ca_ctx_t* ctx)

Sets the *len* entries in *res* to zeros.

void `ca_vec_zero`(*ca_vec_t* res, *slong* len, *ca_ctx_t* ctx)

Sets *res* to the length *len* zero vector.

10.5.6 Input and output

void `ca_vec_print`(const *ca_vec_t* vec, *ca_ctx_t* ctx)

Prints *vec* to standard output. The coefficients are printed on separate lines.

void `ca_vec_printn`(const *ca_vec_t* vec, *slong* digits, *ca_ctx_t* ctx)

Prints a decimal representation of *vec* with precision specified by *digits*. The coefficients are comma-separated and the whole list is enclosed in square brackets.

10.5.7 List operations

void `ca_vec_append`(*ca_vec_t* vec, const *ca_t* f, *ca_ctx_t* ctx)

Appends *f* to the end of *vec*.

10.5.8 Arithmetic

void `_ca_vec_neg`(*ca_ptr* res, *ca_srcptr* src, *slong* len, *ca_ctx_t* ctx)

void `ca_vec_neg`(*ca_vec_t* res, const *ca_vec_t* src, *ca_ctx_t* ctx)

Sets *res* to the negation of *src*.

void `_ca_vec_add`(*ca_ptr* res, *ca_srcptr* vec1, *ca_srcptr* vec2, *slong* len, *ca_ctx_t* ctx)

void `_ca_vec_sub`(*ca_ptr* res, *ca_srcptr* vec1, *ca_srcptr* vec2, *slong* len, *ca_ctx_t* ctx)

Sets *res* to the sum or difference of *vec1* and *vec2*, all vectors having length *len*.

void `_ca_vec_scalar_mul_ca`(*ca_ptr* res, *ca_srcptr* src, *slong* len, const *ca_t* c, *ca_ctx_t* ctx)

Sets *res* to *src* multiplied by *c*, all vectors having length *len*.

void `_ca_vec_scalar_div_ca`(*ca_ptr* res, *ca_srcptr* src, *slong* len, const *ca_t* c, *ca_ctx_t* ctx)

Sets *res* to *src* divided by *c*, all vectors having length *len*.

void `_ca_vec_scalar_addmul_ca`(*ca_ptr* res, *ca_srcptr* src, *slong* len, const *ca_t* c, *ca_ctx_t* ctx)

Adds *src* multiplied by *c* to the vector *res*, all vectors having length *len*.

void `_ca_vec_scalar_submul_ca`(*ca_ptr* res, *ca_srcptr* src, *slong* len, const *ca_t* c, *ca_ctx_t* ctx)

Subtracts *src* multiplied by *c* from the vector *res*, all vectors having length *len*.

10.5.9 Comparisons and properties

truth_t `_ca_vec_check_is_zero`(*ca_srcptr* *vec*, *slong* *len*, *ca_ctx_t* *ctx*)

Returns whether *vec* is the zero vector.

10.5.10 Internal representation

int `_ca_vec_is_fmpq_vec`(*ca_srcptr* *vec*, *slong* *len*, *ca_ctx_t* *ctx*)

Checks if all elements of *vec* are structurally rational numbers.

int `_ca_vec_fmpq_vec_is_fmpz_vec`(*ca_srcptr* *vec*, *slong* *len*, *ca_ctx_t* *ctx*)

Assuming that all elements of *vec* are structurally rational numbers, checks if all elements are integers.

void `_ca_vec_fmpq_vec_get_fmpz_vec_den`(*fmpz_t* *c*, *fmpz_t* *den*, *ca_srcptr* *vec*, *slong* *len*, *ca_ctx_t* *ctx*)

Assuming that all elements of *vec* are structurally rational numbers, converts them to a vector of integers *c* on a common denominator *den*.

void `_ca_vec_set_fmpz_vec_div_fmpz`(*ca_ptr* *res*, const *fmpz_t* *v*, const *fmpz_t* *den*, *slong* *len*, *ca_ctx_t* *ctx*)

Sets *res* to the rational vector given by numerators *v* and the common denominator *den*.

10.6 `ca_poly.h` – dense univariate polynomials over the real and complex numbers

A `ca_poly_t` represents a univariate polynomial over the real or complex numbers (an element of $\mathbb{R}[X]$ or $\mathbb{C}[X]$), implemented as an array of coefficients of type `ca_struct`.

Most functions are provided in two versions: an underscore method which operates directly on pre-allocated arrays of coefficients and generally has some restrictions (such as requiring the lengths to be nonzero and not supporting aliasing of the input and output arrays), and a non-underscore method which performs automatic memory management and handles degenerate cases.

Warnings:

- A polynomial is always normalised by removing zero coefficients at the top. Coefficients will not be removed when Calcium is unable to prove that they are zero. The represented degree can therefore be larger than the degree of the mathematical polynomial. When the correct degree is needed, it is important to verify the leading coefficient. (Of course, this will never be an issue with polynomials that are explicitly monic, for example.)
- The special values *Undefined*, unsigned infinity and signed infinity supported by the scalar `ca_t` type are not really meaningful as coefficients of polynomials. We normally assume that the user does not assign those values to coefficients of polynomials, and the functions in this module will likewise normally not generate such coefficients. *Unknown* can still appear as a coefficient representing a number that is inaccessible for computation.

A polynomial with numerical coefficients and with a nonzero leading coefficient is called *proper*. The function `ca_poly_is_proper()` can be used to check for violations.

10.6.1 Types, macros and constants

type `ca_poly_struct`

type `ca_poly_t`

Contains a pointer to an array of coefficients (*coeffs*), the used length (*length*), and the allocated size of the array (*alloc*).

A `ca_poly_t` is defined as an array of length one of type `ca_poly_struct`, permitting an `ca_poly_t` to be passed by reference.

10.6.2 Memory management

void `ca_poly_init(ca_poly_t poly, ca_ctx_t ctx)`

Initializes the polynomial for use, setting it to the zero polynomial.

void `ca_poly_clear(ca_poly_t poly, ca_ctx_t ctx)`

Clears the polynomial, deallocating all coefficients and the coefficient array.

void `ca_poly_fit_length(ca_poly_t poly, slong len, ca_ctx_t ctx)`

Makes sure that the coefficient array of the polynomial contains at least *len* initialized coefficients.

void `_ca_poly_set_length(ca_poly_t poly, slong len, ca_ctx_t ctx)`

Directly changes the length of the polynomial, without allocating or deallocating coefficients. The value should not exceed the allocation length.

void `_ca_poly_normalise(ca_poly_t poly, ca_ctx_t ctx)`

Strips any top coefficients which can be proved identical to zero.

10.6.3 Assignment and simple values

void **ca_poly_zero**(*ca_poly_t* poly, *ca_ctx_t* ctx)

Sets *poly* to the zero polynomial.

void **ca_poly_one**(*ca_poly_t* poly, *ca_ctx_t* ctx)

Sets *poly* to the constant polynomial 1.

void **ca_poly_x**(*ca_poly_t* poly, *ca_ctx_t* ctx)

Sets *poly* to the monomial x .

void **ca_poly_set_ca**(*ca_poly_t* poly, const *ca_t* c, *ca_ctx_t* ctx)

void **ca_poly_set_si**(*ca_poly_t* poly, *slong* c, *ca_ctx_t* ctx)

Sets *poly* to the constant polynomial c .

void **ca_poly_set**(*ca_poly_t* res, const *ca_poly_t* src, *ca_ctx_t* ctx)

void **ca_poly_set_fmpz_poly**(*ca_poly_t* res, const *fmpz_poly_t* src, *ca_ctx_t* ctx)

void **ca_poly_set_fmpq_poly**(*ca_poly_t* res, const *fmpq_poly_t* src, *ca_ctx_t* ctx)

Sets *poly* the polynomial *src*.

void **ca_poly_set_coeff_ca**(*ca_poly_t* poly, *slong* n, const *ca_t* x, *ca_ctx_t* ctx)

Sets the coefficient at position n in *poly* to x .

void **ca_poly_transfer**(*ca_poly_t* res, *ca_ctx_t* res_ctx, const *ca_poly_t* src, *ca_ctx_t* src_ctx)

Sets *res* to *src* where the corresponding context objects *res_ctx* and *src_ctx* may be different.

This operation preserves the mathematical value represented by *src*, but may result in a different internal representation depending on the settings of the context objects.

10.6.4 Random generation

void **ca_poly_randtest**(*ca_poly_t* poly, *flint_rand_t* state, *slong* len, *slong* depth, *slong* bits, *ca_ctx_t* ctx)

Sets *poly* to a random polynomial of length up to *len* and with entries having complexity up to *depth* and *bits* (see *ca_randtest()*).

void **ca_poly_randtest_rational**(*ca_poly_t* poly, *flint_rand_t* state, *slong* len, *slong* bits, *ca_ctx_t* ctx)

Sets *poly* to a random rational polynomial of length up to *len* and with entries up to *bits* bits in size.

10.6.5 Input and output

void **ca_poly_print**(const *ca_poly_t* poly, *ca_ctx_t* ctx)

Prints *poly* to standard output. The coefficients are printed on separate lines.

void **ca_poly_printn**(const *ca_poly_t* poly, *slong* digits, *ca_ctx_t* ctx)

Prints a decimal representation of *poly* with precision specified by *digits*. The coefficients are comma-separated and the whole list is enclosed in square brackets.

10.6.6 Degree and leading coefficient

int `ca_poly_is_proper`(const `ca_poly_t` poly, `ca_ctx_t` ctx)

Checks that *poly* represents an element of $\mathbb{C}[X]$ with well-defined degree. This returns 1 if the leading coefficient of *poly* is nonzero and all coefficients of *poly* are numbers (not special values). It returns 0 otherwise. It returns 1 when *poly* is precisely the zero polynomial (which does not have a leading coefficient).

int `ca_poly_make_monic`(`ca_poly_t` res, const `ca_poly_t` poly, `ca_ctx_t` ctx)

Makes *poly* monic by dividing by the leading coefficient if possible and returns 1. Returns 0 if the leading coefficient cannot be certified to be nonzero, or if *poly* is the zero polynomial.

void `_ca_poly_reverse`(`ca_ptr` res, `ca_srcptr` poly, *slong* len, *slong* n, `ca_ctx_t` ctx)

void `ca_poly_reverse`(`ca_poly_t` res, const `ca_poly_t` poly, *slong* n, `ca_ctx_t` ctx)

Sets *res* to the reversal of *poly* considered as a polynomial of length *n*, zero-padding if needed. The underscore method assumes that *len* is positive and less than or equal to *n*.

10.6.7 Comparisons

truth_t `_ca_poly_check_equal`(`ca_srcptr` poly1, *slong* len1, `ca_srcptr` poly2, *slong* len2, `ca_ctx_t` ctx)

truth_t `ca_poly_check_equal`(const `ca_poly_t` poly1, const `ca_poly_t` poly2, `ca_ctx_t` ctx)

Checks if *poly1* and *poly2* represent the same polynomial. The underscore method assumes that *len1* is at least as large as *len2*.

truth_t `ca_poly_check_is_zero`(const `ca_poly_t` poly, `ca_ctx_t` ctx)

Checks if *poly* is the zero polynomial.

truth_t `ca_poly_check_is_one`(const `ca_poly_t` poly, `ca_ctx_t` ctx)

Checks if *poly* is the constant polynomial 1.

10.6.8 Arithmetic

void `_ca_poly_shift_left`(`ca_ptr` res, `ca_srcptr` poly, *slong* len, *slong* n, `ca_ctx_t` ctx)

void `ca_poly_shift_left`(`ca_poly_t` res, const `ca_poly_t` poly, *slong* n, `ca_ctx_t` ctx)

Sets *res* to *poly* shifted *n* coefficients to the left; that is, multiplied by x^n .

void `_ca_poly_shift_right`(`ca_ptr` res, `ca_srcptr` poly, *slong* len, *slong* n, `ca_ctx_t` ctx)

void `ca_poly_shift_right`(`ca_poly_t` res, const `ca_poly_t` poly, *slong* n, `ca_ctx_t` ctx)

Sets *res* to *poly* shifted *n* coefficients to the right; that is, divided by x^n .

void `ca_poly_neg`(`ca_poly_t` res, const `ca_poly_t` src, `ca_ctx_t` ctx)

Sets *res* to the negation of *src*.

void `_ca_poly_add`(`ca_ptr` res, `ca_srcptr` poly1, *slong* len1, `ca_srcptr` poly2, *slong* len2, `ca_ctx_t` ctx)

void `ca_poly_add`(`ca_poly_t` res, const `ca_poly_t` poly1, const `ca_poly_t` poly2, `ca_ctx_t` ctx)

Sets *res* to the sum of *poly1* and *poly2*.

void `_ca_poly_sub`(`ca_ptr` res, `ca_srcptr` poly1, *slong* len1, `ca_srcptr` poly2, *slong* len2, `ca_ctx_t` ctx)

void `ca_poly_sub`(`ca_poly_t` res, const `ca_poly_t` poly1, const `ca_poly_t` poly2, `ca_ctx_t` ctx)

Sets *res* to the difference of *poly1* and *poly2*.

void `_ca_poly_mul`(`ca_ptr` res, `ca_srcptr` poly1, *slong* len1, `ca_srcptr` poly2, *slong* len2, `ca_ctx_t` ctx)

```
void ca_poly_mul(ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, ca_ctx_t ctx)
    Sets res to the product of poly1 and poly2.

void _ca_poly_mullo((ca_ptr C, ca_srcptr poly1, slong len1, ca_srcptr poly2, slong len2, slong n,
    ca_ctx_t ctx)

void ca_poly_mullo(ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, slong n,
    ca_ctx_t ctx)
    Sets res to the product of poly1 and poly2 truncated to length n.

void ca_poly_mul_ca(ca_poly_t res, const ca_poly_t poly, const ca_t c, ca_ctx_t ctx)
    Sets res to poly multiplied by the scalar c.

void ca_poly_div_ca(ca_poly_t res, const ca_poly_t poly, const ca_t c, ca_ctx_t ctx)
    Sets res to poly divided by the scalar c.

void _ca_poly_divrem_basecase(ca_ptr Q, ca_ptr R, ca_srcptr A, slong lenA, ca_srcptr B, slong
    lenB, const ca_t invB, ca_ctx_t ctx)

int ca_poly_divrem_basecase(ca_poly_t Q, ca_poly_t R, const ca_poly_t A, const ca_poly_t B,
    ca_ctx_t ctx)

void _ca_poly_divrem(ca_ptr Q, ca_ptr R, ca_srcptr A, slong lenA, ca_srcptr B, slong lenB, const
    ca_t invB, ca_ctx_t ctx)

int ca_poly_divrem(ca_poly_t Q, ca_poly_t R, const ca_poly_t A, const ca_poly_t B, ca_ctx_t
    ctx)

int ca_poly_div(ca_poly_t Q, const ca_poly_t A, const ca_poly_t B, ca_ctx_t ctx)

int ca_poly_rem(ca_poly_t R, const ca_poly_t A, const ca_poly_t B, ca_ctx_t ctx)
    If the leading coefficient of B can be proved invertible, sets Q and R to the quotient and remainder
    of polynomial division of A by B and returns 1. If the leading coefficient cannot be proved invertible,
    returns 0. The underscore method takes a precomputed inverse of the leading coefficient of B.

void _ca_poly_pow_ui_trunc(ca_ptr res, ca_srcptr f, slong flen, ulong exp, slong len, ca_ctx_t ctx)

void ca_poly_pow_ui_trunc(ca_poly_t res, const ca_poly_t poly, ulong exp, slong len, ca_ctx_t ctx)
    Sets res to poly raised to the power exp, truncated to length len.

void _ca_poly_pow_ui(ca_ptr res, ca_srcptr f, slong flen, ulong exp, ca_ctx_t ctx)

void ca_poly_pow_ui(ca_poly_t res, const ca_poly_t poly, ulong exp, ca_ctx_t ctx)
    Sets res to poly raised to the power exp.
```

10.6.9 Evaluation and composition

```
void _ca_poly_evaluate_horner(ca_t res, ca_srcptr f, slong len, const ca_t x, ca_ctx_t ctx)

void ca_poly_evaluate_horner(ca_t res, const ca_poly_t f, const ca_t a, ca_ctx_t ctx)

void _ca_poly_evaluate(ca_t res, ca_srcptr f, slong len, const ca_t x, ca_ctx_t ctx)

void ca_poly_evaluate(ca_t res, const ca_poly_t f, const ca_t a, ca_ctx_t ctx)
    Sets res to f evaluated at the point a.

void _ca_poly_compose(ca_ptr res, ca_srcptr poly1, slong len1, ca_srcptr poly2, slong len2,
    ca_ctx_t ctx)

void ca_poly_compose(ca_poly_t res, const ca_poly_t poly1, const ca_poly_t poly2, ca_ctx_t ctx)
    Sets res to the composition of poly1 with poly2.
```

10.6.10 Derivative and integral

void `_ca_poly_derivative`(*ca_ptr* res, *ca_srcptr* poly, *slong* len, *ca_ctx_t* ctx)

void `ca_poly_derivative`(*ca_poly_t* res, const *ca_poly_t* poly, *ca_ctx_t* ctx)

Sets *res* to the derivative of *poly*. The underscore method needs one less coefficient than *len* for the output array.

void `_ca_poly_integral`(*ca_ptr* res, *ca_srcptr* poly, *slong* len, *ca_ctx_t* ctx)

void `ca_poly_integral`(*ca_poly_t* res, const *ca_poly_t* poly, *ca_ctx_t* ctx)

Sets *res* to the integral of *poly*. The underscore method needs one more coefficient than *len* for the output array.

10.6.11 Power series division

void `_ca_poly_inv_series`(*ca_ptr* res, *ca_srcptr* f, *slong* flen, *slong* len, *ca_ctx_t* ctx)

void `ca_poly_inv_series`(*ca_poly_t* res, const *ca_poly_t* f, *slong* len, *ca_ctx_t* ctx)

Sets *res* to the power series inverse of *f* truncated to length *len*.

void `_ca_poly_div_series`(*ca_ptr* res, *ca_srcptr* f, *slong* flen, *ca_srcptr* g, *slong* glen, *slong* len, *ca_ctx_t* ctx)

void `ca_poly_div_series`(*ca_poly_t* res, const *ca_poly_t* f, const *ca_poly_t* g, *slong* len, *ca_ctx_t* ctx)

Sets *res* to the power series quotient of *f* and *g* truncated to length *len*. This function divides by zero if *g* has constant term zero; the user should manually remove initial zeros when an exact cancellation is required.

10.6.12 Elementary functions

void `_ca_poly_exp_series`(*ca_ptr* res, *ca_srcptr* f, *slong* flen, *slong* len, *ca_ctx_t* ctx)

void `ca_poly_exp_series`(*ca_poly_t* res, const *ca_poly_t* f, *slong* len, *ca_ctx_t* ctx)

Sets *res* to the power series exponential of *f* truncated to length *len*.

void `_ca_poly_log_series`(*ca_ptr* res, *ca_srcptr* f, *slong* flen, *slong* len, *ca_ctx_t* ctx)

void `ca_poly_log_series`(*ca_poly_t* res, const *ca_poly_t* f, *slong* len, *ca_ctx_t* ctx)

Sets *res* to the power series logarithm of *f* truncated to length *len*.

10.6.13 Greatest common divisor

slong `_ca_poly_gcd_euclidean`(*ca_ptr* res, *ca_srcptr* A, *slong* lenA, *ca_srcptr* B, *slong* lenB, *ca_ctx_t* ctx)

int `ca_poly_gcd_euclidean`(*ca_poly_t* res, const *ca_poly_t* A, const *ca_poly_t* B, *ca_ctx_t* ctx)

slong `_ca_poly_gcd`(*ca_ptr* res, *ca_srcptr* A, *slong* lenA, *ca_srcptr* B, *slong* lenB, *ca_ctx_t* ctx)

int `ca_poly_gcd`(*ca_poly_t* res, const *ca_poly_t* A, const *ca_poly_t* B, *ca_ctx_t* ctx)

Sets *res* to the GCD of *A* and *B* and returns 1 on success. On failure, returns 0 leaving the value of *res* arbitrary. The computation can fail if testing a leading coefficient for zero fails in the execution of the GCD algorithm. The output is normalized to be monic if it is not the zero polynomial.

The underscore methods assume $\text{lenA} \geq \text{lenB} \geq 1$, and that both *A* and *B* have nonzero leading coefficient. They return the length of the GCD, or 0 if the computation fails.

The *euclidean* version implements the standard Euclidean algorithm. The default version first checks for rational polynomials or attempts to certify numerically that the polynomials are co-prime and otherwise falls back to an automatic choice of algorithm (currently only the Euclidean algorithm).

10.6.14 Roots and factorization

```
int ca_poly_factor_squarefree(ca_t c, ca_poly_vec_t fac, ulong *exp, const ca_poly_t F,
                             ca_ctx_t ctx)
```

Computes the squarefree factorization of F , giving a product $F = cf_1f_2^2 \dots f_n^n$ where all f_i with $f_i \neq 1$ are squarefree and pairwise coprime. The nontrivial factors f_i are written to *fac* and the corresponding exponents are written to *exp*. This algorithm can fail if GCD computation fails internally. Returns 1 on success and 0 on failure.

```
int ca_poly_squarefree_part(ca_poly_t res, const ca_poly_t poly, ca_ctx_t ctx)
```

Sets *res* to the squarefree part of *poly*, normalized to be monic. This algorithm can fail if GCD computation fails internally. Returns 1 on success and 0 on failure.

```
void _ca_poly_set_roots(ca_ptr poly, ca_srcptr roots, const ulong *exp, slong n, ca_ctx_t ctx)
```

```
void ca_poly_set_roots(ca_poly_t poly, ca_vec_t roots, const ulong *exp, ca_ctx_t ctx)
```

Sets *poly* to the monic polynomial with the n roots given in the vector *roots*, with multiplicities given in the vector *exp*. In other words, this constructs the polynomial $(x - r_0)^{e_0}(x - r_1)^{e_1} \dots (x - r_{n-1})^{e_{n-1}}$. Uses binary splitting.

```
int _ca_poly_roots(ca_ptr roots, ca_srcptr poly, slong len, ca_ctx_t ctx)
```

```
int ca_poly_roots(ca_vec_t roots, ulong *exp, const ca_poly_t poly, ca_ctx_t ctx)
```

Attempts to compute all complex roots of the given polynomial *poly*. On success, returns 1 and sets *roots* to a vector containing all the distinct roots with corresponding multiplicities in *exp*. On failure, returns 0 and leaves the values in *roots* arbitrary. The roots are returned in arbitrary order.

Failure will occur if the leading coefficient of *poly* cannot be proved to be nonzero, if determining the correct multiplicities fails, or if the builtin algorithms do not have a means to represent the roots symbolically.

The underscore method assumes that the polynomial is squarefree. The non-underscore method performs a squarefree factorization.

10.6.15 Vectors of polynomials

```
type ca_poly_vec_struct
```

```
type ca_poly_vec_t
```

Represents a vector of polynomials.

```
ca_poly_struct *_ca_poly_vec_init(slong len, ca_ctx_t ctx)
```

```
void ca_poly_vec_init(ca_poly_vec_t res, slong len, ca_ctx_t ctx)
```

Initializes a vector with *len* polynomials.

```
void _ca_poly_vec_fit_length(ca_poly_vec_t vec, slong len, ca_ctx_t ctx)
```

Allocates space for *len* polynomials in *vec*.

```
void ca_poly_vec_set_length(ca_poly_vec_t vec, slong len, ca_ctx_t ctx)
```

Resizes *vec* to length *len*, zero-extending if needed.

```
void _ca_poly_vec_clear(ca_poly_struct *vec, slong len, ca_ctx_t ctx)
```

```
void ca_poly_vec_clear(ca_poly_vec_t vec, ca_ctx_t ctx)
```

Clears the vector *vec*.

```
void ca_poly_vec_append(ca_poly_vec_t vec, const ca_poly_t poly, ca_ctx_t ctx)
```

Appends *poly* to the end of the vector *vec*.

10.7 `ca_mat.h` – matrices over the real and complex numbers

A `ca_mat_t` represents a dense matrix over the real or complex numbers, implemented as an array of entries of type `ca_struct`. The dimension (number of rows and columns) of a matrix is fixed at initialization, and the user must ensure that inputs and outputs to an operation have compatible dimensions. The number of rows or columns in a matrix can be zero.

10.7.1 Types, macros and constants

type `ca_mat_struct`

type `ca_mat_t`

Contains a pointer to a flat array of the entries (*entries*), an array of pointers to the start of each row (*rows*), and the number of rows (*r*) and columns (*c*).

A `ca_mat_t` is defined as an array of length one of type `ca_mat_struct`, permitting a `ca_mat_t` to be passed by reference.

`ca_mat_entry(mat, i, j)`

Macro giving a pointer to the entry at row *i* and column *j*.

`ca_mat_nrows(mat)`

Returns the number of rows of the matrix.

`ca_mat_ncols(mat)`

Returns the number of columns of the matrix.

`ca_ptr ca_mat_entry_ptr(ca_mat_t mat, slong i, slong j)`

Returns a pointer to the entry at row *i* and column *j*. Equivalent to `ca_mat_entry` but implemented as a function.

10.7.2 Memory management

void `ca_mat_init(ca_mat_t mat, slong r, slong c, ca_ctx_t ctx)`

Initializes the matrix, setting it to the zero matrix with *r* rows and *c* columns.

void `ca_mat_clear(ca_mat_t mat, ca_ctx_t ctx)`

Clears the matrix, deallocating all entries.

void `ca_mat_swap(ca_mat_t mat1, ca_mat_t mat2, ca_ctx_t ctx)`

Efficiently swaps *mat1* and *mat2*.

void `ca_mat_window_init(ca_mat_t window, const ca_mat_t mat, slong r1, slong c1, slong r2, slong c2, ca_ctx_t ctx)`

Initializes *window* to a window matrix into the submatrix of *mat* starting at the corner at row *r1* and column *c1* (inclusive) and ending at row *r2* and column *c2* (exclusive).

void `ca_mat_window_clear(ca_mat_t window, ca_ctx_t ctx)`

Frees the window matrix.

10.7.3 Assignment and conversions

void **ca_mat_set**(*ca_mat_t* dest, const *ca_mat_t* src, *ca_ctx_t* ctx)

void **ca_mat_set_fmpz_mat**(*ca_mat_t* dest, const *fmpz_mat_t* src, *ca_ctx_t* ctx)

void **ca_mat_set_fmpq_mat**(*ca_mat_t* dest, const *fmpq_mat_t* src, *ca_ctx_t* ctx)

Sets *dest* to *src*. The operands must have identical dimensions.

void **ca_mat_set_ca**(*ca_mat_t* mat, const *ca_t* c, *ca_ctx_t* ctx)

Sets *mat* to the matrix with the scalar *c* on the main diagonal and zeros elsewhere.

void **ca_mat_transfer**(*ca_mat_t* res, *ca_ctx_t* res_ctx, const *ca_mat_t* src, *ca_ctx_t* src_ctx)

Sets *res* to *src* where the corresponding context objects *res_ctx* and *src_ctx* may be different.

This operation preserves the mathematical value represented by *src*, but may result in a different internal representation depending on the settings of the context objects.

10.7.4 Random generation

void **ca_mat_randtest**(*ca_mat_t* mat, *flint_rand_t* state, *slong* depth, *slong* bits, *ca_ctx_t* ctx)

Sets *mat* to a random matrix with entries having complexity up to *depth* and *bits* (see *ca_randtest()*).

void **ca_mat_randtest_rational**(*ca_mat_t* mat, *flint_rand_t* state, *slong* bits, *ca_ctx_t* ctx)

Sets *mat* to a random rational matrix with entries up to *bits* bits in size.

void **ca_mat_randops**(*ca_mat_t* mat, *flint_rand_t* state, *slong* count, *ca_ctx_t* ctx)

Randomizes *mat* in-place by performing elementary row or column operations. More precisely, at most count random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, the determinant) unchanged.

10.7.5 Input and output

void **ca_mat_print**(const *ca_mat_t* mat, *ca_ctx_t* ctx)

Prints *mat* to standard output. The entries are printed on separate lines.

void **ca_mat_printn**(const *ca_mat_t* mat, *slong* digits, *ca_ctx_t* ctx)

Prints a decimal representation of *mat* with precision specified by *digits*. The entries are comma-separated with square brackets and comma separation for the rows.

10.7.6 Special matrices

void **ca_mat_zero**(*ca_mat_t* mat, *ca_ctx_t* ctx)

Sets all entries in *mat* to zero.

void **ca_mat_one**(*ca_mat_t* mat, *ca_ctx_t* ctx)

Sets the entries on the main diagonal of *mat* to one, and all other entries to zero.

void **ca_mat_ones**(*ca_mat_t* mat, *ca_ctx_t* ctx)

Sets all entries in *mat* to one.

void **ca_mat_pascal**(*ca_mat_t* mat, int triangular, *ca_ctx_t* ctx)

Sets *mat* to a Pascal matrix, whose entries are binomial coefficients. If *triangular* is 0, constructs a full symmetric matrix with the rows of Pascal's triangle as successive antidiagonals. If *triangular* is 1, constructs the upper triangular matrix with the rows of Pascal's triangle as columns, and if *triangular* is -1, constructs the lower triangular matrix with the rows of Pascal's triangle as rows.

void **ca_mat_stirling**(*ca_mat_t* mat, int kind, *ca_ctx_t* ctx)

Sets *mat* to a Stirling matrix, whose entries are Stirling numbers. If *kind* is 0, the entries are set to the unsigned Stirling numbers of the first kind. If *kind* is 1, the entries are set to the signed Stirling numbers of the first kind. If *kind* is 2, the entries are set to the Stirling numbers of the second kind.

void **ca_mat_hilbert**(*ca_mat_t* mat, *ca_ctx_t* ctx)

Sets *mat* to the Hilbert matrix, which has entries $A_{i,j} = 1/(i + j + 1)$.

void **ca_mat_dft**(*ca_mat_t* mat, int type, *ca_ctx_t* ctx)

Sets *mat* to the DFT (discrete Fourier transform) matrix of order n where n is the smallest dimension of *mat* (if *mat* is not square, the matrix is extended periodically along the larger dimension). The *type* parameter selects between four different versions of the DFT matrix (in which $\omega = e^{2\pi i/n}$):

- Type 0 – entries $A_{j,k} = \omega^{-jk}$
- Type 1 – entries $A_{j,k} = \omega^{jk}/n$
- Type 2 – entries $A_{j,k} = \omega^{-jk}/\sqrt{n}$
- Type 3 – entries $A_{j,k} = \omega^{jk}/\sqrt{n}$

The type 0 and 1 matrices are inverse pairs, and similarly for the type 2 and 3 matrices.

10.7.7 Comparisons and properties

truth_t **ca_mat_check_equal**(const *ca_mat_t* A, const *ca_mat_t* B, *ca_ctx_t* ctx)

Compares *A* and *B* for equality.

truth_t **ca_mat_check_is_zero**(const *ca_mat_t* A, *ca_ctx_t* ctx)

Tests if *A* is the zero matrix.

truth_t **ca_mat_check_is_one**(const *ca_mat_t* A, *ca_ctx_t* ctx)

Tests if *A* has ones on the main diagonal and zeros elsewhere.

10.7.8 Conjugate and transpose

void **ca_mat_transpose**(*ca_mat_t* res, const *ca_mat_t* A, *ca_ctx_t* ctx)

Sets *res* to the transpose of *A*. The operands must have compatible dimensions. Aliasing is allowed for square matrices.

void **ca_mat_conj**(*ca_mat_t* res, const *ca_mat_t* A, *ca_ctx_t* ctx)

Sets *res* to the entrywise complex conjugate of *A*.

void **ca_mat_conj_transpose**(*ca_mat_t* res, const *ca_mat_t* A, *ca_ctx_t* ctx)

Sets *res* to the conjugate transpose (Hermitian transpose) of *A*.

10.7.9 Arithmetic

void **ca_mat_neg**(*ca_mat_t* res, const *ca_mat_t* A, *ca_ctx_t* ctx)

Sets *res* to the negation of *A*.

void **ca_mat_add**(*ca_mat_t* res, const *ca_mat_t* A, const *ca_mat_t* B, *ca_ctx_t* ctx)

Sets *res* to the sum of *A* and *B*.

void **ca_mat_sub**(*ca_mat_t* res, const *ca_mat_t* A, const *ca_mat_t* B, *ca_ctx_t* ctx)

Sets *res* to the difference of *A* and *B*.

void **ca_mat_mul_classical**(*ca_mat_t* res, const *ca_mat_t* A, const *ca_mat_t* B, *ca_ctx_t* ctx)

```
void ca_mat_mul_same_nf(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_field_t K,
                       ca_ctx_t ctx)
```

```
void ca_mat_mul(ca_mat_t res, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)
```

Sets *res* to the matrix product of *A* and *B*. The *classical* version uses classical multiplication. The *same_nf* version assumes (not checked) that both *A* and *B* have coefficients in the same simple algebraic number field *K* or in $\mathbb{Q}$. The default version chooses an algorithm automatically.

```
void ca_mat_mul_si(ca_mat_t B, const ca_mat_t A, slong c, ca_ctx_t ctx)
```

```
void ca_mat_mul_fmpz(ca_mat_t B, const ca_mat_t A, const fmpz_t c, ca_ctx_t ctx)
```

```
void ca_mat_mul_fmpq(ca_mat_t B, const ca_mat_t A, const fmpq_t c, ca_ctx_t ctx)
```

```
void ca_mat_mul_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
```

Sets *B* to *A* multiplied by the scalar *c*.

```
void ca_mat_div_si(ca_mat_t B, const ca_mat_t A, slong c, ca_ctx_t ctx)
```

```
void ca_mat_div_fmpz(ca_mat_t B, const ca_mat_t A, const fmpz_t c, ca_ctx_t ctx)
```

```
void ca_mat_div_fmpq(ca_mat_t B, const ca_mat_t A, const fmpq_t c, ca_ctx_t ctx)
```

```
void ca_mat_div_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
```

Sets *B* to *A* divided by the scalar *c*.

```
void ca_mat_add_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
```

```
void ca_mat_sub_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
```

Sets *B* to *A* plus or minus the scalar *c* (interpreted as a diagonal matrix).

```
void ca_mat_addmul_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
```

```
void ca_mat_submul_ca(ca_mat_t B, const ca_mat_t A, const ca_t c, ca_ctx_t ctx)
```

Sets the matrix *B* to *B* plus (or minus) the matrix *A* multiplied by the scalar *c*.

10.7.10 Powers

```
void ca_mat_sqr(ca_mat_t B, const ca_mat_t A, ca_ctx_t ctx)
```

Sets *B* to the square of *A*.

```
void ca_mat_pow_ui_binexp(ca_mat_t B, const ca_mat_t A, ulong exp, ca_ctx_t ctx)
```

Sets *B* to *A* raised to the power *exp*, evaluated using binary exponentiation.

10.7.11 Polynomial evaluation

```
void _ca_mat_ca_poly_evaluate(ca_mat_t res, ca_srcptr poly, slong len, const ca_mat_t A,
                             ca_ctx_t ctx)
```

```
void ca_mat_ca_poly_evaluate(ca_mat_t res, const ca_poly_t poly, const ca_mat_t A, ca_ctx_t
                             ctx)
```

Sets *res* to $f(A)$ where f is the polynomial given by *poly* and *A* is a square matrix. Uses the Paterson-Stockmeyer algorithm.

10.7.12 Gaussian elimination and LU decomposition

truth_t **ca_mat_find_pivot**(*slong* *pivot_row, *ca_mat_t* mat, *slong* start_row, *slong* end_row, *slong* column, *ca_ctx_t* ctx)

Attempts to find a nonzero entry in *mat* with column index *column* and row index between *start_row* (inclusive) and *end_row* (exclusive).

If the return value is **T_TRUE**, such an element exists, and *pivot_row* is set to the row index. If the return value is **T_FALSE**, no such element exists (all entries in this part of the column are zero). If the return value is **T_UNKNOWN**, it is unknown whether such an element exists (zero certification failed).

This function is destructive: any elements that are nontrivially zero but can be certified zero will be overwritten by exact zeros.

int **ca_mat_lu_classical**(*slong* *rank, *slong* *P, *ca_mat_t* LU, const *ca_mat_t* A, int rank_check, *ca_ctx_t* ctx)

int **ca_mat_lu_recursive**(*slong* *rank, *slong* *P, *ca_mat_t* LU, const *ca_mat_t* A, int rank_check, *ca_ctx_t* ctx)

int **ca_mat_lu**(*slong* *rank, *slong* *P, *ca_mat_t* LU, const *ca_mat_t* A, int rank_check, *ca_ctx_t* ctx)

Computes a generalized LU decomposition $A = PLU$ of a given matrix *A*, writing the rank of *A* to *rank*.

If *A* is a nonsingular square matrix, *LU* will be set to a unit diagonal lower triangular matrix *L* and an upper triangular matrix *U* (the diagonal of *L* will not be stored explicitly).

If *A* is an arbitrary matrix of rank *r*, *U* will be in row echelon form having *r* nonzero rows, and *L* will be lower triangular but truncated to *r* columns, having implicit ones on the *r* first entries of the main diagonal. All other entries will be zero.

If a nonzero value for **rank_check** is passed, the function will abandon the output matrix in an undefined state and set the rank to 0 if *A* is detected to be rank-deficient.

The algorithm can fail if it fails to certify that a pivot element is zero or nonzero, in which case the correct rank cannot be determined. The return value is 1 on success and 0 on failure. On failure, the data in the output variables **rank**, **P** and **LU** will be meaningless.

The *classical* version uses iterative Gaussian elimination. The *recursive* version uses a block recursive algorithm to take advantage of fast matrix multiplication.

int **ca_mat_fflu**(*slong* *rank, *slong* *P, *ca_mat_t* LU, *ca_t* den, const *ca_mat_t* A, int rank_check, *ca_ctx_t* ctx)

Similar to **ca_mat_lu()**, but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to *den*. Note that despite being “fraction-free”, this algorithm may introduce fractions due to incomplete symbolic simplifications.

truth_t **ca_mat_nonsingular_lu**(*slong* *P, *ca_mat_t* LU, const *ca_mat_t* A, *ca_ctx_t* ctx)

Wrapper for **ca_mat_lu()**. If *A* can be proved to be invertible/nonsingular, returns **T_TRUE** and sets *P* and *LU* to a LU decomposition $A = PLU$. If *A* can be proved to be singular, returns **T_FALSE**. If *A* cannot be proved to be either singular or nonsingular, returns **T_UNKNOWN**. When the return value is **T_FALSE** or **T_UNKNOWN**, the LU factorization is not completed and the values of *P* and *LU* are arbitrary.

truth_t **ca_mat_nonsingular_fflu**(*slong* *P, *ca_mat_t* LU, *ca_t* den, const *ca_mat_t* A, *ca_ctx_t* ctx)

Wrapper for **ca_mat_fflu()**. Similar to **ca_mat_nonsingular_lu()**, but computes a fraction-free LU decomposition using the Bareiss algorithm. The denominator is written to *den*. Note that despite being “fraction-free”, this algorithm may introduce fractions due to incomplete symbolic simplifications.

10.7.13 Solving and inverse

`truth_t ca_mat_inv(ca_mat_t X, const ca_mat_t A, ca_ctx_t ctx)`

Determines if the square matrix A is nonsingular, and if successful, sets $X = A^{-1}$ and returns `T_TRUE`. Returns `T_FALSE` if A is singular, and `T_UNKNOWN` if the rank of A cannot be determined.

`truth_t ca_mat_nonsingular_solve_adjugate(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)`

`truth_t ca_mat_nonsingular_solve_fflu(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)`

`truth_t ca_mat_nonsingular_solve_lu(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)`

`truth_t ca_mat_nonsingular_solve(ca_mat_t X, const ca_mat_t A, const ca_mat_t B, ca_ctx_t ctx)`

Determines if the square matrix A is nonsingular, and if successful, solves $AX = B$ and returns `T_TRUE`. Returns `T_FALSE` if A is singular, and `T_UNKNOWN` if the rank of A cannot be determined.

`void ca_mat_solve_tril_classical(ca_mat_t X, const ca_mat_t L, const ca_mat_t B, int unit, ca_ctx_t ctx)`

`void ca_mat_solve_tril_recursive(ca_mat_t X, const ca_mat_t L, const ca_mat_t B, int unit, ca_ctx_t ctx)`

`void ca_mat_solve_tril(ca_mat_t X, const ca_mat_t L, const ca_mat_t B, int unit, ca_ctx_t ctx)`

`void ca_mat_solve_triu_classical(ca_mat_t X, const ca_mat_t U, const ca_mat_t B, int unit, ca_ctx_t ctx)`

`void ca_mat_solve_triu_recursive(ca_mat_t X, const ca_mat_t U, const ca_mat_t B, int unit, ca_ctx_t ctx)`

`void ca_mat_solve_triu(ca_mat_t X, const ca_mat_t U, const ca_mat_t B, int unit, ca_ctx_t ctx)`

Solves the lower triangular system $LX = B$ or the upper triangular system $UX = B$, respectively. It is assumed (not checked) that the diagonal entries are nonzero. If *unit* is set, the main diagonal of L or U is taken to consist of all ones, and in that case the actual entries on the diagonal are not read at all and can contain other data.

The *classical* versions perform the computations iteratively while the *recursive* versions perform the computations in a block recursive way to benefit from fast matrix multiplication. The default versions choose an algorithm automatically.

`void ca_mat_solve_fflu_precomp(ca_mat_t X, const slong *perm, const ca_mat_t A, const ca_t den, const ca_mat_t B, ca_ctx_t ctx)`

`void ca_mat_solve_lu_precomp(ca_mat_t X, const slong *P, const ca_mat_t LU, const ca_mat_t B, ca_ctx_t ctx)`

Solves $AX = B$ given the precomputed nonsingular LU decomposition $A = PLU$ or fraction-free LU decomposition with denominator *den*. The matrices X and B are allowed to be aliased with each other, but X is not allowed to be aliased with LU .

10.7.14 Rank and echelon form

`int ca_mat_rank(slong *rank, const ca_mat_t A, ca_ctx_t ctx)`

Computes the rank of the matrix A . If successful, returns 1 and writes the rank to *rank*. If unsuccessful, returns 0.

`int ca_mat_rref_fflu(slong *rank, ca_mat_t R, const ca_mat_t A, ca_ctx_t ctx)`

`int ca_mat_rref_lu(slong *rank, ca_mat_t R, const ca_mat_t A, ca_ctx_t ctx)`

```
int ca_mat_rref(slong *rank, ca_mat_t R, const ca_mat_t A, ca_ctx_t ctx)
```

Computes the reduced row echelon form (rref) of a given matrix. On success, sets *R* to the rref of *A*, writes the rank to *rank*, and returns 1. On failure to certify the correct rank, returns 0, leaving the data in *rank* and *R* meaningless.

The *fflu* version computes a fraction-free LU decomposition and then converts the output to rref form. The *lu* version computes a regular LU decomposition and then converts the output to rref form. The default version uses an automatic algorithm choice and may implement additional methods for special cases.

```
int ca_mat_right_kernel(ca_mat_t X, const ca_mat_t A, ca_ctx_t ctx)
```

Sets *X* to a basis of the right kernel (nullspace) of *A*. The output matrix *X* will be resized in-place to have a number of columns equal to the nullity of *A*. Returns 1 on success. On failure, returns 0 and leaves the data in *X* meaningless.

10.7.15 Determinant and trace

```
void ca_mat_trace(ca_t trace, const ca_mat_t mat, ca_ctx_t ctx)
```

Sets *trace* to the sum of the entries on the main diagonal of *mat*.

```
void ca_mat_det_berkowitz(ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

```
int ca_mat_det_lu(ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

```
int ca_mat_det_bareiss(ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

```
void ca_mat_det_cofactor(ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

```
void ca_mat_det(ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

Sets *det* to the determinant of the square matrix *A*. Various algorithms are available:

- The *berkowitz* version uses the division-free Berkowitz algorithm performing $O(n^4)$ operations. Since no zero tests are required, it is guaranteed to succeed.
- The *cofactor* version performs cofactor expansion. This is currently only supported for matrices up to size 4.
- The *lu* and *bareiss* versions use rational LU decomposition and fraction-free LU decomposition (Bareiss algorithm) respectively, requiring $O(n^3)$ operations. These algorithms can fail if zero certification fails (see *ca_mat_nonsingular_lu()*); they return 1 for success and 0 for failure. Note that the Bareiss algorithm, despite being “fraction-free”, may introduce fractions due to incomplete symbolic simplifications.

The default function chooses an algorithm automatically. It will, in addition, recognize trivially rational and integer matrices and evaluate those determinants using *fmpq_mat_t* or *fmpz_mat_t*.

The various algorithms can produce different symbolic forms of the same determinant. Which algorithm performs better depends strongly and sometimes unpredictably on the structure of the matrix.

```
void ca_mat_adjugate_cofactor(ca_mat_t adj, ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

```
void ca_mat_adjugate_charpoly(ca_mat_t adj, ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

```
void ca_mat_adjugate(ca_mat_t adj, ca_t det, const ca_mat_t A, ca_ctx_t ctx)
```

Sets *adj* to the adjugate matrix of *A* and *det* to the determinant of *A*, both computed simultaneously. The *cofactor* version uses cofactor expansion. The *charpoly* version computes and evaluates the characteristic polynomial. The default version uses an automatic algorithm choice.

10.7.16 Characteristic polynomial

```
void _ca_mat_charpoly_berkowitz(ca_ptr cp, const ca_mat_t mat, ca_ctx_t ctx)
void ca_mat_charpoly_berkowitz(ca_poly_t cp, const ca_mat_t mat, ca_ctx_t ctx)
int _ca_mat_charpoly_danilevsky(ca_ptr cp, const ca_mat_t mat, ca_ctx_t ctx)
int ca_mat_charpoly_danilevsky(ca_poly_t cp, const ca_mat_t mat, ca_ctx_t ctx)
void _ca_mat_charpoly(ca_ptr cp, const ca_mat_t mat, ca_ctx_t ctx)
void ca_mat_charpoly(ca_poly_t cp, const ca_mat_t mat, ca_ctx_t ctx)
```

Sets *poly* to the characteristic polynomial of *mat* which must be a square matrix. If the matrix has n rows, the underscore method requires space for $n + 1$ output coefficients.

The *berkowitz* version uses a division-free algorithm requiring $O(n^4)$ operations. The *danilevsky* version only performs $O(n^3)$ operations, but performs divisions and needs to check for zero which can fail. This version returns 1 on success and 0 on failure. The default version chooses an algorithm automatically.

```
int ca_mat_companion(ca_mat_t mat, const ca_poly_t poly, ca_ctx_t ctx)
```

Sets *mat* to the companion matrix of *poly*. This function verifies that the leading coefficient of *poly* is provably nonzero and that the output matrix has the right size, returning 1 on success. It returns 0 if the leading coefficient of *poly* cannot be proved nonzero or if the size of the output matrix does not match.

10.7.17 Eigenvalues and eigenvectors

```
int ca_mat_eigenvalues(ca_vec_t lambda, ulong *exp, const ca_mat_t mat, ca_ctx_t ctx)
```

Attempts to compute all complex eigenvalues of the given matrix *mat*. On success, returns 1 and sets *lambda* to the distinct eigenvalues with corresponding multiplicities in *exp*. The eigenvalues are returned in arbitrary order. On failure, returns 0 and leaves the values in *lambda* and *exp* arbitrary.

This function effectively computes the characteristic polynomial and then calls *ca_poly_roots*.

```
truth_t ca_mat_diagonalization(ca_mat_t D, ca_mat_t P, const ca_mat_t A, ca_ctx_t ctx)
```

Matrix diagonalization: attempts to compute a diagonal matrix D and an invertible matrix P such that $A = PDP^{-1}$. Returns `T_TRUE` if A is diagonalizable and the computation succeeds, `T_FALSE` if A is provably not diagonalizable, and `T_UNKNOWN` if it is unknown whether A is diagonalizable. If the return value is not `T_TRUE`, the values in D and P are arbitrary.

10.7.18 Jordan canonical form

```
int ca_mat_jordan_blocks(ca_vec_t lambda, slong *num_blocks, slong *block_lambda, slong
                        *block_size, const ca_mat_t A, ca_ctx_t ctx)
```

Computes the blocks of the Jordan canonical form of A . On success, returns 1 and sets *lambda* to the unique eigenvalues of A , sets *num_blocks* to the number of Jordan blocks, entry i of *block_lambda* to the index of the eigenvalue in Jordan block i , and entry i of *block_size* to the size of Jordan block i . On failure, returns 0, leaving arbitrary values in the output variables. The user should allocate space in *block_lambda* and *block_size* for up to n entries where n is the size of the matrix.

The Jordan form is unique up to the ordering of blocks, which is arbitrary.

```
void ca_mat_set_jordan_blocks(ca_mat_t mat, const ca_vec_t lambda, slong num_blocks, slong
                             *block_lambda, slong *block_size, ca_ctx_t ctx)
```

Sets *mat* to the concatenation of the Jordan blocks given in *lambda*, *num_blocks*, *block_lambda* and *block_size*. See *ca_mat_jordan_blocks()* for an explanation of these variables.

```
int ca_mat_jordan_transformation(ca_mat_t mat, const ca_vec_t lambda, slong num_blocks,
                                slong *block_lambda, slong *block_size, const ca_mat_t A,
                                ca_ctx_t ctx)
```

Given the precomputed Jordan block decomposition (λ , num_blocks , block_lambda , block_size) of the square matrix A , computes the corresponding transformation matrix P such that $A = PJP^{-1}$. On success, writes P to mat and returns 1. On failure, returns 0, leaving the value of mat arbitrary.

```
int ca_mat_jordan_form(ca_mat_t J, ca_mat_t P, const ca_mat_t A, ca_ctx_t ctx)
```

Computes the Jordan decomposition $A = PJP^{-1}$ of the given square matrix A . The user can pass NULL for the output variable P , in which case only J is computed. On success, returns 1. On failure, returns 0, leaving the values of J and P arbitrary.

This function is a convenience wrapper around `ca_mat_jordan_blocks()`, `ca_mat_set_jordan_blocks()` and `ca_mat_jordan_transformation()`. For computations with the Jordan decomposition, it is often better to use those methods directly since they give direct access to the spectrum and block structure.

10.7.19 Matrix functions

```
int ca_mat_exp(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)
```

Matrix exponential: given a square matrix A , sets res to e^A and returns 1 on success. If unsuccessful, returns 0, leaving the values in res arbitrary.

This function uses Jordan decomposition. The matrix exponential always exists, but computation can fail if computing the Jordan decomposition fails.

```
truth_t ca_mat_log(ca_mat_t res, const ca_mat_t A, ca_ctx_t ctx)
```

Matrix logarithm: given a square matrix A , sets res to a logarithm $\log(A)$ and returns `T_TRUE` on success. If A can be proved to have no logarithm, returns `T_FALSE`. If the existence of a logarithm cannot be proved, returns `T_UNKNOWN`.

This function uses the Jordan decomposition, and the branch of the matrix logarithm is defined by taking the principal values of the logarithms of all eigenvalues.

10.8 ca_ext.h – real and complex extension numbers

A `ca_ext_t` represents a fixed real or complex number a . The content of a `ca_ext_t` can be one of the following:

- An algebraic constant represented in canonical form by a `qqbar_t` instance (example: i , represented as the root of $x^2 + 1$ with positive imaginary part).
- A constant of the form $f(x_1, \dots, x_n)$ where f is a builtin symbolic function and $x_1, \dots, x_n$ are given `ca_t` instances.
- A builtin symbolic constant such as π . (This is just a special case of the above with a zero-length argument list.)
- (Not implemented): a user-defined constant or function defined by supplying a function pointer for Arb numerical evaluation to specified precision.

The `ca_ext_t` structure is heavy-weight object, not just meant to act as a node in a symbolic expression. It will cache various data to support repeated computation with this particular number, including its numerical enclosure and number field data in the case of algebraic numbers.

Extension numbers are used internally by the `ca_t` type to define the embeddings $\mathbb{Q}(a) \rightarrow \mathbb{C}$ of formal fields. The user does not normally need to create `ca_ext_t` instances directly; the intended way for the user to work with the extension number a is to create a `ca_t` representing the field element $1 \cdot a$. The underlying `ca_ext_t` may be accessed to determine symbolic and numerical properties of this number.

Since extension numbers may depend recursively on nontrivial fields for function arguments, `ca_ext_t` operations require a `ca_ctx_t` context object.

10.8.1 Type and macros

For all types, a `type_t` is defined as an array of length one of type `type_struct`, permitting a `type_t` to be passed by reference.

type `ca_ext_struct`

type `ca_ext_t`

An extension number object contains a header, a hash value, data (a `qqbar_t` instance and an Antic `nf_t` in the case of algebraic numbers, and a pointer to arguments in the case of a symbolic function), and a cached `acb_t` enclosure (in the case of a `qqbar_t`, the enclosure internal to that structure is used).

type `ca_ext_ptr`

Alias for `ca_ext_struct *`.

type `ca_ext_srcptr`

Alias for `const ca_ext_struct *`.

`CA_EXT_HEAD(x)`

Accesses the head (a `calcium_func_code`) of x . This is `CA_QQBar` if x represents an algebraic constant in canonical form, and `CA_Exp`, `CA_Pi`, etc. for symbolic functions and constants.

`CA_EXT_HASH(x)`

Accesses the hash value of x .

`CA_EXT_QQBAR(x)`

Assuming that x represents an algebraic constant in canonical form, accesses this `qqbar_t` object.

`CA_EXT_QQBAR_NF(x)`

Assuming that x represents an algebraic constant in canonical form, accesses the corresponding Antic number field `nf_t` object.

`CA_EXT_FUNC_ARGS(x)`

Assuming that x represents a symbolic constant or function, accesses the argument list (as a `ca_ptr`).

`CA_EXT_FUNC_NARGS(x)`

Assuming that x represents a symbolic constant or function, accesses the number of function arguments.

`CA_EXT_FUNC_ENCLOSURE(x)`

Assuming that x represents a symbolic constant or function, accesses the cached `acb_t` numerical enclosure.

`CA_EXT_FUNC_PREC(x)`

Assuming that x represents a symbolic constant or function, accesses the working precision of the cached numerical enclosure.

10.8.2 Memory management

void **ca_ext_init_qqbar**(*ca_ext_t* res, const *qqbar_t* x, *ca_ctx_t* ctx)

Initializes *res* and sets it to the algebraic constant *x*.

void **ca_ext_init_const**(*ca_ext_t* res, *calcium_func_code* func, *ca_ctx_t* ctx)

Initializes *res* and sets it to the constant defined by *func* (example: *func* = *CA_Pi* for $x = \pi$).

void **ca_ext_init_fx**(*ca_ext_t* res, *calcium_func_code* func, const *ca_t* x, *ca_ctx_t* ctx)

Initializes *res* and sets it to the univariate function value $f(x)$ where f is defined by *func* (example: *func* = *CA_Exp* for e^x).

void **ca_ext_init_fxy**(*ca_ext_t* res, *calcium_func_code* func, const *ca_t* x, const *ca_t* y, *ca_ctx_t* ctx)

Initializes *res* and sets it to the bivariate function value $f(x, y)$ where f is defined by *func* (example: *func* = *CA_Pow* for x^y).

void **ca_ext_init_fxn**(*ca_ext_t* res, *calcium_func_code* func, *ca_srcptr* x, *slong* nargs, *ca_ctx_t* ctx)

Initializes *res* and sets it to the multivariate function value $f(x_1, \dots, x_n)$ where f is defined by *func* and n is given by *nargs*.

void **ca_ext_init_set**(*ca_ext_t* res, const *ca_ext_t* x, *ca_ctx_t* ctx)

Initializes *res* and sets it to a copy of *x*.

void **ca_ext_clear**(*ca_ext_t* res, *ca_ctx_t* ctx)

Clears *res*.

10.8.3 Structure

slong **ca_ext_nargs**(const *ca_ext_t* x, *ca_ctx_t* ctx)

Returns the number of function arguments of *x*. The return value is 0 for any algebraic constant and for any built-in symbolic constant such as π .

void **ca_ext_get_arg**(*ca_t* res, const *ca_ext_t* x, *slong* i, *ca_ctx_t* ctx)

Sets *res* to argument *i* (indexed from zero) of *x*. This calls *flint_abort* if *i* is out of range.

ulong **ca_ext_hash**(const *ca_ext_t* x, *ca_ctx_t* ctx)

Returns a hash of the structural representation of *x*.

int **ca_ext_equal_repr**(const *ca_ext_t* x, const *ca_ext_t* y, *ca_ctx_t* ctx)

Tests *x* and *y* for structural equality, returning 0 (false) or 1 (true).

int **ca_ext_cmp_repr**(const *ca_ext_t* x, const *ca_ext_t* y, *ca_ctx_t* ctx)

Compares the representations of *x* and *y* in a canonical sort order, returning -1, 0 or 1. This only performs a structural comparison of the symbolic representations; the return value does not say anything meaningful about the numbers represented by *x* and *y*.

10.8.4 Input and output

void **ca_ext_print**(const *ca_ext_t* x, *ca_ctx_t* ctx)

Prints a description of *x* to standard output.

10.8.5 Numerical evaluation

void **ca_ext_get_acb_raw**(*acb_t* res, *ca_ext_t* x, *slong* prec, *ca_ctx_t* ctx)

Sets *res* to an enclosure of the numerical value of *x*. A working precision of *prec* bits is used for the evaluation, without adaptive refinement.

10.8.6 Cache

type **ca_ext_cache_struct**

type **ca_ext_cache_t**

Represents a set of structurally distinct *ca_ext_t* instances. This object contains an array of pointers to individual heap-allocated *ca_ext_struct* objects as well as a hash table for quick lookup.

void **ca_ext_cache_init**(*ca_ext_cache_t* cache, *ca_ctx_t* ctx)

Initializes *cache* for use.

void **ca_ext_cache_clear**(*ca_ext_cache_t* cache, *ca_ctx_t* ctx)

Clears *cache*, freeing the memory allocated internally.

ca_ext_ptr **ca_ext_cache_insert**(*ca_ext_cache_t* cache, const *ca_ext_t* x, *ca_ctx_t* ctx)

Adds *x* to *cache* without duplication. If a structurally identical instance already exists in *cache*, a pointer to that instance is returned. Otherwise, a copy of *x* is inserted into *cache* and a pointer to that new instance is returned.

10.9 `ca_field.h` – extension fields

A `ca_field_t` represents the parent field $K = \mathbb{Q}(a_1, \dots, a_n)$ of a `ca_t` element. A `ca_field_t` contains a list of pointers to `ca_ext_t` objects as well as a reduction ideal.

The user does not normally need to create `ca_field_t` objects manually: a `ca_ctx_t` context object manages a cache of fields automatically.

Internally, three types of field representation are used:

- The trivial field $\mathbb{Q}$.
- An Antic number field $\mathbb{Q}(a)$ where a is defined by a `qqbar_t`
- A generic field $\mathbb{Q}(a_1, \dots, a_n)$ where $n \geq 1$, and a_1 is not defined by a `qqbar_t` if $n = 1$.

The field type mainly affects the internal storage of the field elements; the distinction is mostly transparent to the external interface.

10.9.1 Type and macros

For all types, a `type_t` is defined as an array of length one of type `type_struct`, permitting a `type_t` to be passed by reference.

type `ca_field_struct`

type `ca_field_t`

Represents a formal field.

type `ca_field_ptr`

Alias for `ca_field_struct *`.

type `ca_field_srcptr`

Alias for `const ca_field_struct *`.

`CA_FIELD_LENGTH(K)`

Accesses the number n of extension numbers of K . This is 0 if $K = \mathbb{Q}$.

`CA_FIELD_EXT(K)`

Accesses the array of extension numbers as a `ca_ext_ptr`.

`CA_FIELD_EXT_ELEM(K, i)`

Accesses the extension number at position i (indexed from zero) as a `ca_ext_t`.

`CA_FIELD_HASH(K)`

Accesses the hash value of K .

`CA_FIELD_IS_QQ(K)`

Returns whether K is the trivial field $\mathbb{Q}$.

`CA_FIELD_IS_NF(K)`

Returns whether K represents an Antic number field $K = \mathbb{Q}(a)$ where a is represented by a `qqbar_t`.

`CA_FIELD_IS_GENERIC(K)`

Returns whether K represents a generic field.

`CA_FIELD_NF(K)`

Assuming that K represents an Antic number field $K = \mathbb{Q}(a)$, accesses the `nf_t` object representing this field.

CA_FIELD_NF_QQBAR(K)

Assuming that K represents an Antic number field $K = \mathbb{Q}(a)$, accesses the `qqbar_t` object representing a .

CA_FIELD_IDEAL(K)

Assuming that K represents a multivariate field, accesses the reduction ideal as a `fmpz_poly_t` array.

CA_FIELD_IDEAL_ELEM(K, i)

Assuming that K represents a multivariate field, accesses element i (indexed from zero) of the reduction ideal as a `fmpz_poly_t`.

CA_FIELD_IDEAL_LENGTH(K)

Assuming that K represents a multivariate field, accesses the number of polynomials in the reduction ideal.

CA_FIELD_MCTX(K, ctx)

Assuming that K represents a multivariate field, accesses the `fmpz_poly_ctx_t` context object for multivariate polynomial arithmetic on the internal representation of elements in this field.

10.9.2 Memory management

void **ca_field_init_qq**(`ca_field_t` K, `ca_ctx_t` ctx)

Initializes K to represent the trivial field $\mathbb{Q}$.

void **ca_field_init_nf**(`ca_field_t` K, const `qqbar_t` x, `ca_ctx_t` ctx)

Initializes K to represent the algebraic number field $\mathbb{Q}(x)$.

void **ca_field_init_const**(`ca_field_t` K, `calcium_func_code` func, `ca_ctx_t` ctx)

Initializes K to represent the field $\mathbb{Q}(x)$ where x is a builtin constant defined by `func` (example: `func = CA_Pi` for $x = \pi$).

void **ca_field_init_fx**(`ca_field_t` K, `calcium_func_code` func, const `ca_t` x, `ca_ctx_t` ctx)

Initializes K to represent the field $\mathbb{Q}(a)$ where $a = f(x)$, given a number x and a builtin univariate function `func` (example: `func = CA_Exp` for e^x).

void **ca_field_init_fxy**(`ca_field_t` K, `calcium_func_code` func, const `ca_t` x, const `ca_t` y, `ca_ctx_t` ctx)

Initializes K to represent the field $\mathbb{Q}(a, b)$ where $a = f(x, y)$.

void **ca_field_init_multi**(`ca_field_t` K, `slong` len, `ca_ctx_t` ctx)

Initializes K to represent a multivariate field $\mathbb{Q}(a_1, \dots, a_n)$ in n extension numbers. The extension numbers must subsequently be assigned one by one using `ca_field_set_ext()`.

void **ca_field_set_ext**(`ca_field_t` K, `slong` i, `ca_ext_srcptr` x_index, `ca_ctx_t` ctx)

Sets the extension number at position i (here indexed from 0) of K to the generator of the field with index `x_index` in `ctx`. (It is assumed that the generating field is a univariate field.)

This only inserts a shallow reference: the field at index `x_index` must be kept alive until K has been cleared.

void **ca_field_clear**(`ca_field_t` K, `ca_ctx_t` ctx)

Clears the field K . This does not clear the individual extension numbers, which are only held as references.

10.9.3 Input and output

void **ca_field_print**(const *ca_field_t* K, *ca_ctx_t* ctx)
 Prints a description of the field K to standard output.

10.9.4 Ideal

void **ca_field_build_ideal**(*ca_field_t* K, *ca_ctx_t* ctx)
 Given K with assigned extension numbers, builds the reduction ideal in-place.

void **ca_field_build_ideal_erf**(*ca_field_t* K, *ca_ctx_t* ctx)
 Builds relations for error functions present among the extension numbers in K . This heuristic adds relations that are consequences of the functional equations $\operatorname{erf}(x) = -\operatorname{erf}(-x)$, $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$, $\operatorname{erfi}(x) = -i \operatorname{erf}(ix)$.

10.9.5 Structure operations

int **ca_field_cmp**(const *ca_field_t* K1, const *ca_field_t* K2, *ca_ctx_t* ctx)
 Compares the field objects $K1$ and $K2$ in a canonical sort order, returning -1, 0 or 1. This only performs a lexicographic comparison of the representations of $K1$ and $K2$; the return value does not say anything meaningful about the relative structures of $K1$ and $K2$ as mathematical fields.

10.9.6 Cache

type **ca_field_cache_struct**

type **ca_field_cache_t**
 Represents a set of distinct *ca_field_t* instances. This object contains an array of pointers to individual heap-allocated *ca_field_struct* objects as well as a hash table for quick lookup.

void **ca_field_cache_init**(*ca_field_cache_t* cache, *ca_ctx_t* ctx)
 Initializes *cache* for use.

void **ca_field_cache_clear**(*ca_field_cache_t* cache, *ca_ctx_t* ctx)
 Clears *cache*, freeing the memory allocated internally. This does not clear the individual extension numbers, which are only held as references.

ca_field_ptr **ca_field_cache_insert_ext**(*ca_field_cache_t* cache, *ca_ext_struct* **x, *slong* len, *ca_ctx_t* ctx)

Adds the field defined by the length-*len* list of extension numbers x to *cache* without duplication. If such a field already exists in *cache*, a pointer to that instance is returned. Otherwise, a field with extension numbers x is inserted into *cache* and a pointer to that new instance is returned. Upon insertion of a new field, the reduction ideal is constructed via *ca_field_build_ideal()*.

10.10 `fexpr.h` – flat-packed symbolic expressions

This module supports working with symbolic expressions.

10.10.1 Introduction

Formally, a symbolic expression is either:

- An atom, being one of the following:
 - An integer, for example 0 or -34.
 - A symbol, for example `x`, `Mul`, `SomeUserNamedSymbol`. Symbols should be valid C identifiers (containing only the characters A-Z, a-z, 0-9, `_`, and not starting with a digit).
 - A string, for example `"Hello, world!"`. For the moment, we only consider ASCII strings, but there is no obstacle in principle to supporting UTF-8.
- A non-atomic expression, representing a function call $e_0(e_1, \dots, e_n)$ where $e_0, \dots, e_n$ are symbolic expressions.

The meaning of an expression depends on the interpretation of symbols in a given context. For example, with a standard interpretation (used within Calcium) of the symbols `Mul`, `Add` and `Neg`, the expression `Mul(3, Add(Neg(x), y))` encodes the formula $3 \cdot ((-x) + y)$ where `x` and `y` are symbolic variables. See *[fexpr_builtin.h – builtin symbols](#)* for documentation of builtin symbols.

Computing and embedding data

Symbolic expressions are usually not the best data structure to use directly for heavy-duty computations. Functions acting on symbolic expressions will typically convert to a dedicated data structure (e.g. polynomials) internally and (optionally) convert the final result back to a symbolic expression.

Symbolic expressions do not allow embedding arbitrary binary objects such as FLINT/Arb/Antic/Calcium types as atoms. This is done on purpose to make symbolic expressions easy to use as a data exchange format. To embed an object in an expression, one has the following options:

- Represent the object structurally using atoms supported natively by symbolic expressions (for example, an integer polynomial can be represented as a list of coefficients or as an arithmetic expression tree).
- Introduce a dummy symbol to represent the object, maintaining an external translation table mapping this symbol to the intended value.
- Encode the object using a string or symbol name. This is generally not recommended, as it requires parsing; properly used, symbolic expressions have the benefit of being able to represent the parsed structure.

Flat-packed representation

Symbolic expressions are often implemented using trees of pointers (often together with hash tables for uniqueness), requiring some form of memory management. The *`fexpr_t`* type, by contrast, stores a symbolic expression using a “flat-packed” representation without internal pointers. The expression data is just an array of words (`ulong`). The first word is a header encoding type information (whether the expression is a function call or an atom, and the type of the atom) and the total number of words in the expression. For atoms, the data is stored either in the header word itself (small integers and short symbols/strings) or in the following words. For function calls, the header is followed by the expressions $e_0, \dots, e_n$ packed contiguously in memory.

Pros:

- Memory management is trivial.
- Copying an expression is just copying an array of words.
- Comparing expressions for equality is just comparing arrays of words.
- Merging expressions is basically just concatenating arrays of words.
- Expression data can be shared freely in binary form between threads and even between machines (as long as all machines have the same word size and endianness).

Cons:

- Repeated instances of the same subexpression cannot share memory (a workaround is to introduce local dummy symbols for repeated subexpressions).
- Extracting a subexpression for modification generally requires making a complete copy of that subexpression (however, for read-only access to subexpressions, one can use “view” expressions which have zero overhead).
- Manipulating a part of an expression generally requires rebuilding the whole expression.
- Building an expression incrementally is typically $O(n^2)$. As a workaround, it is a good idea to work with balanced (low-depth) expressions and try to construct an expression in one go (for example, to create a sum, create a single `Add` expression with many arguments instead of chaining binary `Add` operations).

10.10.2 Types and macros

type `fexpr_struct`

type `fexpr_t`

An *fexpr_struct* consists of a pointer to an array of words along with a record of the number of allocated words.

An *fexpr_t* is defined as an array of length one of type *fexpr_struct*, permitting an *fexpr_t* to be passed by reference.

type `fexpr_ptr`

Alias for `fexpr_struct *`, used for arrays of expressions.

type `fexpr_srcptr`

Alias for `const fexpr_struct *`, used for arrays of expressions when passed as constant input to functions.

type `fexpr_vec_struct`

type `fexpr_vec_t`

A type representing a vector of expressions with managed length. The structure contains an *fexpr_ptr* *entries* for the entries, an integer *length* (the size of the vector), and an integer *alloc* (the number of allocated entries).

`fexpr_vec_entry`(vec, i)

Returns a pointer to entry *i* in the vector *vec*.

10.10.3 Memory management

void **fexpr_init**(*fexpr_t* expr)

Initializes *expr* for use. Its value is set to the atomic integer 0.

void **fexpr_clear**(*fexpr_t* expr)

Clears *expr*, freeing its allocated memory.

fexpr_ptr **fexpr_vec_init**(*slong* len)

Returns a heap-allocated vector of *len* initialized expressions.

void **_fexpr_vec_clear**(*fexpr_ptr* vec, *slong* len)

Clears the *len* expressions in *vec* and frees *vec* itself.

void **fexpr_fit_size**(*fexpr_t* expr, *slong* size)

Ensures that *expr* has room for *size* words.

void **fexpr_set**(*fexpr_t* res, const *fexpr_t* expr)

Sets *res* to the a copy of *expr*.

void **fexpr_swap**(*fexpr_t* a, *fexpr_t* b)

Swaps *a* and *b* efficiently.

10.10.4 Size information

slong **fexpr_depth**(const *fexpr_t* expr)

Returns the depth of *expr* as a symbolic expression tree.

slong **fexpr_num_leaves**(const *fexpr_t* expr)

Returns the number of leaves (atoms, counted with repetition) in the expression *expr*.

slong **fexpr_size**(const *fexpr_t* expr)

Returns the number of words in the internal representation of *expr*.

slong **fexpr_size_bytes**(const *fexpr_t* expr)

Returns the number of bytes in the internal representation of *expr*. The count excludes the size of the structure itself. Add `sizeof(fexpr_struct)` to get the size of the object as a whole.

slong **fexpr_allocated_bytes**(const *fexpr_t* expr)

Returns the number of allocated bytes in the internal representation of *expr*. The count excludes the size of the structure itself. Add `sizeof(fexpr_struct)` to get the size of the object as a whole.

10.10.5 Comparisons

int **fexpr_equal**(const *fexpr_t* a, const *fexpr_t* b)

Checks if *a* and *b* are exactly equal as expressions.

int **fexpr_equal_si**(const *fexpr_t* expr, *slong* c)

int **fexpr_equal_ui**(const *fexpr_t* expr, *ulong* c)

Checks if *expr* is an atomic integer exactly equal to *c*.

ulong **fexpr_hash**(const *fexpr_t* expr)

Returns a hash of the expression *expr*.

int **fexpr_cmp_fast**(const *fexpr_t* a, const *fexpr_t* b)

Compares *a* and *b* using an ordering based on the internal representation, returning -1, 0 or 1. This can be used, for instance, to maintain sorted arrays of expressions for binary search; the sort order has no mathematical significance.

10.10.6 Atoms

int **fexpr_is_integer**(const *fexpr_t* expr)

Returns whether *expr* is an atomic integer

int **fexpr_is_symbol**(const *fexpr_t* expr)

Returns whether *expr* is an atomic symbol.

int **fexpr_is_string**(const *fexpr_t* expr)

Returns whether *expr* is an atomic string.

int **fexpr_is_atom**(const *fexpr_t* expr)

Returns whether *expr* is any atom.

void **fexpr_zero**(*fexpr_t* res)

Sets *res* to the atomic integer 0.

int **fexpr_is_zero**(const *fexpr_t* expr)

Returns whether *expr* is the atomic integer 0.

int **fexpr_is_neg_integer**(const *fexpr_t* expr)

Returns whether *expr* is any negative atomic integer.

void **fexpr_set_si**(*fexpr_t* res, *slong* c)

void **fexpr_set_ui**(*fexpr_t* res, *ulong* c)

void **fexpr_set_fmpz**(*fexpr_t* res, const *fmpz_t* c)

Sets *res* to the atomic integer *c*.

int **fexpr_get_fmpz**(*fmpz_t* res, const *fexpr_t* expr)

Sets *res* to the atomic integer in *expr*. This aborts if *expr* is not an atomic integer.

void **fexpr_set_symbol_builtin**(*fexpr_t* res, *slong* id)

Sets *res* to the builtin symbol with internal index *id* (see *fexpr_builtin.h* – builtin symbols).

int **fexpr_is_builtin_symbol**(const *fexpr_t* expr, *slong* id)

Returns whether *expr* is the builtin symbol with index *id* (see *fexpr_builtin.h* – builtin symbols).

int **fexpr_is_any_builtin_symbol**(const *fexpr_t* expr)

Returns whether *expr* is any builtin symbol (see *fexpr_builtin.h* – builtin symbols).

void **fexpr_set_symbol_str**(*fexpr_t* res, const char *s)

Sets *res* to the symbol given by *s*.

char ***fexpr_get_symbol_str**(const *fexpr_t* expr)

Returns the symbol in *expr* as a string. The string must be freed with *flint_free()*. This aborts if *expr* is not an atomic symbol.

void **fexpr_set_string**(*fexpr_t* res, const char *s)

Sets *res* to the atomic string *s*.

char ***fexpr_get_string**(const *fexpr_t* expr)

Assuming that *expr* is an atomic string, returns a copy of this string. The string must be freed with *flint_free()*.

10.10.7 Input and output

void **fexpr_write**(*calcium_stream_t* stream, const *fexpr_t* expr)

Writes *expr* to *stream*.

void **fexpr_print**(const *fexpr_t* expr)

Prints *expr* to standard output.

char ***fexpr_get_str**(const *fexpr_t* expr)

Returns a string representation of *expr*. The string must be freed with *flint_free()*.

Warning: string literals appearing in expressions are currently not escaped.

10.10.8 LaTeX output

void **fexpr_write_latex**(*calcium_stream_t* stream, const *fexpr_t* expr, *ulong* flags)

Writes the LaTeX representation of *expr* to *stream*.

void **fexpr_print_latex**(const *fexpr_t* expr, *ulong* flags)

Prints the LaTeX representation of *expr* to standard output.

char ***fexpr_get_str_latex**(const *fexpr_t* expr, *ulong* flags)

Returns a string of the LaTeX representation of *expr*. The string must be freed with *flint_free()*.

Warning: string literals appearing in expressions are currently not escaped.

The *flags* parameter allows specifying options for LaTeX output. The following flags are supported:

FEXPR_LATEX_SMALL

Generate more compact formulas, most importantly by printing fractions inline as p/q instead of as $\frac{p}{q}$. This flag is automatically activated within subscripts and superscripts and in certain other parts of formulas.

FEXPR_LATEX_LOGIC

Use symbols for logical operators such as Not, And, Or, which by default are rendered as words for legibility.

10.10.9 Function call structure

slong **fexpr_nargs**(const *fexpr_t* expr)

Returns the number of arguments n in the function call $f(e_1, \dots, e_n)$ represented by *expr*. If *expr* is an atom, returns -1.

void **fexpr_func**(*fexpr_t* res, const *fexpr_t* expr)

Assuming that *expr* represents a function call $f(e_1, \dots, e_n)$, sets *res* to the function expression f .

void **fexpr_view_func**(*fexpr_t* view, const *fexpr_t* expr)

As *fexpr_func()*, but sets *view* to a shallow view instead of copying the expression. The variable *view* must not be initialized before use or cleared after use, and *expr* must not be modified or cleared as long as *view* is in use.

void **fexpr_arg**(*fexpr_t* res, const *fexpr_t* expr, *slong* i)

Assuming that *expr* represents a function call $f(e_1, \dots, e_n)$, sets *res* to the argument e_{i+1} . Note that indexing starts from 0. The index must be in bounds, with $0 \leq i < n$.

void **fexpr_view_arg**(*fexpr_t* view, const *fexpr_t* expr, *slong* i)

As *fexpr_arg()*, but sets *view* to a shallow view instead of copying the expression. The variable *view* must not be initialized before use or cleared after use, and *expr* must not be modified or cleared as long as *view* is in use.

void **fexpr_view_next**(*fexpr_t* view)

Assuming that *view* is a shallow view of a function argument e_i in a function call $f(e_1, \dots, e_n)$, sets *view* to a view of the next argument e_{i+1} . This function can be called when *view* refers to the last argument e_n , provided that *view* is not used afterwards. This function can also be called when *view* refers to the function f , in which case it will make *view* point to e_1 .

int **fexpr_is_builtin_call**(const *fexpr_t* expr, *slong* id)

Returns whether *expr* has the form $f(\dots)$ where f is a builtin function defined by *id* (see *fexpr_builtin.h* – *builtin symbols*).

int **fexpr_is_any_builtin_call**(const *fexpr_t* expr)

Returns whether *expr* has the form $f(\dots)$ where f is any builtin function (see *fexpr_builtin.h* – *builtin symbols*).

10.10.10 Composition

void **fexpr_call0**(*fexpr_t* res, const *fexpr_t* f)

void **fexpr_call1**(*fexpr_t* res, const *fexpr_t* f, const *fexpr_t* x1)

void **fexpr_call2**(*fexpr_t* res, const *fexpr_t* f, const *fexpr_t* x1, const *fexpr_t* x2)

void **fexpr_call3**(*fexpr_t* res, const *fexpr_t* f, const *fexpr_t* x1, const *fexpr_t* x2, const *fexpr_t* x3)

void **fexpr_call4**(*fexpr_t* res, const *fexpr_t* f, const *fexpr_t* x1, const *fexpr_t* x2, const *fexpr_t* x3, const *fexpr_t* x4)

void **fexpr_call_vec**(*fexpr_t* res, const *fexpr_t* f, *fexpr_srcptr* args, *slong* len)

Creates the function call $f(x_1, \dots, x_n)$. The *vec* version takes the arguments as an array *args* and n is given by *len*. Warning: aliasing between inputs and outputs is not implemented.

void **fexpr_call_builtin1**(*fexpr_t* res, *slong* f, const *fexpr_t* x1)

void **fexpr_call_builtin2**(*fexpr_t* res, *slong* f, const *fexpr_t* x1, const *fexpr_t* x2)

Creates the function call $f(x_1, \dots, x_n)$, where f defines a builtin symbol.

10.10.11 Subexpressions and replacement

int **fexpr_contains**(const *fexpr_t* expr, const *fexpr_t* x)

Returns whether *expr* contains the expression x as a subexpression (this includes the case where *expr* and x are equal).

int **fexpr_replace**(*fexpr_t* res, const *fexpr_t* expr, const *fexpr_t* x, const *fexpr_t* y)

Sets *res* to the expression *expr* with all occurrences of the subexpression x replaced by the expression y . Returns a boolean value indicating whether any replacements have been performed. Aliasing is allowed between *res* and *expr* but not between *res* and x or y .

int **fexpr_replace2**(*fexpr_t* res, const *fexpr_t* expr, const *fexpr_t* x1, const *fexpr_t* y1, const *fexpr_t* x2, const *fexpr_t* y2)

Like *fexpr_replace()*, but simultaneously replaces $x1$ by $y1$ and $x2$ by $y2$.

int **fexpr_replace_vec**(*fexpr_t* res, const *fexpr_t* expr, const *fexpr_vec_t* xs, const *fexpr_vec_t* ys)

Sets *res* to the expression *expr* with all occurrences of the subexpressions given by entries in *xs* replaced by the corresponding expressions in *ys*. It is required that *xs* and *ys* have the same length. Returns a boolean value indicating whether any replacements have been performed. Aliasing is allowed between *res* and *expr* but not between *res* and the entries of *xs* or *ys*.

10.10.12 Arithmetic expressions

void **fexpr_set_fmpq**(*fexpr_t* res, const *fmpq_t* x)

Sets *res* to the rational number *x*. This creates an atomic integer if the denominator of *x* is one, and otherwise creates a division expression.

void **fexpr_set_arf**(*fexpr_t* res, const *arf_t* x)

void **fexpr_set_d**(*fexpr_t* res, double x)

Sets *res* to an expression for the value of the floating-point number *x*. NaN is represented as Undefined. For a regular value, this creates an atomic integer or a rational fraction if the exponent is small, and otherwise creates an expression of the form `Mul(m, Pow(2, e))`.

void **fexpr_set_re_im_d**(*fexpr_t* res, double x, double y)

Sets *res* to an expression for the complex number with real part *x* and imaginary part *y*.

void **fexpr_neg**(*fexpr_t* res, const *fexpr_t* a)

void **fexpr_add**(*fexpr_t* res, const *fexpr_t* a, const *fexpr_t* b)

void **fexpr_sub**(*fexpr_t* res, const *fexpr_t* a, const *fexpr_t* b)

void **fexpr_mul**(*fexpr_t* res, const *fexpr_t* a, const *fexpr_t* b)

void **fexpr_div**(*fexpr_t* res, const *fexpr_t* a, const *fexpr_t* b)

void **fexpr_pow**(*fexpr_t* res, const *fexpr_t* a, const *fexpr_t* b)

Constructs an arithmetic expression with given arguments. No simplifications whatsoever are performed.

int **fexpr_is_arithmetic_operation**(const *fexpr_t* expr)

Returns whether *expr* is of the form $f(e_1, \dots, e_n)$ where *f* is one of the arithmetic operators Pos, Neg, Add, Sub, Mul, Div.

void **fexpr_arithmetic_nodes**(*fexpr_vec_t* nodes, const *fexpr_t* expr)

Sets *nodes* to a vector of subexpressions of *expr* such that *expr* is an arithmetic expression with *nodes* as leaves. More precisely, *expr* will be constructed out of nested application the arithmetic operators Pos, Neg, Add, Sub, Mul, Div with integers and expressions in *nodes* as leaves. Powers Pow with an atomic integer exponent are also allowed. The nodes are output without repetition but are not automatically sorted in a canonical order.

int **fexpr_get_fmpz_mpoly_q**(*fmpz_mpoly_q_t* res, const *fexpr_t* expr, const *fexpr_vec_t* vars, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to the expression *expr* as a formal rational function of the subexpressions in *vars*. The vector *vars* must have the same length as the number of variables specified in *ctx*. To build *vars* automatically for a given expression, **fexpr_arithmetic_nodes()** may be used.

Returns 1 on success and 0 on failure. Failure can occur for the following reasons:

- A subexpression is encountered that cannot be interpreted as an arithmetic operation and does not appear (exactly) in *vars*.
- Overflow (too many terms or too large exponent).
- Division by zero (a zero denominator is encountered).

It is important to note that this function views *expr* as a formal rational function with *vars* as formal indeterminates. It does thus not check for algebraic relations between *vars* and can implicitly divide by zero if *vars* are not algebraically independent.

void **fexpr_set_fmpz_mpoly**(*fexpr_t* res, const *fmpz_mpoly_t* poly, const *fexpr_vec_t* vars, const *fmpz_mpoly_ctx_t* ctx)

void **fexpr_set_fmpz_mpoly_q**(*fexpr_t* res, const *fmpz_mpoly_q_t* frac, const *fexpr_vec_t* vars, const *fmpz_mpoly_ctx_t* ctx)

Sets *res* to an expression for the multivariate polynomial *poly* (or rational function *frac*), using the expressions in *vars* as the variables. The length of *vars* must agree with the number of variables in *ctx*. If *NULL* is passed for *vars*, a default choice of symbols is used.

int **fexpr_expanded_normal_form**(*fexpr_t* res, const *fexpr_t* expr, *ulong* flags)

Sets *res* to *expr* converted to expanded normal form viewed as a formal rational function with its non-arithmetic subexpressions as terminal nodes. This function first computes nodes with *fexpr_arithmetic_nodes()*, sorts the nodes, evaluates to a rational function with *fexpr_get_fmpz_mpoly_q()*, and then converts back to an expression with *fexpr_set_fmpz_mpoly_q()*. Optional *flags* are reserved for future use.

10.10.13 Vectors

void **fexpr_vec_init**(*fexpr_vec_t* vec, *slong* len)

Initializes *vec* to a vector of length *len*. All entries are set to the atomic integer 0.

void **fexpr_vec_clear**(*fexpr_vec_t* vec)

Clears the vector *vec*.

void **fexpr_vec_print**(const *fexpr_vec_t* vec)

Prints *vec* to standard output.

void **fexpr_vec_swap**(*fexpr_vec_t* x, *fexpr_vec_t* y)

Swaps *x* and *y* efficiently.

void **fexpr_vec_fit_length**(*fexpr_vec_t* vec, *slong* len)

Ensures that *vec* has space for *len* entries.

void **fexpr_vec_set**(*fexpr_vec_t* dest, const *fexpr_vec_t* src)

Sets *dest* to a copy of *src*.

void **fexpr_vec_append**(*fexpr_vec_t* vec, const *fexpr_t* expr)

Appends *expr* to the end of the vector *vec*.

slong **fexpr_vec_insert_unique**(*fexpr_vec_t* vec, const *fexpr_t* expr)

Inserts *expr* without duplication into *vec*, returning its position. If this expression already exists, *vec* is unchanged. If this expression does not exist in *vec*, it is appended.

void **fexpr_vec_set_length**(*fexpr_vec_t* vec, *slong* len)

Sets the length of *vec* to *len*, truncating or zero-extending as needed.

void **_fexpr_vec_sort_fast**(*fexpr_ptr* vec, *slong* len)

Sorts the *len* entries in *vec* using the comparison function *fexpr_cmp_fast()*.

10.11 fexpr_builtin.h – builtin symbols

This module defines symbol names with a predefined meaning for use in symbolic expressions. These symbols will eventually all support LaTeX rendering as well as symbolic and numerical evaluation (where applicable).

By convention, all builtin symbol names are at least two characters long and start with an uppercase letter. Single-letter symbol names and symbol names beginning with a lowercase letter are reserved for variables.

For any builtin symbol name `Symbol`, the header file `fexpr_builtin.h` defines a C constant `FEXPR_Symbol` as an index to a builtin symbol table. The symbol will be documented as `Symbol` below.

10.11.1 C helper functions

slong `fexpr_builtin_lookup(const char *s)`

Returns the internal index used to encode the builtin symbol with name *s* in expressions. If *s* is not the name of a builtin symbol, returns -1.

`const char *fexpr_builtin_name(slong n)`

Returns a read-only pointer for a string giving the name of the builtin symbol with index *n*.

slong `fexpr_builtin_length(void)`

Returns the number of builtin symbols.

10.11.2 Variables and iteration

Expressions involving the following symbols have a special role in binding variables.

For

Generator expression. This is a syntactical construct which does not represent a mathematical object on its own. In general, `For(x, ...)` defines the symbol `x` as a locally bound variable in the scope of the parent expression. The following arguments ... specify an evaluation range, set or point. Their interpretation depends on the parent operator. The following cases are possible.

Case 1: `For(x, S)` specifies iteration or comprehension for `x` ranging over the values of the set `S`. This interpretation is used in operators that aggregate values over a set. The `For` expression may be followed by a filter predicate `P(x)` restricting the range to a subset of `S`. Examples:

`Set(f(x), For(x, S))` denotes $\{f(x) : x \in S\}$.

`Set(f(x), For(x, S), P(x))` denotes $\{f(x) : x \in S \text{ and } P(x)\}$.

`Sum(f(x), For(x, S))` denotes $\sum_{x \in S} f(x)$.

`Sum(f(x), For(x, S), P(x))` denotes $\sum_{x \in S, P(x)} f(x)$.

Case 2: `For(x, a, b)` specifies that `x` ranges between the endpoints `a` and `b` in the context of `Sum`, `Product`, `Integral`, and similar operators. Examples:

`Sum(f(n), For(n, a, b))` denotes $\sum_{n=a}^b f(n)$. The iteration is empty if $b < a$.

`Integral(f(x), For(x, a, b))` denotes $\int_a^b f(x)dx$, where the integral follows a straight-line path from *a* to *b*. Swapping *a* and *b* negates the value.

Case 3: `For(x, a)` specifies that `x` approaches the point `a` in the context of `Limit`-type operator, or differentiation with respect to `x` at the point `a` in the context of a `Derivative`-type operator. Examples:

`Derivative(f(x), For(x, a))` denotes $f'(a)$.

`Limit(f(x), For(x, a))` denotes $\lim_{x \rightarrow a} f(x)$.

Case 4: `For(x, a, n)` specifies differentiation with respect to `x` at the point `a` to order `n` in the context of a `Derivative`-type operator. Examples:

`Derivative(f(x), For(x, a, n))` denotes $f^{(n)}(a)$.

Where

`Where(f(x), Def(x, a))` defines the symbol `x` as an alias for the expression `a` and evaluates the expression `f(x)` with this bound value of `x`. This is equivalent to `f(a)`. This may be rendered as $f(x)$ where $x = a$.

`Where(f(x), Def(f(t), a))` defines the symbol `f` as a function mapping the dummy variable `t` to `a`.

`Where(Add(a, b), Def(Tuple(a, b), T))` is a destructuring assignment.

Def

Definition expression. This is a syntactical construct which does not represent a mathematical object on its own. The `Def` expression is used only within a `Where`-expression; see that documentation of that symbol for more examples.

`Def(x, a)` defines the symbol `x` as an alias for the expression `a`.

`Def(f(x, y, z), a)` defines the symbol `f` as a function of three variables. The dummy variables `x`, `y` and `z` may appear within the expression `a`.

Fun

`Fun(x, expr)` defines an anonymous univariate function mapping the symbol `x` to the expression `expr`. The symbol `x` becomes locally bound within this `Fun` expression.

Step

Repeat

10.11.3 Booleans and logic

Equal

`Equal(a, b)`, signifying $a = b$, is `True` if `a` and `b` represent the same object, and `False` otherwise. This operator can be called with any number of arguments, in which case it evaluates whether all arguments are equal.

NotEqual

`NotEqual(a, b)`, signifying $a \neq b$, is equivalent to `Not(Equal(a, b))`.

Same

`Same(a, b)` gives `a` (or equivalently `b`) if `a` and `b` represent the same object, and `Undefined` otherwise. This can be used to assert or emphasize that two expressions represent the same value within a formula. This operator can be called with any number of arguments, in which case it asserts that all arguments are equal.

True

`True` is a logical constant.

False

`False` is a logical constant.

Not

`Not(x)` is the logical negation of `x`.

And

`And(x, y)` is the logical AND of `x` and `y`. This function can be called with any number of arguments.

Or

`Or(x, y)` is the logical OR of x and y . This function can be called with any number of arguments.

Equivalent

`Equivalent(x, y)` denotes the logical equivalence $x \Leftrightarrow y$. Semantically, this is the same as `Equal` called with logical arguments.

Implies

`Implies(x, y)` denotes the logical implication $x \implies y$.

Exists

Existence quantifier.

`Exists(f(x), For(x, S))` denotes $f(x)$ for some $x \in S$.

`Exists(f(x), For(x, S), P(x))` denotes $f(x)$ for some $x \in S$ with $P(x)$.

All

Universal quantifier.

`All(f(x), For(x, S))` denotes $f(x)$ for all $x \in S$.

`All(f(x), For(x, S), P(x))` denotes $f(x)$ for all $x \in S$ with $P(x)$.

Cases

`Cases(Case(f(x), P(x)), Case(g(x), Otherwise))` denotes:

$$\begin{cases} f(x), & P(x) \\ g(x), & \text{otherwise} \end{cases}$$

`Cases(Case(f(x), P(x)), Case(g(x), Q(x)), Case(h(x), Otherwise))` denotes:

$$\begin{cases} f(x), & P(x) \\ g(x), & Q(x) \\ h(x), & \text{otherwise} \end{cases}$$

If both $P(x)$ and $Q(x)$ are true simultaneously, no ordering is implied; it is assumed that $f(x)$ and $g(x)$ give the same value for any such x . More generally, this operator can be called with any number of case distinctions.

If the *Otherwise* case is omitted, the result is undefined if neither predicate is true.

Case

See `Cases`.

Otherwise

See `Cases`.

10.11.4 Tuples, lists and sets

Tuple

List

Set

Item

Element

NotElement

EqualAndElement

Length

Cardinality

Concatenation

Union

Intersection

SetMinus

Subset

SubsetEqual

CartesianProduct

CartesianPower

Subsets

Subsets(S) is the power set $\mathcal{P}(S)$ comprising all subsets of the set **S**.

Sets

Sets is the class **Sets** of all sets.

Tuples

Tuples is the class of all tuples.

Tuples(S) is the set of all tuples with elements in the set **S**.

Tuples(S, n) is the set of all length-**n** tuples with elements in the set **S**.

10.11.5 Numbers and arithmetic

Undefined

Undefined

Undefined is the special value **u** (undefined).

Particular numbers

Pi

Pi is the constant π .

NumberI

NumberI is the imaginary unit i . The verbose name leaves **i** and **I** to be used as a variable names.

NumberE

NumberE is the base of the natural logarithm e . The verbose name leaves **e** and **E** to be used as a variable names.

GoldenRatio

GoldenRatio is the golden ratio φ .

Euler

Euler is Euler's constant γ .

CatalanConstant

CatalanConstant is Catalan's constant G .

KhinchinConstant

KhinchinConstant is Khinchin's constant K .

GlaisherConstant

GlaisherConstant is Glaisher's constant A .

RootOfUnity

RootOfUnity(n) is the principal complex n -th root of unity $\zeta_n = e^{2\pi i/n}$.

RootOfUnity(n , k) is the complex n -th root of unity ζ_n^k .

Number constructors

Remark: the rational number with numerator p and denominator q can be constructed as Div(p , q).

Decimal

Decimal(str) gives the rational number specified by the string str in ordinary decimal floating-point notation (for example $-3.25e-725$).

AlgebraicNumberSerialized**PolynomialRootIndexed****PolynomialRootNearest****Enclosure****Approximation****Guess****Unknown****Arithmetic operations****Pos****Neg****Add****Sub****Mul****Div****Pow****Sqrt****Root**

Inequalities

Less

LessEqual

Greater

GreaterEqual

EqualNearestDecimal

Sets of numbers

NN

NN is the set of natural numbers (including 0), $\mathbb{N}$.

ZZ

ZZ is the set of integers, $\mathbb{Z}$.

QQ

QQ is the set of rational numbers, $\mathbb{Q}$.

RR

RR is the set of real numbers, $\mathbb{R}$.

CC

CC is the set of complex numbers, $\mathbb{C}$.

Primes

Primes is the set of positive prime numbers, $\mathbb{P}$

IntegersGreaterEqual

IntegersGreaterEqual(*x*), given an extended real number *x*, gives the set $\mathbb{Z}_{\geq x}$ of integers greater than or equal to *x*.

IntegersLessEqual

IntegersLessEqual(*x*), given an extended real number *x*, gives the set $\mathbb{Z}_{\leq x}$ of integers less than or equal to *x*.

Range

Range(*a*, *b*), given integers *a* and *b*, gives the set $\{a, a + 1, \dots, b\}$ of integers between *a* and *b*. This is the empty set if *a* is greater than *b*.

AlgebraicNumbers

The set of complex algebraic numbers $\overline{\mathbb{Q}}$.

RealAlgebraicNumbers

The set of real algebraic numbers $\overline{\mathbb{Q}}_{\mathbb{R}}$.

Interval

Interval(*a*, *b*), given extended real numbers *a* and *b*, gives the closed interval $[a, b]$.

OpenInterval

OpenInterval(*a*, *b*), given extended real numbers *a* and *b*, gives the open interval (a, b) .

ClosedOpenInterval

ClosedOpenInterval(*a*, *b*), given extended real numbers *a* and *b*, gives the closed-open interval $[a, b)$.

OpenClosedInterval

`OpenClosedInterval(a, b)`, given extended real numbers a and b , gives the closed-open interval $(a, b]$.

RealBall

`RealBall(m, r)`, given a real number m and an extended real number r , gives the the closed real ball $[m \pm r]$ with center m and radius r .

OpenRealBall

`OpenRealBall(m, r)`, given a real number m and an extended real number r , gives the the open real ball $(m \pm r)$ with center m and radius r .

OpenComplexDisk

`OpenComplexDisk(m, r)`, given a complex number m and an extended real number r , gives the open complex disk $D(m, r)$ with center m and radius r .

ClosedComplexDisk

`ClosedComplexDisk(m, r)`, given a complex number m and a real number r , gives the closed complex disk $\overline{D}(m, r)$ with center m and radius r .

UpperHalfPlane

`UpperHalfPlane` is the set $\mathbb{H}$ of complex numbers with positive imaginary part.

UnitCircle**BernsteinEllipse****Lattice****Infinites and extended numbers****Infinity**

`Infinity` is the positive signed infinity ∞ .

UnsignedInfinity

`UnsignedInfinity` is the unsigned infinity $\tilde{\infty}$.

RealSignedInfinites

`RealSignedInfinites` is the set of real signed infinities $\{+\infty, -\infty\}$.

ComplexSignedInfinites

`ComplexSignedInfinites` is the set of complex signed infinities $\{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\}$.

RealInfinites

`RealInfinites` is the set of real infinities (signed and unsigned) $\{+\infty, -\infty\} \cup \{\tilde{\infty}\}$.

ComplexInfinites

`ComplexInfinites` is the set of complex infinities (signed and unsigned) $\{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\} \cup \{\tilde{\infty}\}$.

ExtendedRealNumbers

`ExtendedRealNumbers` is the set of extended real numbers $\mathbb{R} \cup \{+\infty, -\infty\}$.

ProjectiveRealNumbers

`ProjectiveRealNumbers` is the set of projectively extended real numbers $\mathbb{R} \cup \{\tilde{\infty}\}$.

SignExtendedComplexNumbers

`SignExtendedComplexNumbers` is the set of complex numbers extended with signed infinities $\mathbb{C} \cup \{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\}$.

ProjectiveComplexNumbers

ProjectiveComplexNumbers is the set of projectively extended complex numbers (also known as the Riemann sphere) $\mathbb{C} \cup \{\infty\}$.

RealSingularityClosure

RealSingularityClosure is the Calcium singularity closure for real functions, encompassing real numbers, signed infinities, unsigned infinity, and *undefined* (u). This set is defined as $\mathbb{R}_{\text{Sing}} = \mathbb{R} \cup \{+\infty, -\infty\} \cup \{\infty\} \cup \{u\}$.

ComplexSingularityClosure

ComplexSingularityClosure is the Calcium singularity closure for complex functions, encompassing complex numbers, signed infinities, unsigned infinity, and *undefined* (u). This set is defined as $\mathbb{C}_{\text{Sing}} = \mathbb{C} \cup \{e^{i\theta} \cdot \infty : \theta \in \mathbb{R}\} \cup \{\infty\} \cup \{u\}$.

10.11.6 Operators and calculus

Sums and products

Sum

Product

PrimeSum

PrimeProduct

DivisorSum

DivisorProduct

Solutions and zeros

Zeros

UniqueZero

Solutions

UniqueSolution

Extreme values

Supremum

Infimum

Minimum

Maximum

ArgMin

ArgMax

ArgMinUnique

ArgMaxUnique

Limits

Limit

SequenceLimit

RealLimit

LeftLimit

RightLimit

ComplexLimit

MeromorphicLimit

SequenceLimitInferior

SequenceLimitSuperior

AsymptoticTo

Derivatives

Derivative

RealDerivative

ComplexDerivative

ComplexBranchDerivative

MeromorphicDerivative

Integrals

Integral

Complex analysis

Path

CurvePath

Poles

IsHolomorphicOn

IsMeromorphicOn

Residue

ComplexZeroMultiplicity

AnalyticContinuation

10.11.7 Matrices and linear algebra

Matrix

Row

Column

RowMatrix

ColumnMatrix

DiagonalMatrix

Matrix2x2

ZeroMatrix

IdentityMatrix

Det

Spectrum

SingularValues

Matrices

SL2Z

PSL2Z

SpecialLinearGroup

GeneralLinearGroup

HilbertMatrix

10.11.8 Polynomials, series and rings

Pol

Ser

Polynomial

Coefficient

PolynomialDegree

Polynomials

PolynomialFractions

FormalPowerSeries

FormalLaurentSeries

FormalPuisseuxSeries

Zero

One

Characteristic

Rings

CommutativeRings

Fields

QuotientRing

FiniteField

EqualQSeriesEllipsis

IndefiniteIntegralEqual

QSeriesCoefficient

Call

CallIndeterminate

10.11.9 Special functions

Number parts and step functions

Abs

Sign

Re

Im

Arg

Conjugate

Csgn

RealAbs

Max

Min

Floor

Ceil

KroneckerDelta

Primes and divisibility

IsOdd

IsEven

CongruentMod

Divides

Mod
GCD
LCM
XGCD
IsPrime
Prime
PrimePi
DivisorSigma
MoebiusMu
EulerPhi
DiscreteLog
LegendreSymbol
JacobiSymbol
KroneckerSymbol
SquaresR
LiouvilleLambda

Elementary functions

Exp
Log
Sin
Cos
Tan
Cot
Sec
Csc
Sinh
Cosh
Tanh
Coth
Sech
Csch
Asin

Acos

Atan

Acot

Asec

Acsc

Asinh

Acosh

Atanh

Acoth

Asech

Acsch

Atan2

Sinc

LambertW

Combinatorial functions

SloaneA

SymmetricPolynomial

Cyclotomic

Fibonacci

BernoulliB

BernoulliPolynomial

StirlingCycle

StirlingS1

StirlingS2

EulerE

EulerPolynomial

BellNumber

PartitionsP

LandauG

Gamma function and factorials

Factorial

Binomial

Gamma

LogGamma

DoubleFactorial

RisingFactorial

FallingFactorial

HarmonicNumber

DigammaFunction

DigammaFunctionZero

BetaFunction

BarnesG

LogBarnesG

StirlingSeriesRemainder

LogBarnesGRemainder

Orthogonal polynomials

ChebyshevT

ChebyshevU

LegendreP

JacobiP

HermiteH

LaguerreL

GegenbauerC

SphericalHarmonicY

LegendrePolynomialZero

GaussLegendreWeight

Exponential integrals

Erf

Erfc

Erfi

UpperGamma

LowerGamma

IncompleteBeta

IncompleteBetaRegularized

LogIntegral

ExpIntegralE

ExpIntegralEi

SinIntegral

SinhIntegral

CosIntegral

CoshIntegral

FresnelC

FresnelS

Bessel and Airy functions

AiryAi

AiryBi

AiryAiZero

AiryBiZero

BesselJ

BesselI

BesselY

BesselK

HankelH1

HankelH2

BesselJZero

BesselYZero

CoulombF

CoulombG

CoulombH

CoulombC

CoulombSigma

Hypergeometric functions

Hypergeometric0F1

Hypergeometric1F1

Hypergeometric1F2

Hypergeometric2F1

Hypergeometric2F2

Hypergeometric2F0

Hypergeometric3F2

HypergeometricU

HypergeometricUStar

HypergeometricUStarRemainder

Hypergeometric0F1Regularized

Hypergeometric1F1Regularized

Hypergeometric1F2Regularized

Hypergeometric2F1Regularized

Hypergeometric2F2Regularized

Hypergeometric3F2Regularized

Zeta and L-functions

RiemannZeta

RiemannZetaZero

RiemannHypothesis

RiemannXi

HurwitzZeta

LerchPhi

PolyLog

MultiZetaValue

DirichletL

DirichletLZero

DirichletLambda

DirichletCharacter

DirichletGroup

PrimitiveDirichletCharacters

GeneralizedRiemannHypothesis

ConreyGenerator

GeneralizedBernoulliB

StieltjesGamma

KeiperLiLambda

GaussSum

Elliptic integrals

AGM

AGMSequence

EllipticK

EllipticE

EllipticPi

IncompleteEllipticF

IncompleteEllipticE

IncompleteEllipticPi

CarlsonRF

CarlsonRG

CarlsonRJ

CarlsonRD

CarlsonRC

CarlsonHypergeometricR

CarlsonHypergeometricT

Elliptic, theta and modular functions

JacobiTheta

JacobiThetaQ

DedekindEta

ModularJ

ModularLambda
EisensteinG
EisensteinE
DedekindSum
WeierstrassP
WeierstrassZeta
WeierstrassSigma
EllipticRootE
HilbertClassPolynomial
EulerQSeries
DedekindEtaEpsilon
ModularGroupAction
ModularGroupFundamentalDomain
ModularLambdaFundamentalDomain
PrimitiveReducedPositiveIntegralBinaryQuadraticForms
JacobiThetaEpsilon
JacobiThetaPermutation

Nonsemantic markup

Ellipsis

`Ellipsis` renders as ... in LaTeX. It can be used to indicate missing function arguments for display purposes, but it has no predefined builtin semantics.

Parentheses

`Parentheses(x)` semantically represents `x`, but renders with parentheses $((x))$ when converted to LaTeX.

Brackets

`Brackets(x)` semantically represents `x`, but renders with brackets $([x])$ when converted to LaTeX.

Braces

`Braces(x)` semantically represents `x`, but renders with braces $(\{x\})$ when converted to LaTeX.

AngleBrackets

`AngleBrackets(x)` semantically represents `x`, but renders with angle brackets $(\langle x \rangle)$ when converted to LaTeX.

Logic

`Logic(x)` semantically represents `x`, but forces logical expressions within `x` to be rendered using symbols instead of text.

ShowExpandedNormalForm

`ShowExpandedNormalForm(x)` semantically represents `x`, but displays the expanded normal form of the expression instead of rendering the expression verbatim. Warning: this triggers a nontrivial (potentially very expensive) computation.

Subscript

FINITE FIELDS

11.1 fq.h – finite fields

We represent an element of the finite field $\mathbf{F}_{p^n} \cong \mathbf{F}_p[X]/(f(X))$, where $f(X) \in \mathbf{F}_p[X]$ is a monic, irreducible polynomial of degree n , as a polynomial in $\mathbf{F}_p[X]$ of degree less than n . The underlying data structure is an *fmpz_poly_t*.

The default choice for $f(X)$ is the Conway polynomial for the pair (p, n) , enabled by Frank Lübeck's data base of Conway polynomials using the *_nmod_poly_conway()* function. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

11.1.1 Types, macros and constants

type **fq_ctx_struct**

type **fq_ctx_t**

type **fq_struct**

type **fq_t**

11.1.2 Context Management

void **fq_ctx_init**(*fq_ctx_t* ctx, const *fmpz_t* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name **var** for the generator. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.

Assumes that p is a prime.

Assumes that the string **var** is a null-terminated string of length at least one.

int **_fq_ctx_init_conway**(*fq_ctx_t* ctx, const *fmpz_t* p, *slong* d, const char *var)

Attempts to initialise the context for prime p and extension degree d , with name **var** for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that p is a prime.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_ctx_init_conway**(*fq_ctx_t* ctx, const *fmpz_t* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name **var** for the generator using a Conway polynomial for the modulus.

Assumes that p is a prime.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_ctx_init_modulus**(*fq_ctx_t* ctx, const *fmpz_mod_poly_t* modulus, const *fmpz_mod_ctx_t* ctxp, const char *var)

Initialises the context for given **modulus** with name **var** for the generator.

Assumes that **modulus** is an irreducible polynomial over the finite field $\mathbf{F}_p$ in **ctxp**.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_ctx_init_randtest**(*fq_ctx_t* ctx, *flint_rand_t* state, int type)

Initialises **ctx** to a random finite field, where the prime and degree is set according to **type**. To see what prime and degrees may be output, see **type** in `_nmod_poly_conway_rand()`.

void **fq_ctx_init_randtest_reducible**(*fq_ctx_t* ctx, *flint_rand_t* state, int type)

Initializes **ctx** to a random extension of a prime field, where the prime and degree is set according to **type**. If **type** is 0 the prime and degree may be large, else if **type** is 1 the degree is small but the prime may be large, else if **type** is 2 the prime is small but the degree may be large, else if **type** is 3 both prime and degree are small.

The modulus may or may not be irreducible.

void **fq_ctx_clear**(*fq_ctx_t* ctx)

Clears all memory that has been allocated as part of the context.

const *fmpz_mod_poly_struct* ***fq_ctx_modulus**(const *fq_ctx_t* ctx)

Returns a pointer to the modulus in the context.

slong **fq_ctx_degree**(const *fq_ctx_t* ctx)

Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

const *fmpz* ***fq_ctx_prime**(const *fq_ctx_t* ctx)

Returns a pointer to the prime p in the context.

void **fq_ctx_order**(*fmpz_t* f, const *fq_ctx_t* ctx)

Sets f to be the size of the finite field.

int **fq_ctx_fprint**(FILE *file, const *fq_ctx_t* ctx)

Prints the context information to **file**. Returns 1 for a success and a negative number for an error.

void **fq_ctx_print**(const *fq_ctx_t* ctx)

Prints the context information to **stdout**.

11.1.3 Memory management

void **fq_init**(*fq_t* rop, const *fq_ctx_t* ctx)

Initialises the element **rop**, setting its value to 0.

void **fq_init2**(*fq_t* rop, const *fq_ctx_t* ctx)

Initialises **poly** with at least enough space for it to be an element of **ctx** and sets it to 0.

void **fq_clear**(*fq_t* rop, const *fq_ctx_t* ctx)

Clears the element **rop**.

void **_fq_sparse_reduce**(*fmpz* *R, *slong* lenR, const *fq_ctx_t* ctx)

Reduces (R, **lenR**) modulo the polynomial f given by the modulus of **ctx**.

void **_fq_dense_reduce**(*fmpz* *R, *slong* lenR, const *fq_ctx_t* ctx)

Reduces (R, lenR) modulo the polynomial f given by the modulus of ctx using Newton division.

void **_fq_reduce**(*fmpz* *r, *slong* lenR, const *fq_ctx_t* ctx)

Reduces (R, lenR) modulo the polynomial f given by the modulus of ctx. Does either sparse or dense reduction based on ctx->sparse_modulus.

void **fq_reduce**(*fq_t* rop, const *fq_ctx_t* ctx)

Reduces the polynomial rop as an element of $\mathbf{F}_p[X]/(f(X))$.

11.1.4 Basic arithmetic

void **fq_add**(*fq_t* rop, const *fq_t* op1, const *fq_t* op2, const *fq_ctx_t* ctx)

Sets rop to the sum of op1 and op2.

void **fq_sub**(*fq_t* rop, const *fq_t* op1, const *fq_t* op2, const *fq_ctx_t* ctx)

Sets rop to the difference of op1 and op2.

void **fq_sub_one**(*fq_t* rop, const *fq_t* op1, const *fq_ctx_t* ctx)

Sets rop to the difference of op1 and 1.

void **fq_neg**(*fq_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)

Sets rop to the negative of op.

void **fq_mul**(*fq_t* rop, const *fq_t* op1, const *fq_t* op2, const *fq_ctx_t* ctx)

Sets rop to the product of op1 and op2, reducing the output in the given context.

void **fq_mul_fmpz**(*fq_t* rop, const *fq_t* op, const *fmpz_t* x, const *fq_ctx_t* ctx)

Sets rop to the product of op and x, reducing the output in the given context.

void **fq_mul_si**(*fq_t* rop, const *fq_t* op, *slong* x, const *fq_ctx_t* ctx)

Sets rop to the product of op and x, reducing the output in the given context.

void **fq_mul_ui**(*fq_t* rop, const *fq_t* op, *ulong* x, const *fq_ctx_t* ctx)

Sets rop to the product of op and x, reducing the output in the given context.

void **fq_sqr**(*fq_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)

Sets rop to the square of op, reducing the output in the given context.

void **fq_div**(*fq_t* rop, const *fq_t* op1, const *fq_t* op2, const *fq_ctx_t* ctx)

Sets rop to the quotient of op1 and op2, reducing the output in the given context.

void **_fq_inv**(*fmpz* *rop, const *fmpz* *op, *slong* len, const *fq_ctx_t* ctx)

Sets (rop, d) to the inverse of the non-zero element (op, len).

void **fq_inv**(*fq_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)

Sets rop to the inverse of the non-zero element op.

void **fq_gcdinv**(*fq_t* f, *fq_t* inv, const *fq_t* op, const *fq_ctx_t* ctx)

Sets inv to be the inverse of op modulo the modulus of ctx. If op is not invertible, then f is set to a factor of the modulus; otherwise, it is set to one.

void **_fq_pow**(*fmpz* *rop, const *fmpz* *op, *slong* len, const *fmpz_t* e, const *fq_ctx_t* ctx)

Sets (rop, 2*d-1) to (op, len) raised to the power e, reduced modulo $f(X)$, the modulus of ctx.

Assumes that $e \geq 0$ and that len is positive and at most d.

Although we require that rop provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree d.

Does not support aliasing.

void **fq_pow**(*fq_t* rop, const *fq_t* op, const *fmpz_t* e, const *fq_ctx_t* ctx)

Sets **rop** the **op** raised to the power *e*.

Currently assumes that $e \geq 0$.

Note that for any input **op**, **rop** is set to 1 whenever $e = 0$.

void **fq_pow_ui**(*fq_t* rop, const *fq_t* op, const *ulong* e, const *fq_ctx_t* ctx)

Sets **rop** the **op** raised to the power *e*.

Currently assumes that $e \geq 0$.

Note that for any input **op**, **rop** is set to 1 whenever $e = 0$.

11.1.5 Roots

int **fq_sqrt**(*fq_t* rop, const *fq_t* op1, const *fq_ctx_t* ctx)

Sets **rop** to the square root of **op1** if it is a square, and return 1, otherwise return 0.

void **fq_pth_root**(*fq_t* rop, const *fq_t* op1, const *fq_ctx_t* ctx)

Sets **rop** to a p^{th} root root of **op1**. Currently, this computes the root by raising **op1** to p^{d-1} where d is the degree of the extension.

int **fq_is_square**(const *fq_t* op, const *fq_ctx_t* ctx)

Return 1 if **op** is a square.

11.1.6 Output

int **fq_fprint_pretty**(FILE *file, const *fq_t* op, const *fq_ctx_t* ctx)

Prints a pretty representation of **op** to **file**.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int **fq_print_pretty**(const *fq_t* op, const *fq_ctx_t* ctx)

Prints a pretty representation of **op** to **stdout**.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int **fq_fprint**(FILE *file, const *fq_t* op, const *fq_ctx_t* ctx)

Prints a representation of **op** to **file**.

For further details on the representation used, see `fmpz_mod_poly_fprint()`.

void **fq_print**(const *fq_t* op, const *fq_ctx_t* ctx)

Prints a representation of **op** to **stdout**.

For further details on the representation used, see `fmpz_mod_poly_print()`.

char ***fq_get_str**(const *fq_t* op, const *fq_ctx_t* ctx)

Returns the plain FLINT string representation of the element **op**.

char ***fq_get_str_pretty**(const *fq_t* op, const *fq_ctx_t* ctx)

Returns a pretty representation of the element **op** using the null-terminated string **x** as the variable name.

11.1.7 Randomisation

void **fq_randtest**(*fq_t* rop, *flint_rand_t* state, const *fq_ctx_t* ctx)
 Generates a random element of $\mathbf{F}_q$.

void **fq_randtest_not_zero**(*fq_t* rop, *flint_rand_t* state, const *fq_ctx_t* ctx)
 Generates a random non-zero element of $\mathbf{F}_q$.

void **fq_randtest_dense**(*fq_t* rop, *flint_rand_t* state, const *fq_ctx_t* ctx)
 Generates a random element of $\mathbf{F}_q$ which has an underlying polynomial with dense coefficients.

void **fq_rand**(*fq_t* rop, *flint_rand_t* state, const *fq_ctx_t* ctx)
 Generates a high quality random element of $\mathbf{F}_q$.

void **fq_rand_not_zero**(*fq_t* rop, *flint_rand_t* state, const *fq_ctx_t* ctx)
 Generates a high quality non-zero random element of $\mathbf{F}_q$.

11.1.8 Assignments and conversions

void **fq_set**(*fq_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)
 Sets rop to op.

void **fq_set_si**(*fq_t* rop, const *slong* x, const *fq_ctx_t* ctx)
 Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_set_ui**(*fq_t* rop, const *ulong* x, const *fq_ctx_t* ctx)
 Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_set_fmpz**(*fq_t* rop, const *fmpz_t* x, const *fq_ctx_t* ctx)
 Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_swap**(*fq_t* op1, *fq_t* op2, const *fq_ctx_t* ctx)
 Swaps the two elements op1 and op2.

void **fq_zero**(*fq_t* rop, const *fq_ctx_t* ctx)
 Sets rop to zero.

void **fq_one**(*fq_t* rop, const *fq_ctx_t* ctx)
 Sets rop to one, reduced in the given context.

void **fq_gen**(*fq_t* rop, const *fq_ctx_t* ctx)
 Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int **fq_get_fmpz**(*fmpz_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)
 If op has a lift to the integers, return 1 and set rop to the lift in $[0, p)$. Otherwise, return 0 and leave rop undefined.

void **fq_get_fmpz_poly**(*fmpz_poly_t* a, const *fq_t* b, const *fq_ctx_t* ctx)
 Set a to a representative of b in ctx. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void **fq_set_fmpz_poly**(*fq_t* a, const *fmpz_poly_t* b, const *fq_ctx_t* ctx)
 Set a to the element in ctx with representative b. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void **fq_get_fmpz_mod_mat**(*fmpz_mod_mat_t* col, const *fq_t* a, const *fq_ctx_t* ctx)
 Convert *a* to a column vector of length `degree(ctx)`.

void **fq_set_fmpz_mod_mat**(*fq_t* a, const *fmpz_mod_mat_t* col, const *fq_ctx_t* ctx)
 Convert a column vector *col* of length `degree(ctx)` to an element of *ctx*.

11.1.9 Comparison

int **fq_is_zero**(const *fq_t* op, const *fq_ctx_t* ctx)
 Returns whether *op* is equal to zero.

int **fq_is_one**(const *fq_t* op, const *fq_ctx_t* ctx)
 Returns whether *op* is equal to one.

int **fq_equal**(const *fq_t* op1, const *fq_t* op2, const *fq_ctx_t* ctx)
 Returns whether *op1* and *op2* are equal.

int **fq_is_invertible**(const *fq_t* op, const *fq_ctx_t* ctx)
 Returns whether *op* is an invertible element.

int **fq_is_invertible_f**(*fq_t* f, const *fq_t* op, const *fq_ctx_t* ctx)
 Returns whether *op* is an invertible element. If it is not, then *f* is set of a factor of the modulus.

11.1.10 Special functions

void **_fq_trace**(*fmpz_t* rop, const *fmpz_t* *op, *slong* len, const *fq_ctx_t* ctx)
 Sets *rop* to the trace of the non-zero element (*op*, *len*) in $\mathbf{F}_q$.

void **fq_trace**(*fmpz_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)
 Sets *rop* to the trace of *op*.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

void **_fq_norm**(*fmpz_t* rop, const *fmpz_t* *op, *slong* len, const *fq_ctx_t* ctx)
 Sets *rop* to the norm of the non-zero element (*op*, *len*) in $\mathbf{F}_q$.

void **fq_norm**(*fmpz_t* rop, const *fq_t* op, const *fq_ctx_t* ctx)
 Computes the norm of *op*.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbf{F}_p}(\mathbf{F}_q)$.

Algorithm selection is automatic depending on the input.

void **_fq_frobenius**(*fmpz_t* *rop, const *fmpz_t* *op, *slong* len, *slong* e, const *fq_ctx_t* ctx)
 Sets (*rop*, `2d-1`) to the image of (*op*, *len*) under the Frobenius operator raised to the *e*-th power, assuming that neither *op* nor *e* are zero.

void **fq_frobenius**(*fq_t* rop, const *fq_t* op, *slong* e, const *fq_ctx_t* ctx)
 Evaluates the homomorphism Σ^e at *op*.

Recall that $\mathbf{F}_q/\mathbf{F}_p$ is Galois with Galois group $\langle \sigma \rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma: x \mapsto x^p$.

int **fq_multiplicative_order**(*fmpz* *ord, const *fq_t* op, const *fq_ctx_t* ctx)

Computes the order of op as an element of the multiplicative group of ctx.

Returns 0 if op is 0, otherwise it returns 1 if op is a generator of the multiplicative group, and -1 if it is not.

This function can also be used to check primitivity of a generator of a finite field whose defining polynomial is not primitive.

int **fq_is_primitive**(const *fq_t* op, const *fq_ctx_t* ctx)

Returns whether op is primitive, i.e., whether it is a generator of the multiplicative group of ctx.

11.1.11 Bit packing

void **fq_bit_pack**(*fmpz_t* f, const *fq_t* op, *flint_bitcnt_t* bit_size, const *fq_ctx_t* ctx)

Packs op into bitfields of size bit_size, writing the result to f.

void **fq_bit_unpack**(*fq_t* rop, const *fmpz_t* f, *flint_bitcnt_t* bit_size, const *fq_ctx_t* ctx)

Unpacks into rop the element with coefficients packed into fields of size bit_size as represented by the integer f.

11.2 fq_default.h – unified finite fields

11.2.1 Types, macros and constants

type **fq_default_ctx_t**

type **fq_default_t**

11.2.2 Context Management

void **fq_default_ctx_init_type**(*fq_default_ctx_t* ctx, const *fmpz_t* p, *slong* d, const char *var, int type)

void **fq_default_ctx_init**(*fq_default_ctx_t* ctx, const *fmpz_t* p, *slong* d, const char *var)

Initialises the context ctx for prime *p* and extension degree *d*, with string var of length at least one for the generator display name. By default, it will try use a Conway polynomial; if one is not available, a random irreducible polynomial will be used.

For fq_default_ctx_init, it will choose the best representation for performance.

For fq_default_ctx_init_type, a separate argument type is required which sets which representation to use. These values can be: 0 (which then will act just like fq_default_ctx_init), FQ_DEFAULT_FQ_ZECH, FQ_DEFAULT_FQ_NMOD, FQ_DEFAULT_FQ, FQ_DEFAULT_NMOD and FQ_DEFAULT_FMPZ_MOD.

void **fq_default_ctx_init_modulus_nmod_type**(*fq_default_ctx_t* ctx, const *nmod_poly_t* modulus, const char *var, int type)

void **fq_default_ctx_init_modulus_nmod**(*fq_default_ctx_t* ctx, const *nmod_poly_t* modulus, const char *var)

void **fq_default_ctx_init_modulus_type**(*fq_default_ctx_t* ctx, const *fmpz_mod_poly_t* modulus, *fmpz_mod_ctx_t* mod_ctx, const char *var, int type)


```
void fq_default_ctx_init_modulus(fq_default_ctx_t ctx, const fmpz_mod_poly_t modulus,
                                fmpz_mod_ctx_t mod_ctx, const char *var)
```

Initialises the finite field context `ctx` defined by the given polynomial `modulus`. For the `fmpz_mod_poly` type, the context structure `mod_ctx` for the polynomial must also be given. Sets the printing of variable of the field to the string `var`, which is assumed to be length of at least one.

The context `ctx` will after the call represent the finite field in one of the five different formats: `fq_zech`, `fq_nmod`, `nmod`, `fmpz_mod` and `fq`.

The characteristic of the field will be the modulus of the polynomial and its degree will equal to the degree of the polynomial. Furthermore, it assumes that the characteristic is prime and that the polynomial irreducible. Furthermore, in order for the field to be representable as the Zech logarithm we assume that polynomial is primitive; if it is not, another representation will be chosen.

For `fq_default_ctx_init_modulus_nmod` or `fq_default_ctx_init_modulus`, it chooses the best representation for performance.

For `fq_default_ctx_init_modulus_nmod_type` or `fq_default_ctx_init_modulus_type`, it expects `type` to be one of the following choices: `FQ_DEFAULT_FQ_ZECH`, `FQ_DEFAULT_FQ_NMOD`, `FQ_DEFAULT_FQ`, `FQ_DEFAULT_NMOD` or `FQ_DEFAULT_FMPZ_MOD`. To be clear: if the Zech logarithm is chosen but the polynomial is not primitive, another representation will be chosen.

```
void fq_default_ctx_clear(fq_default_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
int fq_default_ctx_type(const fq_default_ctx_t ctx)
```

Returns 1 if the context contains an `fq_zech` context, 2 if it contains an `fq_mod` context and 3 if it contains an `fq` context.

```
void *fq_default_ctx_inner(const fq_default_ctx_t ctx)
```

Returns a pointer to the internal context object of type `fq_ctx_t`, `fq_zech_ctx_t`, `fmpz_mod_ctx_t`, etc.

```
ulong fq_default_ctx_degree(const fq_default_ctx_t ctx)
```

Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

```
void fq_default_ctx_prime(fmpz_t prime, const fq_default_ctx_t ctx)
```

Sets `prime` to the prime p in the context.

```
void fq_default_ctx_order(fmpz_t f, const fq_default_ctx_t ctx)
```

Sets `f` to be the size of the finite field.

```
void fq_default_ctx_modulus(fmpz_mod_poly_t p, const fq_default_ctx_t ctx)
```

Sets `p` to the defining polynomial of the finite field..

```
int fq_default_ctx_fprint(FILE *file, const fq_default_ctx_t ctx)
```

Prints the context information to `file`. Returns 1 for a success and a negative number for an error.

```
void fq_default_ctx_print(const fq_default_ctx_t ctx)
```

Prints the context information to `stdout`.

```
void fq_default_ctx_init_randtest(fq_default_ctx_t ctx, flint_rand_t state)
```

Initializes `ctx` to a random finite field using one of the five internal representations. Assumes that `fq_default_ctx_init` has not been called on `ctx` already.

```
void fq_default_get_coeff_fmpz(fmpz_t c, fq_default_t op, ulong n, const fq_default_ctx_t ctx)
```

Set `c` to the degree n coefficient of the polynomial representation of the finite field element `op`.

11.2.3 Memory management

void **fq_default_init**(*fq_default_t* rop, const *fq_default_ctx_t* ctx)

Initialises the element *rop*, setting its value to 0.

void **fq_default_init2**(*fq_default_t* rop, const *fq_default_ctx_t* ctx)

Initialises *poly* with at least enough space for it to be an element of *ctx* and sets it to 0.

void **fq_default_clear**(*fq_default_t* rop, const *fq_default_ctx_t* ctx)

Clears the element *rop*.

11.2.4 Predicates

int **fq_default_is_invertible**(const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Return 1 if *op* is an invertible element.

11.2.5 Basic arithmetic

void **fq_default_add**(*fq_default_t* rop, const *fq_default_t* op1, const *fq_default_t* op2, const *fq_default_ctx_t* ctx)

Sets *rop* to the sum of *op1* and *op2*.

void **fq_default_sub**(*fq_default_t* rop, const *fq_default_t* op1, const *fq_default_t* op2, const *fq_default_ctx_t* ctx)

Sets *rop* to the difference of *op1* and *op2*.

void **fq_default_sub_one**(*fq_default_t* rop, const *fq_default_t* op1, const *fq_default_ctx_t* ctx)

Sets *rop* to the difference of *op1* and 1.

void **fq_default_neg**(*fq_default_t* rop, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Sets *rop* to the negative of *op*.

void **fq_default_mul**(*fq_default_t* rop, const *fq_default_t* op1, const *fq_default_t* op2, const *fq_default_ctx_t* ctx)

Sets *rop* to the product of *op1* and *op2*, reducing the output in the given context.

void **fq_default_mul_fmpz**(*fq_default_t* rop, const *fq_default_t* op, const *fmpz_t* x, const *fq_default_ctx_t* ctx)

Sets *rop* to the product of *op* and *x*, reducing the output in the given context.

void **fq_default_mul_si**(*fq_default_t* rop, const *fq_default_t* op, *slong* x, const *fq_default_ctx_t* ctx)

Sets *rop* to the product of *op* and *x*, reducing the output in the given context.

void **fq_default_mul_ui**(*fq_default_t* rop, const *fq_default_t* op, *ulong* x, const *fq_default_ctx_t* ctx)

Sets *rop* to the product of *op* and *x*, reducing the output in the given context.

void **fq_default_sqr**(*fq_default_t* rop, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Sets *rop* to the square of *op*, reducing the output in the given context.

void **fq_default_div**(*fq_default_t* rop, *fq_default_t* op1, *fq_default_t* op2, const *fq_default_ctx_t* ctx)

Sets *rop* to the quotient of *op1* and *op2*, reducing the output in the given context.

void **fq_default_inv**(*fq_default_t* rop, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Sets *rop* to the inverse of the non-zero element *op*.

```
void fq_default_pow(fq_default_t rop, const fq_default_t op, const fmpz_t e, const
                  fq_default_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e .

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

```
void fq_default_pow_ui(fq_default_t rop, const fq_default_t op, const ulong e, const
                    fq_default_ctx_t ctx)
```

Sets `rop` the `op` raised to the power e .

Currently assumes that $e \geq 0$.

Note that for any input `op`, `rop` is set to 1 whenever $e = 0$.

11.2.6 Roots

```
int fq_default_sqrt(fq_default_t rop, const fq_default_t op1, const fq_default_ctx_t ctx)
```

Sets `rop` to the square root of `op1` if it is a square, and return 1, otherwise return 0.

```
void fq_default_pth_root(fq_default_t rop, const fq_default_t op1, const fq_default_ctx_t ctx)
```

Sets `rop` to a p^{th} root of `op1`. Currently, this computes the root by raising `op1` to p^{d-1} where d is the degree of the extension.

```
int fq_default_is_square(const fq_default_t op, const fq_default_ctx_t ctx)
```

Return 1 if `op` is a square.

11.2.7 Output

```
int fq_default_fprint_pretty(FILE *file, const fq_default_t op, const fq_default_ctx_t ctx)
```

Prints a pretty representation of `op` to `file`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
void fq_default_print_pretty(const fq_default_t op, const fq_default_ctx_t ctx)
```

Prints a pretty representation of `op` to `stdout`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
int fq_default_fprint(FILE *file, const fq_default_t op, const fq_default_ctx_t ctx)
```

Prints a representation of `op` to `file`.

```
void fq_default_print(const fq_default_t op, const fq_default_ctx_t ctx)
```

Prints a representation of `op` to `stdout`.

```
char *fq_default_get_str(const fq_default_t op, const fq_default_ctx_t ctx)
```

Returns the plain FLINT string representation of the element `op`.

```
char *fq_default_get_str_pretty(const fq_default_t op, const fq_default_ctx_t ctx)
```

Returns a pretty representation of the element `op` using the null-terminated string `x` as the variable name.

11.2.8 Randomisation

void **fq_default_randtest**(*fq_default_t* rop, *flint_rand_t* state, const *fq_default_ctx_t* ctx)

Generates a random element of $\mathbf{F}_q$.

void **fq_default_randtest_not_zero**(*fq_default_t* rop, *flint_rand_t* state, const *fq_default_ctx_t* ctx)

Generates a random non-zero element of $\mathbf{F}_q$.

void **fq_default_rand**(*fq_default_t* rop, *flint_rand_t* state, const *fq_default_ctx_t* ctx)

Generates a high quality random element of $\mathbf{F}_q$.

void **fq_default_rand_not_zero**(*fq_default_t* rop, *flint_rand_t* state, const *fq_default_ctx_t* ctx)

Generates a high quality non-zero random element of $\mathbf{F}_q$.

11.2.9 Assignments and conversions

void **fq_default_set**(*fq_default_t* rop, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Sets rop to op.

void **fq_default_set_si**(*fq_default_t* rop, const *slong* x, const *fq_default_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_default_set_ui**(*fq_default_t* rop, const *ulong* x, const *fq_default_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_default_set_fmpz**(*fq_default_t* rop, const *fmpz_t* x, const *fq_default_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_default_swap**(*fq_default_t* op1, *fq_default_t* op2, const *fq_default_ctx_t* ctx)

Swaps the two elements op1 and op2.

void **fq_default_zero**(*fq_default_t* rop, const *fq_default_ctx_t* ctx)

Sets rop to zero.

void **fq_default_one**(*fq_default_t* rop, const *fq_default_ctx_t* ctx)

Sets rop to one, reduced in the given context.

void **fq_default_gen**(*fq_default_t* rop, const *fq_default_ctx_t* ctx)

Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int **fq_default_get_fmpz**(*fmpz_t* rop, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

If op has a lift to the integers, return 1 and set rop to the lift in $[0, p)$. Otherwise, return 0 and leave rop undefined.

void **fq_default_get_nmod_poly**(*nmod_poly_t* poly, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Sets poly to the polynomial representation of op. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

void **fq_default_set_nmod_poly**(*fq_default_t* op, const *nmod_poly_t* poly, const *fq_default_ctx_t* ctx)

Sets op to the finite field element represented by the polynomial poly. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

void **fq_default_get_fmpz_mod_poly**(*fmpz_mod_poly_t* poly, const *fq_default_t* op, const *fq_default_ctx_t* ctx)

Sets poly to the polynomial representation of op. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

```
void fq_default_set_fmpz_mod_poly(fq_default_t op, const fmpz_mod_poly_t poly, const
                                fq_default_ctx_t ctx)
```

Sets `op` to the finite field element represented by the polynomial `poly`. Assumes the characteristic of the field and the modulus of the polynomial are the same. No checking of this occurs.

```
void fq_default_get_fmpz_poly(fmpz_poly_t a, const fq_default_t b, const fq_default_ctx_t ctx)
```

Set `a` to a representative of `b` in `ctx`. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in `ctx`.

```
void fq_default_set_fmpz_poly(fq_default_t a, const fmpz_poly_t b, const fq_default_ctx_t ctx)
```

Set `a` to the element in `ctx` with representative `b`. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in `ctx`.

11.2.10 Comparison

```
int fq_default_is_zero(const fq_default_t op, const fq_default_ctx_t ctx)
```

Returns whether `op` is equal to zero.

```
int fq_default_is_one(const fq_default_t op, const fq_default_ctx_t ctx)
```

Returns whether `op` is equal to one.

```
int fq_default_equal(const fq_default_t op1, const fq_default_t op2, const fq_default_ctx_t ctx)
```

Returns whether `op1` and `op2` are equal.

11.2.11 Special functions

```
void fq_default_trace(fmpz_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
```

Sets `rop` to the trace of `op`.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

```
void fq_default_norm(fmpz_t rop, const fq_default_t op, const fq_default_ctx_t ctx)
```

Computes the norm of `op`.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbf{F}_p}(\mathbf{F}_q)$.

Algorithm selection is automatic depending on the input.

```
void fq_default_frobenius(fq_default_t rop, const fq_default_t op, slong e, const fq_default_ctx_t
                        ctx)
```

Evaluates the homomorphism Σ^e at `op`.

Recall that $\mathbf{F}_q/\mathbf{F}_p$ is Galois with Galois group $\langle \sigma \rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma: x \mapsto x^p$.

11.3 fq_vec.h – vectors over finite fields

11.3.1 Memory management

fq_struct *_fq_vec_init(*slong* len, const *fq_ctx_t* ctx)

Returns an initialised vector of *fq*'s of given length.

void _fq_vec_clear(*fq_struct* *vec, *slong* len, const *fq_ctx_t* ctx)

Clears the entries of (*vec*, *len*) and frees the space allocated for *vec*.

11.3.2 Randomisation

void _fq_vec_randtest(*fq_struct* *f, *flint_rand_t* state, *slong* len, const *fq_ctx_t* ctx)

Sets the entries of a vector of the given length to elements of the finite field.

11.3.3 Input and output

int _fq_vec_fprint(FILE *file, const *fq_struct* *vec, *slong* len, const *fq_ctx_t* ctx)

Prints the vector of given length to the stream *file*. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int _fq_vec_print(const *fq_struct* *vec, *slong* len, const *fq_ctx_t* ctx)

Prints the vector of given length to *stdout*.

For further details, see *_fq_vec_fprint()*.

11.3.4 Assignment and basic manipulation

void _fq_vec_set(*fq_struct* *vec1, const *fq_struct* *vec2, *slong* len2, const *fq_ctx_t* ctx)

Makes a copy of (*vec2*, *len2*) into *vec1*.

void _fq_vec_swap(*fq_struct* *vec1, *fq_struct* *vec2, *slong* len2, const *fq_ctx_t* ctx)

Swaps the elements in (*vec1*, *len2*) and (*vec2*, *len2*).

void _fq_vec_zero(*fq_struct* *vec, *slong* len, const *fq_ctx_t* ctx)

Zeros the entries of (*vec*, *len*).

void _fq_vec_neg(*fq_struct* *vec1, const *fq_struct* *vec2, *slong* len2, const *fq_ctx_t* ctx)

Negates (*vec2*, *len2*) and places it into *vec1*.

11.3.5 Comparison

int _fq_vec_equal(const *fq_struct* *vec1, const *fq_struct* *vec2, *slong* len, const *fq_ctx_t* ctx)

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

int _fq_vec_is_zero(const *fq_struct* *vec, *slong* len, const *fq_ctx_t* ctx)

Returns 1 if (*vec*, *len*) is zero, and 0 otherwise.

11.3.6 Addition and subtraction

```
void _fq_vec_add(fq_struct *res, const fq_struct *vec1, const fq_struct *vec2, slong len2, const
                fq_ctx_t ctx)
```

Sets `(res, len2)` to the sum of `(vec1, len2)` and `(vec2, len2)`.

```
void _fq_vec_sub(fq_struct *res, const fq_struct *vec1, const fq_struct *vec2, slong len2, const
                fq_ctx_t ctx)
```

Sets `(res, len2)` to `(vec1, len2)` minus `(vec2, len2)`.

11.3.7 Scalar multiplication and division

```
void _fq_vec_scalar_addmul_fq(fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_t c,
                              const fq_ctx_t ctx)
```

Adds `(vec2, len2)` times `c` to `(vec1, len2)`, where `c` is a `fq_t`.

```
void _fq_vec_scalar_submul_fq(fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_t c,
                              const fq_ctx_t ctx)
```

Subtracts `(vec2, len2)` times `c` from `(vec1, len2)`, where `c` is a `fq_t`.

11.3.8 Dot products

```
void _fq_vec_dot(fq_t res, const fq_struct *vec1, const fq_struct *vec2, slong len2, const fq_ctx_t
                ctx)
```

Sets `res` to the dot product of `(vec1, len)` and `(vec2, len)`.

11.4 fq_mat.h – matrices over finite fields

11.4.1 Types, macros and constants

```
type fq_mat_struct
```

```
type fq_mat_t
```

11.4.2 Memory management

```
void fq_mat_init(fq_mat_t mat, slong rows, slong cols, const fq_ctx_t ctx)
```

Initialises `mat` to a `rows`-by-`cols` matrix with coefficients in $\mathbf{F}_q$ given by `ctx`. All elements are set to zero.

```
void fq_mat_init_set(fq_mat_t mat, const fq_mat_t src, const fq_ctx_t ctx)
```

Initialises `mat` and sets its dimensions and elements to those of `src`.

```
void fq_mat_clear(fq_mat_t mat, const fq_ctx_t ctx)
```

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an `fq_mat_t` object.

```
void fq_mat_set(fq_mat_t mat, const fq_mat_t src, const fq_ctx_t ctx)
```

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

11.4.3 Basic properties and manipulation

fq_struct ***fq_mat_entry**(const *fq_mat_t* mat, *slong* i, *slong* j)

Directly accesses the entry in **mat** in row *i* and column *j*, indexed from zero. No bounds checking is performed.

void **fq_mat_entry_set**(*fq_mat_t* mat, *slong* i, *slong* j, const *fq_t* x, const *fq_ctx_t* ctx)

Sets the entry in **mat** in row *i* and column *j* to *x*.

slong **fq_mat_nrows**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Returns the number of rows in **mat**.

slong **fq_mat_ncols**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Returns the number of columns in **mat**.

void **fq_mat_swap**(*fq_mat_t* mat1, *fq_mat_t* mat2, const *fq_ctx_t* ctx)

Swaps two matrices. The dimensions of **mat1** and **mat2** are allowed to be different.

void **fq_mat_swap_entrywise**(*fq_mat_t* mat1, *fq_mat_t* mat2, const *fq_ctx_t* ctx)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void **fq_mat_zero**(*fq_mat_t* mat, const *fq_ctx_t* ctx)

Sets all entries of **mat** to 0.

void **fq_mat_one**(*fq_mat_t* mat, const *fq_ctx_t* ctx)

Sets all the diagonal entries of **mat** to 1 and all other entries to 0.

void **fq_mat_swap_rows**(*fq_mat_t* mat, *slong* *perm, *slong* r, *slong* s, const *fq_ctx_t* ctx)

Swaps rows *r* and *s* of **mat**. If *perm* is non-NULL, the permutation of the rows will also be applied to *perm*.

void **fq_mat_swap_cols**(*fq_mat_t* mat, *slong* *perm, *slong* r, *slong* s, const *fq_ctx_t* ctx)

Swaps columns *r* and *s* of **mat**. If *perm* is non-NULL, the permutation of the columns will also be applied to *perm*.

void **fq_mat_invert_rows**(*fq_mat_t* mat, *slong* *perm, const *fq_ctx_t* ctx)

Swaps rows *i* and *r - i* of **mat** for $0 \leq i < r/2$, where *r* is the number of rows of **mat**. If *perm* is non-NULL, the permutation of the rows will also be applied to *perm*.

void **fq_mat_invert_cols**(*fq_mat_t* mat, *slong* *perm, const *fq_ctx_t* ctx)

Swaps columns *i* and *c - i* of **mat** for $0 \leq i < c/2$, where *c* is the number of columns of **mat**. If *perm* is non-NULL, the permutation of the columns will also be applied to *perm*.

11.4.4 Conversions

void **fq_mat_set_nmod_mat**(*fq_mat_t* mat1, const *nmod_mat_t* mat2, const *fq_ctx_t* ctx)

Sets the matrix **mat1** to the matrix **mat2**.

void **fq_mat_set_fmpz_mod_mat**(*fq_mat_t* mat1, const *fmpz_mod_mat_t* mat2, const *fq_ctx_t* ctx)

Sets the matrix **mat1** to the matrix **mat2**.

11.4.5 Concatenate

void **fq_mat_concat_vertical**(*fq_mat_t* res, const *fq_mat_t* mat1, const *fq_mat_t* mat2, const *fq_ctx_t* ctx)

Sets **res** to vertical concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $k \times n$, **res** : $(m + k) \times n$.

void **fq_mat_concat_horizontal**(*fq_mat_t* res, const *fq_mat_t* mat1, const *fq_mat_t* mat2, const *fq_ctx_t* ctx)

Sets **res** to horizontal concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $m \times k$, **res** : $m \times (n + k)$.

11.4.6 Printing

int **fq_mat_print_pretty**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Pretty-prints **mat** to **stdout**. A header is printed followed by the rows enclosed in brackets.

int **fq_mat_fprint_pretty**(FILE *file, const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Pretty-prints **mat** to **file**. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_mat_print**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Prints **mat** to **stdout**. A header is printed followed by the rows enclosed in brackets.

int **fq_mat_fprint**(FILE *file, const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Prints **mat** to **file**. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.4.7 Window

void **fq_mat_window_init**(*fq_mat_t* window, const *fq_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2, const *fq_ctx_t* ctx)

Initializes the matrix **window** to be an $r2 - r1$ by $c2 - c1$ submatrix of **mat** whose (0,0) entry is the (**r1**, **c1**) entry of **mat**. The memory for the elements of **window** is shared with **mat**.

void **fq_mat_window_clear**(*fq_mat_t* window, const *fq_ctx_t* ctx)

Clears the matrix **window** and releases any memory that it uses. Note that the memory to the underlying matrix that **window** points to is not freed.

11.4.8 Random matrix generation

void **fq_mat_randtest**(*fq_mat_t* mat, *flint_rand_t* state, const *fq_ctx_t* ctx)

Sets the elements of **mat** to random elements of $\mathbf{F}_q$, given by **ctx**.

int **fq_mat_randpermdiag**(*fq_mat_t* mat, *flint_rand_t* state, *fq_struct* *diag, *slong* n, const *fq_ctx_t* ctx)

Sets **mat** to a random permutation of the diagonal matrix with n leading entries given by the vector **diag**. It is assumed that the main diagonal of **mat** has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

void **fq_mat_randrank**(*fq_mat_t* mat, *flint_rand_t* state, *slong* rank, const *fq_ctx_t* ctx)

Sets **mat** to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbf{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling *fq_mat_randops*().

void **fq_mat_randops**(*fq_mat_t* mat, *flint_rand_t* state, *slong* count, const *fq_ctx_t* ctx)

Randomises **mat** by performing elementary row or column operations. More precisely, at most **count** random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void **fq_mat_randtril**(*fq_mat_t* mat, *flint_rand_t* state, int unit, const *fq_ctx_t* ctx)

Sets **mat** to a random lower triangular matrix. If **unit** is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

void **fq_mat_randtriu**(*fq_mat_t* mat, *flint_rand_t* state, int unit, const *fq_ctx_t* ctx)

Sets **mat** to a random upper triangular matrix. If **unit** is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

11.4.9 Comparison

int **fq_mat_equal**(const *fq_mat_t* mat1, const *fq_mat_t* mat2, const *fq_ctx_t* ctx)

Returns nonzero if mat1 and mat2 have the same dimensions and elements, and zero otherwise.

int **fq_mat_is_zero**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Returns a non-zero value if all entries of **mat** are zero, and otherwise returns zero.

int **fq_mat_is_one**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Returns a non-zero value if all entries **mat** are zero except the diagonal entries which must be one, otherwise returns zero..

int **fq_mat_is_empty**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Returns a non-zero value if the number of rows or the number of columns in **mat** is zero, and otherwise returns zero.

int **fq_mat_is_square**(const *fq_mat_t* mat, const *fq_ctx_t* ctx)

Returns a non-zero value if the number of rows is equal to the number of columns in **mat**, and otherwise returns zero.

11.4.10 Transpose

void **fq_mat_transpose**(*fq_mat_t* B, const *fq_mat_t* A, const *fq_ctx_t* ctx)

Sets B to A^T , the transpose of A . Dimensions must be compatible. Aliasing is allowed for square matrices.

11.4.11 Addition and subtraction

void **fq_mat_add**(*fq_mat_t* C, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Computes $C = A + B$. Dimensions must be identical.

void **fq_mat_sub**(*fq_mat_t* C, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Computes $C = A - B$. Dimensions must be identical.

void **fq_mat_neg**(*fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Sets $B = -A$. Dimensions must be identical.

11.4.12 Matrix multiplication

void **fq_mat_mul**(*fq_mat_t* C, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

void **fq_mat_mul_classical**(*fq_mat_t* C, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses classical matrix multiplication.

void **fq_mat_mul_KS**(*fq_mat_t* C, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses Kronecker substitution to perform the multiplication over the integers.

void **fq_mat_submul**(*fq_mat_t* D, const *fq_mat_t* C, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B .

void **fq_mat_mul_vec**(*fq_struct* *c, const *fq_mat_t* A, const *fq_struct* *b, *slong* blen, const *fq_ctx_t* ctx)

void **fq_mat_mul_vec_ptr**(*fq_struct* *const *c, const *fq_mat_t* A, const *fq_struct* *const *b, *slong* blen, const *fq_ctx_t* ctx)

Compute a matrix-vector product of A and (b, blen) and store the result in c . The vector (b, blen) is either truncated or zero-extended to the number of columns of A . The number entries written to c is always equal to the number of rows of A .

void **fq_mat_vec_mul**(*fq_struct* *c, const *fq_struct* *a, *slong* alen, const *fq_mat_t* B, const *fq_ctx_t* ctx)

void **fq_mat_vec_mul_ptr**(*fq_struct* *const *c, const *fq_struct* *const *a, *slong* alen, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Compute a vector-matrix product of (a, alen) and B and store the result in c . The vector (a, alen) is either truncated or zero-extended to the number of rows of B . The number entries written to c is always equal to the number of columns of B .

11.4.13 Inverse

int **fq_mat_inv**(*fq_mat_t* B, *fq_mat_t* A, const *fq_ctx_t* ctx)

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of B to undefined values.

A and B must be square matrices with the same dimensions.

11.4.14 LU decomposition

slong **fq_mat_lu**(*slong* *P, *fq_mat_t* A, int rank_check, const *fq_ctx_t* ctx)

Computes a generalised LU decomposition $PLU = A$ of a given matrix A , returning the rank of A .

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

11.4.15 Reduced row echelon form

slong **fq_mat_rref**(*fq_mat_t* B, const *fq_mat_t* A, const *fq_ctx_t* ctx)

Puts B in reduced row echelon form and returns the rank of A .

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

slong **fq_mat_reduce_row**(*fq_mat_t* A, *slong* *P, *slong* *L, *slong* n, const *fq_ctx_t* ctx)

Reduce row n of the matrix A , assuming the prior rows are in Gauss form. However those rows may not be in order. The entry i of the array P is the row of A which has a pivot in the i -th column. If no such row exists, the entry of P will be -1 . The function returns the column in which the n -th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in P . Entry i of the array L contains the number of possibly nonzero columns of A row i . This speeds up reduction in the case that A is chambered on the right. Otherwise the entries of L can all be set to the number of columns of A . We require the entries of L to be monotonic increasing.

11.4.16 Triangular solving

void **fq_mat_solve_tril**(*fq_mat_t* X, const *fq_mat_t* L, const *fq_mat_t* B, int unit, const *fq_ctx_t* ctx)

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` = 1, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

void **fq_mat_solve_triu**(*fq_mat_t* X, const *fq_mat_t* U, const *fq_mat_t* B, int unit, const *fq_ctx_t* ctx)

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` = 1, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

11.4.17 Solving

int **fq_mat_solve**(*fq_mat_t* X, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

The matrix A must be square.

int **fq_mat_can_solve**(*fq_mat_t* X, const *fq_mat_t* A, const *fq_mat_t* B, const *fq_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$ over Fq .

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of X to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of A and it may be singular.

11.4.18 Transforms

void **fq_mat_similarity**(*fq_mat_t* M, *slong* r, *fq_t* d, const *fq_ctx_t* ctx)

Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value d is required to be reduced modulo the modulus of the entries in the matrix.

11.4.19 Characteristic polynomial

void **fq_mat_charpoly**(*fq_poly_t* p, const *fq_mat_t* M, const *fq_ctx_t* ctx)

Compute the characteristic polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.4.20 Minimal polynomial

void **fq_mat_minpoly**(*fq_poly_t* p, const *fq_mat_t* M, const *fq_ctx_t* ctx)

Compute the minimal polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.5 fq_default_mat.h – matrices over finite fields

11.5.1 Types, macros and constants

type **fq_default_mat_t**

11.5.2 Memory management

void **fq_default_mat_init**(*fq_default_mat_t* mat, *slong* rows, *slong* cols, const *fq_default_ctx_t* ctx)

Initialises **mat** to a rows-by-cols matrix with coefficients in $\mathbf{F}_q$ given by **ctx**. All elements are set to zero.

void **fq_default_mat_init_set**(*fq_default_mat_t* mat, const *fq_default_mat_t* src, const *fq_default_ctx_t* ctx)

Initialises **mat** and sets its dimensions and elements to those of **src**.

void **fq_default_mat_clear**(*fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an **fq_default_mat_t** object.

void **fq_default_mat_set**(*fq_default_mat_t* mat, const *fq_default_mat_t* src, const *fq_default_ctx_t* ctx)

Sets **mat** to a copy of **src**. It is assumed that **mat** and **src** have identical dimensions.

11.5.3 Basic properties and manipulation

void **fq_default_mat_entry**(*fq_default_t* val, const *fq_default_mat_t* mat, *slong* i, *slong* j, const *fq_default_ctx_t* ctx)

Directly accesses the entry in **mat** in row *i* and column *j*, indexed from zero by setting **val** to the value of that entry. No bounds checking is performed.

void **fq_default_mat_entry_set**(*fq_default_mat_t* mat, *slong* i, *slong* j, const *fq_default_t* x, const *fq_default_ctx_t* ctx)

Sets the entry in **mat** in row *i* and column *j* to **x**.

void **fq_default_mat_entry_set_fmpz**(*fq_default_mat_t* mat, *slong* i, *slong* j, const *fmpz_t* x, const *fq_default_ctx_t* ctx)

Sets the entry in **mat** in row *i* and column *j* to **x**.

slong **fq_default_mat_nrows**(const *fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Returns the number of rows in **mat**.

slong **fq_default_mat_ncols**(const *fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Returns the number of columns in **mat**.

void **fq_default_mat_swap**(*fq_default_mat_t* mat1, *fq_default_mat_t* mat2, const *fq_default_ctx_t* ctx)

Swaps two matrices. The dimensions of **mat1** and **mat2** are allowed to be different.

void **fq_default_mat_zero**(*fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Sets all entries of **mat** to 0.

void **fq_default_mat_one**(*fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Sets the diagonal entries of **mat** to 1 and all other entries to 0.

void **fq_default_mat_swap_rows**(*fq_default_mat_t* mat, *slong* *perm, *slong* r, *slong* s, const *fq_default_ctx_t* ctx)

Swaps rows **r** and **s** of **mat**. If **perm** is non-NULL, the permutation of the rows will also be applied to **perm**.

void **fq_default_mat_swap_cols**(*fq_default_mat_t* mat, *slong* *perm, *slong* r, *slong* s, const *fq_default_ctx_t* ctx)

Swaps columns **r** and **s** of **mat**. If **perm** is non-NULL, the permutation of the columns will also be applied to **perm**.

void **fq_default_mat_invert_rows**(*fq_default_mat_t* mat, *slong* *perm, const *fq_default_ctx_t* ctx)

Swaps rows **i** and **r - i** of **mat** for $0 \leq i < r/2$, where **r** is the number of rows of **mat**. If **perm** is non-NULL, the permutation of the rows will also be applied to **perm**.

void **fq_default_mat_invert_cols**(*fq_default_mat_t* mat, *slong* *perm, const *fq_default_ctx_t* ctx)

Swaps columns **i** and **c - i** of **mat** for $0 \leq i < c/2$, where **c** is the number of columns of **mat**. If **perm** is non-NULL, the permutation of the columns will also be applied to **perm**.

11.5.4 Conversions

void **fq_default_mat_set_nmod_mat**(*fq_default_mat_t* mat1, const *nmod_mat_t* mat2, const *fq_default_ctx_t* ctx)

Sets the matrix **mat1** to the matrix **mat2**.

void **fq_default_mat_set_fmpz_mod_mat**(*fq_default_mat_t* mat1, const *fmpz_mod_mat_t* mat2, const *fq_default_ctx_t* ctx)

Sets the matrix **mat1** to the matrix **mat2**.


```
void fq_default_mat_set_fmpz_mat(fq_default_mat_t mat1, const fmpz_mat_t mat2, const  
                                fq_default_ctx_t ctx)
```

Sets the matrix `mat1` to the matrix `mat2`, reducing the entries modulo the characteristic of the finite field.

11.5.5 Concatenate

```
void fq_default_mat_concat_vertical(fq_default_mat_t res, const fq_default_mat_t mat1, const  
                                   fq_default_mat_t mat2, const fq_default_ctx_t ctx)
```

Sets `res` to vertical concatenation of `(mat1, mat2)` in that order. Matrix dimensions : `mat1` : $m \times n$, `mat2` : $k \times n$, `res` : $(m + k) \times n$.

```
void fq_default_mat_concat_horizontal(fq_default_mat_t res, const fq_default_mat_t mat1,  
                                     const fq_default_mat_t mat2, const fq_default_ctx_t  
                                     ctx)
```

Sets `res` to horizontal concatenation of `(mat1, mat2)` in that order. Matrix dimensions : `mat1` : $m \times n$, `mat2` : $m \times k$, `res` : $m \times (n + k)$.

11.5.6 Printing

```
int fq_default_mat_print_pretty(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Pretty-prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_default_mat_fprint_pretty(FILE *file, const fq_default_mat_t mat, const fq_default_ctx_t  
                                ctx)
```

Pretty-prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_default_mat_print(const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

```
int fq_default_mat_fprint(FILE *file, const fq_default_mat_t mat, const fq_default_ctx_t ctx)
```

Prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.5.7 Window

```
void fq_default_mat_window_init(fq_default_mat_t window, const fq_default_mat_t mat, slong  
                               r1, slong c1, slong r2, slong c2, const fq_default_ctx_t ctx)
```

Initializes the matrix `window` to be an $r2 - r1$ by $c2 - c1$ submatrix of `mat` whose $(0,0)$ entry is the $(r1, c1)$ entry of `mat`. The memory for the elements of `window` is shared with `mat`.

```
void fq_default_mat_window_clear(fq_default_mat_t window, const fq_default_ctx_t ctx)
```

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

11.5.8 Random matrix generation

void **fq_default_mat_randtest**(*fq_default_mat_t* mat, *flint_rand_t* state, const *fq_default_ctx_t* ctx)

Sets the elements of **mat** to random elements of $\mathbf{F}_q$, given by **ctx**.

int **fq_default_mat_randpermdiag**(*fq_mat_t* mat, *flint_rand_t* state, *fq_struct* *diag, *slong* n, const *fq_ctx_t* ctx)

Sets **mat** to a random permutation of the diagonal matrix with n leading entries given by the vector **diag**. It is assumed that the main diagonal of **mat** has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

void **fq_default_mat_randrank**(*fq_default_mat_t* mat, *flint_rand_t* state, *slong* rank, const *fq_default_ctx_t* ctx)

Sets **mat** to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbf{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling *fq_default_mat_randops*().

void **fq_default_mat_randops**(*fq_default_mat_t* mat, *flint_rand_t* state, *slong* count, const *fq_default_ctx_t* ctx)

Randomises **mat** by performing elementary row or column operations. More precisely, at most **count** random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void **fq_default_mat_randtril**(*fq_default_mat_t* mat, *flint_rand_t* state, int unit, const *fq_default_ctx_t* ctx)

Sets **mat** to a random lower triangular matrix. If **unit** is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

void **fq_default_mat_randtriu**(*fq_default_mat_t* mat, *flint_rand_t* state, int unit, const *fq_default_ctx_t* ctx)

Sets **mat** to a random upper triangular matrix. If **unit** is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

11.5.9 Comparison

int **fq_default_mat_equal**(const *fq_default_mat_t* mat1, const *fq_default_mat_t* mat2, const *fq_default_ctx_t* ctx)

Returns nonzero if **mat1** and **mat2** have the same dimensions and elements, and zero otherwise.

int **fq_default_mat_is_zero**(const *fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Returns a non-zero value if all entries of **mat** are zero, and otherwise returns zero.

int **fq_default_mat_is_one**(const *fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Returns a non-zero value if all diagonal entries of **mat** are one and all other entries are zero, and otherwise returns zero.

int **fq_default_mat_is_empty**(const *fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Returns a non-zero value if the number of rows or the number of columns in **mat** is zero, and otherwise returns zero.

int **fq_default_mat_is_square**(const *fq_default_mat_t* mat, const *fq_default_ctx_t* ctx)

Returns a non-zero value if the number of rows is equal to the number of columns in **mat**, and otherwise returns zero.

11.5.10 Transpose

```
void fq_default_mat_transpose(fq_default_mat_t B, const fq_default_mat_t A, const
                             fq_default_ctx_t ctx)
```

Sets B to A^T , the transpose of A . Dimensions must be compatible. Aliasing is allowed for square matrices.

11.5.11 Addition and subtraction

```
void fq_default_mat_add(fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t
                      B, const fq_default_ctx_t ctx)
```

Computes $C = A + B$. Dimensions must be identical.

```
void fq_default_mat_sub(fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t
                      B, const fq_default_ctx_t ctx)
```

Computes $C = A - B$. Dimensions must be identical.

```
void fq_default_mat_neg(fq_default_mat_t A, const fq_default_mat_t B, const fq_default_ctx_t
                      ctx)
```

Sets $B = -A$. Dimensions must be identical.

11.5.12 Matrix multiplication

```
void fq_default_mat_mul(fq_default_mat_t C, const fq_default_mat_t A, const fq_default_mat_t
                      B, const fq_default_ctx_t ctx)
```

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

```
void fq_default_mat_submul(fq_default_mat_t D, const fq_default_mat_t C, const
                          fq_default_mat_t A, const fq_default_mat_t B, const
                          fq_default_ctx_t ctx)
```

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B .

11.5.13 Inverse

```
int fq_default_mat_inv(fq_default_mat_t B, fq_default_mat_t A, const fq_default_ctx_t ctx)
```

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of B to undefined values.

A and B must be square matrices with the same dimensions.

11.5.14 LU decomposition

```
slong fq_default_mat_lu(slong *P, fq_default_mat_t A, int rank_check, const fq_default_ctx_t
                      ctx)
```

Computes a generalised LU decomposition $PLU = A$ of a given matrix A , returning the rank of A .

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

This function calls `fq_default_mat_lu_recursive`.

11.5.15 Reduced row echelon form

ulong `fq_default_mat_rref`(*fq_default_mat_t* B, const *fq_default_mat_t* A, const *fq_default_ctx_t* ctx)

Puts B in reduced row echelon form and returns the rank of A .

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

11.5.16 Triangular solving

void `fq_default_mat_solve_tril`(*fq_default_mat_t* X, const *fq_default_mat_t* L, const *fq_default_mat_t* B, int unit, const *fq_default_ctx_t* ctx)

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` = 1, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

void `fq_default_mat_solve_triu`(*fq_default_mat_t* X, const *fq_default_mat_t* U, const *fq_default_mat_t* B, int unit, const *fq_default_ctx_t* ctx)

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` = 1, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

11.5.17 Solving

int `fq_default_mat_solve`(*fq_default_mat_t* X, const *fq_default_mat_t* A, const *fq_default_mat_t* B, const *fq_default_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

The matrix A must be square.

int `fq_default_mat_can_solve`(*fq_default_mat_t* X, const *fq_default_mat_t* A, const *fq_default_mat_t* B, const *fq_default_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$ over Fq .

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of X to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of A and it may be singular.

11.5.18 Transforms

```
void fq_default_mat_similarity(fq_default_mat_t M, slong r, fq_default_t d, const
                             fq_default_ctx_t ctx)
```

Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value d is required to be reduced modulo the modulus of the entries in the matrix.

11.5.19 Characteristic polynomial

```
void fq_default_mat_charpoly(fq_default_poly_t p, const fq_default_mat_t M, const
                             fq_default_ctx_t ctx)
```

Compute the characteristic polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.5.20 Minimal polynomial

```
void fq_default_mat_minpoly(fq_default_poly_t p, const fq_default_mat_t M, const
                             fq_default_ctx_t ctx)
```

Compute the minimal polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.6 fq_poly.h – univariate polynomials over finite fields

We represent a polynomial in $\mathbf{F}_q[X]$ as a **struct** which includes an array **coeffs** with the coefficients, as well as the length **length** and the number **alloc** of coefficients for which memory has been allocated.

As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $\mathbf{F}_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

11.6.1 Types, macros and constants

```
type fq_poly_struct
```

```
type fq_poly_t
```

11.6.2 Memory management

```
void fq_poly_init(fq_poly_t poly, const fq_ctx_t ctx)
```

Initialises **poly** for use, with context **ctx**, and setting its length to zero. A corresponding call to **fq_poly_clear()** must be made after finishing with the **fq_poly_t** to free the memory used by the polynomial.

```
void fq_poly_init2(fq_poly_t poly, slong alloc, const fq_ctx_t ctx)
```

Initialises **poly** with space for at least **alloc** coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to **fq_poly_clear()** must be made after finishing with the **fq_poly_t** to free the memory used by the polynomial.

void **fq_poly_realloc**(*fq_poly_t* poly, *slong* alloc, const *fq_ctx_t* ctx)

Reallocates the given polynomial to have space for *alloc* coefficients. If *alloc* is zero the polynomial is cleared and then reinitialised. If the current length is greater than *alloc* the polynomial is first truncated to length *alloc*.

void **fq_poly_fit_length**(*fq_poly_t* poly, *slong* len, const *fq_ctx_t* ctx)

If *len* is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least *len* coefficients. No data is lost when calling this function.

The function efficiently deals with the case where **fit_length** is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void **_fq_poly_set_length**(*fq_poly_t* poly, *slong* newlen, const *fq_ctx_t* ctx)

Sets the coefficients of *poly* beyond *len* to zero and sets the length of *poly* to *len*.

void **fq_poly_clear**(*fq_poly_t* poly, const *fq_ctx_t* ctx)

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void **_fq_poly_normalise**(*fq_poly_t* poly, const *fq_ctx_t* ctx)

Sets the length of *poly* so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **_fq_poly_normalise2**(const *fq_struct* *poly, *slong* *length, const *fq_ctx_t* ctx)

Sets the length *length* of (*poly*, *length*) so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **fq_poly_truncate**(*fq_poly_t* poly, *slong* newlen, const *fq_ctx_t* ctx)

Truncates the polynomial to length at most *n*.

void **fq_poly_set_trunc**(*fq_poly_t* poly1, *fq_poly_t* poly2, *slong* newlen, const *fq_ctx_t* ctx)

Sets *poly1* to *poly2* truncated to length *n*.

void **_fq_poly_reverse**(*fq_struct* *output, const *fq_struct* *input, *slong* len, *slong* m, const *fq_ctx_t* ctx)

Sets *output* to the reverse of *input*, which is of length *len*, but thinking of it as a polynomial of length *m*, notionally zero-padded if necessary. The length *m* must be non-negative, but there are no other restrictions. The polynomial *output* must have space for *m* coefficients.

void **fq_poly_reverse**(*fq_poly_t* output, const *fq_poly_t* input, *slong* m, const *fq_ctx_t* ctx)

Sets *output* to the reverse of *input*, thinking of it as a polynomial of length *m*, notionally zero-padded if necessary). The length *m* must be non-negative, but there are no other restrictions. The output polynomial will be set to length *m* and then normalised.

11.6.3 Polynomial parameters

slong **fq_poly_degree**(const *fq_poly_t* poly, const *fq_ctx_t* ctx)

Returns the degree of the polynomial *poly*.

slong **fq_poly_length**(const *fq_poly_t* poly, const *fq_ctx_t* ctx)

Returns the length of the polynomial *poly*.

fq_struct ***fq_poly_lead**(const *fq_poly_t* poly, const *fq_ctx_t* ctx)

Returns a pointer to the leading coefficient of *poly*, or NULL if *poly* is the zero polynomial.

11.6.4 Randomisation

void **fq_poly_randtest**(*fq_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_ctx_t* ctx)
Sets *f* to a random polynomial of length at most *len* with entries in the field described by *ctx*.

void **fq_poly_randtest_not_zero**(*fq_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_ctx_t* ctx)
Same as **fq_poly_randtest** but guarantees that the polynomial is not zero.

void **fq_poly_randtest_monic**(*fq_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_ctx_t* ctx)
Sets *f* to a random monic polynomial of length *len* with entries in the field described by *ctx*.

void **fq_poly_randtest_irreducible**(*fq_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_ctx_t* ctx)
Sets *f* to a random monic, irreducible polynomial of length *len* with entries in the field described by *ctx*.

11.6.5 Assignment and basic manipulation

void **_fq_poly_set**(*fq_struct* *rop, const *fq_struct* *op, *slong* len, const *fq_ctx_t* ctx)
Sets (rop, len) to (op, len).

void **fq_poly_set**(*fq_poly_t* poly1, const *fq_poly_t* poly2, const *fq_ctx_t* ctx)
Sets the polynomial poly1 to the polynomial poly2.

void **fq_poly_set_fq**(*fq_poly_t* poly, const *fq_t* c, const *fq_ctx_t* ctx)
Sets the polynomial poly to c.

void **fq_poly_set_fmpz_mod_poly**(*fq_poly_t* rop, const *fmpz_mod_poly_t* op, const *fq_ctx_t* ctx)
Sets the polynomial rop to the polynomial op

void **fq_poly_set_nmod_poly**(*fq_poly_t* rop, const *nmod_poly_t* op, const *fq_ctx_t* ctx)
Sets the polynomial rop to the polynomial op

void **fq_poly_swap**(*fq_poly_t* op1, *fq_poly_t* op2, const *fq_ctx_t* ctx)
Swaps the two polynomials op1 and op2.

void **_fq_poly_zero**(*fq_struct* *rop, *slong* len, const *fq_ctx_t* ctx)
Sets (rop, len) to the zero polynomial.

void **fq_poly_zero**(*fq_poly_t* poly, const *fq_ctx_t* ctx)
Sets poly to the zero polynomial.

void **fq_poly_one**(*fq_poly_t* poly, const *fq_ctx_t* ctx)
Sets poly to the constant polynomial 1.

void **fq_poly_gen**(*fq_poly_t* poly, const *fq_ctx_t* ctx)
Sets poly to the polynomial *x*.

void **fq_poly_make_monic**(*fq_poly_t* rop, const *fq_poly_t* op, const *fq_ctx_t* ctx)
Sets rop to op, normed to have leading coefficient 1.

void **_fq_poly_make_monic**(*fq_struct* *rop, const *fq_struct* *op, *slong* length, const *fq_ctx_t* ctx)
Sets rop to (op, length), normed to have leading coefficient 1. Assumes that rop has enough space for the polynomial, assumes that op is not zero (and thus has an invertible leading coefficient).

11.6.6 Getting and setting coefficients

void **fq_poly_get_coeff**(*fq_t* x, const *fq_poly_t* poly, *slong* n, const *fq_ctx_t* ctx)

Sets x to the coefficient of X^n in poly .

void **fq_poly_set_coeff**(*fq_poly_t* poly, *slong* n, const *fq_t* x, const *fq_ctx_t* ctx)

Sets the coefficient of X^n in poly to x .

void **fq_poly_set_coeff_fmpz**(*fq_poly_t* poly, *slong* n, const *fmpz_t* x, const *fq_ctx_t* ctx)

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

11.6.7 Comparison

int **fq_poly_equal**(const *fq_poly_t* poly1, const *fq_poly_t* poly2, const *fq_ctx_t* ctx)

Returns nonzero if the two polynomials poly1 and poly2 are equal, otherwise returns zero.

int **fq_poly_equal_trunc**(const *fq_poly_t* poly1, const *fq_poly_t* poly2, *slong* n, const *fq_ctx_t* ctx)

Notionally truncate poly1 and poly2 to length n and return nonzero if they are equal, otherwise return zero.

int **fq_poly_is_zero**(const *fq_poly_t* poly, const *fq_ctx_t* ctx)

Returns whether the polynomial poly is the zero polynomial.

int **fq_poly_is_one**(const *fq_poly_t* op, const *fq_ctx_t* ctx)

Returns whether the polynomial poly is equal to the constant polynomial 1.

int **fq_poly_is_gen**(const *fq_poly_t* op, const *fq_ctx_t* ctx)

Returns whether the polynomial poly is equal to the polynomial x .

int **fq_poly_is_unit**(const *fq_poly_t* op, const *fq_ctx_t* ctx)

Returns whether the polynomial poly is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

int **fq_poly_equal_fq**(const *fq_poly_t* poly, const *fq_t* c, const *fq_ctx_t* ctx)

Returns whether the polynomial poly is equal the (constant) $\mathbf{F}_q$ element c

11.6.8 Addition and subtraction

void **_fq_poly_add**(*fq_struct* *res, const *fq_struct* *poly1, *slong* len1, const *fq_struct* *poly2, *slong* len2, const *fq_ctx_t* ctx)

Sets res to the sum of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$.

void **fq_poly_add**(*fq_poly_t* res, const *fq_poly_t* poly1, const *fq_poly_t* poly2, const *fq_ctx_t* ctx)

Sets res to the sum of poly1 and poly2 .

void **fq_poly_add_si**(*fq_poly_t* res, const *fq_poly_t* poly1, *slong* c, const *fq_ctx_t* ctx)

Sets res to the sum of poly1 and c .

void **fq_poly_add_series**(*fq_poly_t* res, const *fq_poly_t* poly1, const *fq_poly_t* poly2, *slong* n, const *fq_ctx_t* ctx)

Notionally truncate poly1 and poly2 to length n and set res to the sum.

void **_fq_poly_sub**(*fq_struct* *res, const *fq_struct* *poly1, *slong* len1, const *fq_struct* *poly2, *slong* len2, const *fq_ctx_t* ctx)

Sets res to the difference of $(\text{poly1}, \text{len1})$ and $(\text{poly2}, \text{len2})$.

void **fq_poly_sub**(*fq_poly_t* res, const *fq_poly_t* poly1, const *fq_poly_t* poly2, const *fq_ctx_t* ctx)

Sets res to the difference of poly1 and poly2 .

```
void fq_poly_sub_series(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, slong n, const
fq_ctx_t ctx)
```

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the difference.

```
void _fq_poly_neg(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Sets `rop` to the additive inverse of `(poly, len)`.

```
void fq_poly_neg(fq_poly_t res, const fq_poly_t poly, const fq_ctx_t ctx)
```

Sets `res` to the additive inverse of `poly`.

11.6.9 Scalar multiplication and division

```
void _fq_poly_scalar_mul_fq(fq_struct *rop, const fq_struct *op, slong len, const fq_t x, const
fq_ctx_t ctx)
```

Sets `(rop, len)` to the product of `(op, len)` by the scalar `x`, in the context defined by `ctx`.

```
void fq_poly_scalar_mul_fq(fq_poly_t rop, const fq_poly_t op, const fq_t x, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_poly_scalar_addmul_fq(fq_struct *rop, const fq_struct *op, slong len, const fq_t x, const
fq_ctx_t ctx)
```

Adds to `(rop, len)` the product of `(op, len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_poly_scalar_addmul_fq(fq_poly_t rop, const fq_poly_t op, const fq_t x, const fq_ctx_t
ctx)
```

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_poly_scalar_submul_fq(fq_struct *rop, const fq_struct *op, slong len, const fq_t x, const
fq_ctx_t ctx)
```

Subtracts from `(rop, len)` the product of `(op, len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

```
void fq_poly_scalar_submul_fq(fq_poly_t rop, const fq_poly_t op, const fq_t x, const fq_ctx_t
ctx)
```

Subtracts from `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_poly_scalar_div_fq(fq_struct *rop, const fq_struct *op, slong len, const fq_t x, const
fq_ctx_t ctx)
```

Sets `(rop, len)` to the quotient of `(op, len)` by the scalar `x`, in the context defined by `ctx`. An exception is raised if `x` is zero.

```
void fq_poly_scalar_div_fq(fq_poly_t rop, const fq_poly_t op, const fq_t x, const fq_ctx_t ctx)
```

Sets `rop` to the quotient of `op` by the scalar `x`, in the context defined by `ctx`. An exception is raised if `x` is zero.

11.6.10 Multiplication

```
void _fq_poly_mul_classical(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2,
slong len2, const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` is at least `len2` and neither is zero.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_poly_mul_classical(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const
fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using classical polynomial multiplication.

```
void _fq_poly_mul_reorder(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2,
                        slong len2, const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, assuming that `len1` and `len2` are non-zero.

Permits zero padding. Supports aliasing.

```
void fq_poly_mul_reorder(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t
                        ctx)
```

Sets `rop` to the product of `op1` and `op2`, reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Suppose $\mathbf{F}_q = \mathbf{F}_p[X]/(f(X))$ and recall that elements of $\mathbf{F}_q$ are internally represented by elements of type `mpz_poly`. For small degree extensions but polynomials in $\mathbf{F}_q[Y]$ of large degree n , we change the representation to

$$\begin{aligned} g(Y) &= \sum_{i=0}^n a_i(X) Y^i \\ &= \sum_{j=0}^d \sum_{i=0}^n \text{Coeff}(a_i(X), j) Y^i. \end{aligned}$$

This allows us to use a poor algorithm (such as classical multiplication) in the X -direction and leverage the existing fast integer multiplication routines in the Y -direction where the polynomial degree n is large.

```
void _fq_poly_mul_univariate(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct
                           *op2, slong len2, const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_poly_mul_univariate(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const
                           fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using a bivariate to univariate transformation and reducing this problem to multiplying two univariate polynomials.

```
void _fq_poly_mul_KS(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong
                    len2, const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_poly_mul_KS(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_poly_mul(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2,
                  const fq_ctx_t ctx)
```

Sets `(rop, len1 + len2 - 1)` to the product of `(op1, len1)` and `(op2, len2)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_poly_mul(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, choosing an appropriate algorithm.

```
void _fq_poly_mullo_classical(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct
                             *op2, slong len2, slong n, const fq_ctx_t ctx)
```

Sets `(rop, n)` to the first n coefficients of `(op1, len1)` multiplied by `(op2, len2)`.

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Assumes neither `len1` nor `len2` is zero.

```
void fq_poly_mullo_classical(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, slong n,
                             const fq_ctx_t ctx)
```

Sets `rop` to the product of `poly1` and `poly2`, computed using the classical or schoolbook method.

```
void _fq_poly_mullo_univariate(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct
                               *op2, slong len2, slong n, const fq_ctx_t ctx)
```

Sets `(rop, n)` to the lowest n coefficients of the product of `(op1, len1)` and `(op2, len2)`, computed using a bivariate to univariate transformation.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between `res`, `poly1` and `poly2`.

```
void fq_poly_mullo_univariate(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, slong n,
                              const fq_ctx_t ctx)
```

Sets `rop` to the lowest n coefficients of the product of `op1` and `op2`, computed using a bivariate to univariate transformation.

```
void _fq_poly_mullo_KS(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2,
                      slong len2, slong n, const fq_ctx_t ctx)
```

Sets `(rop, n)` to the lowest n coefficients of the product of `(op1, len1)` and `(op2, len2)`.

Assumes that `len1` and `len2` are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between `rop`, `op1` and `op2`.

```
void fq_poly_mullo_KS(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, slong n, const
                      fq_ctx_t ctx)
```

Sets `rop` to the lowest n coefficients of the product of `op1` and `op2`.

```
void _fq_poly_mullo(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong
                    len2, slong n, const fq_ctx_t ctx)
```

Sets `(rop, n)` to the lowest n coefficients of the product of `(op1, len1)` and `(op2, len2)`.

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fq_poly_mullo(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, slong n, const
                   fq_ctx_t ctx)
```

Sets `rop` to the lowest n coefficients of the product of `op1` and `op2`.

```
void _fq_poly_mulhigh_classical(fq_struct *res, const fq_struct *poly1, slong len1, const fq_struct
                               *poly2, slong len2, slong start, const fq_ctx_t ctx)
```

Computes the product of `(poly1, len1)` and `(poly2, len2)` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Assumes that `len1` $\geq$ `len2` $>$ 0. Aliasing of inputs and output is not permitted. Algorithm is classical multiplication.

```
void fq_poly_mulhigh_classical(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, slong
                              start, const fq_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Algorithm is classical multiplication.

```
void _fq_poly_mulhigh(fq_struct *res, const fq_struct *poly1, slong len1, const fq_struct *poly2,
                      slong len2, slong start, fq_ctx_t ctx)
```

Computes the product of `(poly1, len1)` and `(poly2, len2)` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Assumes that `len1` $\geq$ `len2` $>$ 0. Aliasing of inputs and output is not permitted.

```
void fq_poly_mulhigh(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, slong start, const
fq_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced.

```
void _fq_poly_mulmod(fq_struct *res, const fq_struct *poly1, slong len1, const fq_struct *poly2,
slong len2, const fq_struct *f, slong lenf, const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that $\text{len1} + \text{len2} - \text{lenf} > 0$, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void fq_poly_mulmod(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, const fq_poly_t f,
const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fq_poly_mulmod_preinv(fq_struct *res, const fq_struct *poly1, slong len1, const fq_struct
*poly2, slong len2, const fq_struct *f, slong lenf, const fq_struct *finv,
slong lenfinv, const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod x^{lenf} .

Aliasing of `res` with any of the inputs is not permitted.

```
void fq_poly_mulmod_preinv(fq_poly_t res, const fq_poly_t poly1, const fq_poly_t poly2, const
fq_poly_t f, const fq_poly_t finv, const fq_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

11.6.11 Squaring

```
void _fq_poly_sqr_classical(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op, len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_poly_sqr_classical(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
```

Sets `rop` to the square of `op` using classical polynomial multiplication.

```
void _fq_poly_sqr_reorder(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2*len-1)` to the square of `(op, len)`, assuming that `len` is not zero reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Permits zero padding. Supports aliasing.

```
void fq_poly_sqr_reorder(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
```

Sets `rop` to the square of `op`, assuming that `len` is not zero reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$. See `fq_poly_mul_reorder`.

```
void _fq_poly_sqr_KS(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

void **fq_poly_sqr_KS**(*fq_poly_t* rop, const *fq_poly_t* op, const *fq_ctx_t* ctx)
Sets *rop* to the square of *op* using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

void **_fq_poly_sqr**(*fq_struct* *rop, const *fq_struct* *op, *slong* len, const *fq_ctx_t* ctx)
Sets (*rop*, $2 * \text{len} - 1$) to the square of (*op*, *len*), choosing an appropriate algorithm.
Permits zero padding. Does not support aliasing.

void **fq_poly_sqr**(*fq_poly_t* rop, const *fq_poly_t* op, const *fq_ctx_t* ctx)
Sets *rop* to the square of *op*, choosing an appropriate algorithm.

11.6.12 Powering

void **_fq_poly_pow**(*fq_struct* *rop, const *fq_struct* *op, *slong* len, *ulong* e, const *fq_ctx_t* ctx)
Sets *rop* = op^e , assuming that *e*, *len* > 0 and that *rop* has space for $e * (\text{len} - 1) + 1$ coefficients. Does not support aliasing.

void **fq_poly_pow**(*fq_poly_t* rop, const *fq_poly_t* op, *ulong* e, const *fq_ctx_t* ctx)
Computes *rop* = op^e . If *e* is zero, returns one, so that in particular $0^0 = 1$.

void **_fq_poly_powmod_ui_binexp**(*fq_struct* *res, const *fq_struct* *poly, *ulong* e, const *fq_struct* *f, *slong* lenf, const *fq_ctx_t* ctx)
Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require *e* > 0.
We require *lenf* > 1. It is assumed that *poly* is already reduced modulo *f* and zero-padded as necessary to have length exactly *lenf* - 1. The output *res* must have room for *lenf* - 1 coefficients.

void **fq_poly_powmod_ui_binexp**(*fq_poly_t* res, const *fq_poly_t* poly, *ulong* e, const *fq_poly_t* f, const *fq_ctx_t* ctx)
Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require *e* >= 0.

void **_fq_poly_powmod_ui_binexp_preinv**(*fq_struct* *res, const *fq_struct* *poly, *ulong* e, const *fq_struct* *f, *slong* lenf, const *fq_struct* *finv, *slong* lenfinv, const *fq_ctx_t* ctx)
Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require *e* > 0.
We require *finv* to be the inverse of the reverse of *f*.
We require *lenf* > 1. It is assumed that *poly* is already reduced modulo *f* and zero-padded as necessary to have length exactly *lenf* - 1. The output *res* must have room for *lenf* - 1 coefficients.

void **fq_poly_powmod_ui_binexp_preinv**(*fq_poly_t* res, const *fq_poly_t* poly, *ulong* e, const *fq_poly_t* f, const *fq_poly_t* finv, const *fq_ctx_t* ctx)
Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require *e* >= 0. We require *finv* to be the inverse of the reverse of *f*.

void **_fq_poly_powmod_fmpz_binexp**(*fq_struct* *res, const *fq_struct* *poly, const *fmpz_t* e, const *fq_struct* *f, *slong* lenf, const *fq_ctx_t* ctx)
Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require *e* > 0.
We require *lenf* > 1. It is assumed that *poly* is already reduced modulo *f* and zero-padded as necessary to have length exactly *lenf* - 1. The output *res* must have room for *lenf* - 1 coefficients.

void **fq_poly_powmod_fmpz_binexp**(*fq_poly_t* res, const *fq_poly_t* poly, const *fmpz_t* e, const *fq_poly_t* f, const *fq_ctx_t* ctx)
Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require *e* >= 0.


```
void _fq_poly_powmod_fmpz_binexp_preinv(fq_struct *res, const fq_struct *poly, const fmpz_t e,
                                       const fq_struct *f, slong lenf, const fq_struct *finv,
                                       slong lenfinv, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_fmpz_binexp_preinv(fq_poly_t res, const fq_poly_t poly, const fmpz_t e,
                                       const fq_poly_t f, const fq_poly_t finv, const fq_ctx_t
                                       ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_poly_powmod_fmpz_sliding_preinv(fq_struct *res, const fq_struct *poly, const fmpz_t e,
                                       ulong k, const fq_struct *f, slong lenf, const fq_struct
                                       *finv, slong lenfinv, const fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an “optimum” size will be selected automatically base on `e`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_fmpz_sliding_preinv(fq_poly_t res, const fq_poly_t poly, const fmpz_t e,
                                       ulong k, const fq_poly_t f, const fq_poly_t finv, const
                                       fq_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an “optimum” size will be selected automatically base on `e`.

```
void _fq_poly_powmod_x_fmpz_preinv(fq_struct *res, const fmpz_t e, const fq_struct *f, slong lenf,
                                   const fq_struct *finv, slong lenfinv, const fq_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_poly_powmod_x_fmpz_preinv(fq_poly_t res, const fmpz_t e, const fq_poly_t f, const
                                   fq_poly_t finv, const fq_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_poly_pow_trunc_binexp(fq_struct *res, const fq_struct *poly, ulong e, slong trunc, const
                               fq_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void fq_poly_pow_trunc_binexp(fq_poly_t res, const fq_poly_t poly, ulong e, slong trunc, const
                               fq_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```
void _fq_poly_pow_trunc(fq_struct *res, const fq_struct *poly, ulong e, slong trunc, const fq_ctx_t
                        mod)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length

`trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc` > 0 and that `e` > 1 . Aliasing is not permitted.

```
void fq_poly_pow_trunc(fq_poly_t res, const fq_poly_t poly, ulong e, slong trunc, const fq_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.

11.6.13 Shifting

```
void _fq_poly_shift_left(fq_struct *rop, const fq_struct *op, slong len, slong n, const fq_ctx_t ctx)
```

Sets `(rop, len + n)` to `(op, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `rop` fits `len + n` elements. Supports aliasing between `rop` and `op`.

```
void fq_poly_shift_left(fq_poly_t rop, const fq_poly_t op, slong n, const fq_ctx_t ctx)
```

Sets `rop` to `op` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fq_poly_shift_right(fq_struct *rop, const fq_struct *op, slong len, slong n, const fq_ctx_t ctx)
```

Sets `(rop, len - n)` to `(op, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len` $> n$, and that `rop` fits `len - n` elements. Supports aliasing between `rop` and `op`, although in this case the top coefficients of `op` are not set to zero.

```
void fq_poly_shift_right(fq_poly_t rop, const fq_poly_t op, slong n, const fq_ctx_t ctx)
```

Sets `rop` to `op` shifted right by n coefficients. If n is equal to or greater than the current length of `op`, `rop` is set to the zero polynomial.

11.6.14 Norms

```
slong _fq_poly_hamming_weight(const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Returns the number of non-zero entries in `(op, len)`.

```
slong fq_poly_hamming_weight(const fq_poly_t op, const fq_ctx_t ctx)
```

Returns the number of non-zero entries in the polynomial `op`.

11.6.15 Euclidean division

```
void _fq_poly_divrem(fq_struct *Q, fq_struct *R, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB, const fq_t invB, const fq_ctx_t ctx)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (R, lenA) such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

```
void fq_poly_divrem(fq_poly_t Q, fq_poly_t R, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void fq_poly_divrem_f(fq_t f, fq_poly_t Q, fq_poly_t R, const fq_poly_t A, const fq_poly_t B,
                    const fq_ctx_t ctx)
```

Either finds a non-trivial factor f of the modulus of `ctx`, or computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

If the leading coefficient of B is invertible, the division with remainder operation is carried out, Q and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of the modulus and Q and R are not touched.

Assumes that B is non-zero.

```
void _fq_poly_rem(fq_struct *R, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB,
                const fq_t invB, const fq_ctx_t ctx)
```

Sets R to the remainder of the division of (A, lenA) by (B, lenB) . Assumes that the leading coefficient of (B, lenB) is invertible and that `invB` is its inverse.

```
void fq_poly_rem(fq_poly_t R, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Sets R to the remainder of the division of A by B in the context described by `ctx`.

```
void _fq_poly_div(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB,
                const fq_t invB, const fq_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(Q, \text{lenA} - \text{lenB} + 1)$. Allows zero-padding in A but not in B . Assumes that the leading coefficient of B is a unit.

```
void fq_poly_div(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q . If $\text{len}(B) = 0$ an exception is raised.

```
void _fq_poly_div_newton_n_preinv(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct
                                *B, slong lenB, const fq_struct *Binv, slong lenBinv, const
                                fq_ctx_t ctx)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`, but return only Q .

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fq_poly_div_newton_n_preinv(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, const
                                fq_poly_t Binv, const fq_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q .

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \times \text{the length of } B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _fq_poly_divrem_newton_n_preinv(fq_struct *Q, fq_struct *R, const fq_struct *A, slong lenA,
                                    const fq_struct *B, slong lenB, const fq_struct *Binv, slong
                                    lenBinv, const fq_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`. We require that Q have space for `lenA - lenB + 1` coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$. The algorithm used is to call `div_newton_n_preinv()` and then multiply out and compute the remainder.

```
void fq_poly_divrem_newton_n_preinv(fq_poly_t Q, fq_poly_t R, const fq_poly_t A, const
                                   fq_poly_t B, const fq_poly_t Binv, const fq_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \times \text{the length of } B - 2$.

The algorithm used is to call `div_newton_n()` and then multiply out and compute the remainder.

```
void _fq_poly_inv_series_newton(fq_struct *Qinv, const fq_struct *Q, slong n, const fq_t cinv,
                               const fq_ctx_t ctx)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length n such that $Q * Qinv$ is 1 modulo x^n . Requires $n > 0$. This function can be viewed as inverting a power series via Newton iteration.

```
void fq_poly_inv_series_newton(fq_poly_t Qinv, const fq_poly_t Q, slong n, const fq_ctx_t ctx)
```

Given Q find $Qinv$ such that $Q * Qinv$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series via Newton iteration.

```
void _fq_poly_inv_series(fq_struct *Qinv, const fq_struct *Q, slong n, const fq_t cinv, const
                        fq_ctx_t ctx)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length n such that $Q * Qinv$ is 1 modulo x^n . Requires $n > 0$.

```
void fq_poly_inv_series(fq_poly_t Qinv, const fq_poly_t Q, slong n, const fq_ctx_t ctx)
```

Given Q find $Qinv$ such that $Q * Qinv$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$.

```
void _fq_poly_div_series(fq_struct *Q, const fq_struct *A, slong Alen, const fq_struct *B, slong
                        Blen, slong n, const fq_ctx_t ctx)
```

Set (Q, n) to the quotient of the series (A, Alen) and (B, Blen) assuming $\text{Alen}, \text{Blen} \leq n$. We assume the bottom coefficient of B is invertible.

```
void fq_poly_div_series(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, slong n, const
                        fq_ctx_t ctx)
```

Set Q to the quotient of the series A by B , thinking of the series as though they were of length n . We assume that the bottom coefficient of B is invertible.

11.6.16 Greatest common divisor

```
void fq_poly_gcd(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

```
slong _fq_poly_gcd(fq_struct *G, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB,
                  const fq_ctx_t ctx)
```

Computes the GCD of A of length `lenA` and B of length `lenB`, where `lenA` $\geq$ `lenB` > 0 and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for `lenB` coefficients.

```
slong _fq_poly_gcd_euclidean_f(fq_t f, fq_struct *G, const fq_struct *A, slong lenA, const
                              fq_struct *B, slong lenB, const fq_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length, or sets f to a non-trivial factor of the modulus of `ctx` and leaves the contents of the vector $(G, \text{len}B)$ undefined.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

```
void fq_poly_gcd_euclidean_f(fq_t f, fq_poly_t G, const fq_poly_t A, const fq_poly_t B, const
fq_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of A and B or sets f to a factor of the modulus of ctx .

```
slong _fq_poly_xgcd(fq_struct *G, fq_struct *S, fq_struct *T, const fq_struct *A, slong lenA, const
fq_struct *B, slong lenB, const fq_ctx_t ctx)
```

Computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fq_poly_xgcd(fq_poly_t G, fq_poly_t S, fq_poly_t T, const fq_poly_t A, const fq_poly_t B,
const fq_ctx_t ctx)
```

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S*A + T*B = G$. The length of S will be at most lenB and the length of T will be at most lenA .

```
slong _fq_poly_xgcd_euclidean_f(fq_t f, fq_struct *G, fq_struct *S, fq_struct *T, const fq_struct
*A, slong lenA, const fq_struct *B, slong lenB, const fq_ctx_t
ctx)
```

Either sets $f = 1$ and computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$; otherwise, sets f to a non-trivial factor of the modulus of ctx and leaves G , S , and T undefined. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B)$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fq_poly_xgcd_euclidean_f(fq_t f, fq_poly_t G, fq_poly_t S, fq_poly_t T, const fq_poly_t A,
const fq_poly_t B, const fq_ctx_t ctx)
```

Either sets $f = 1$ and computes the GCD of A and B or sets f to a non-trivial factor of the modulus of ctx .

If the GCD is computed, polynomials S and T are computed such that $S*A + T*B = G$; otherwise, they are undefined. The length of S will be at most lenB and the length of T will be at most lenA .

The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

11.6.17 Divisibility testing

```
int _fq_poly_divides(fq_struct *Q, const fq_struct *A, slong lenA, const fq_struct *B, slong lenB,
                    const fq_t invB, const fq_ctx_t ctx)
```

Returns 1 if $(B, \text{len}B)$ divides $(A, \text{len}A)$ exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\text{len}(A) \geq \text{len}(B) > 0$ and that Q has space for $\text{len}(A) - \text{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fq_poly_divides(fq_poly_t Q, const fq_poly_t A, const fq_poly_t B, const fq_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

11.6.18 Derivative

```
void _fq_poly_derivative(fq_struct *rop, const fq_struct *op, slong len, const fq_ctx_t ctx)
```

Sets $(\text{rop}, \text{len} - 1)$ to the derivative of (op, len) . Also handles the cases where len is 0 or 1 correctly. Supports aliasing of rop and op .

```
void fq_poly_derivative(fq_poly_t rop, const fq_poly_t op, const fq_ctx_t ctx)
```

Sets rop to the derivative of op .

11.6.19 Square root

```
void _fq_poly_invsqrt_series(fq_struct *g, const fq_struct *h, slong n, fq_ctx_t mod)
```

Set the first n terms of g to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n . Aliasing is not permitted.

```
void fq_poly_invsqrt_series(fq_poly_t g, const fq_poly_t h, slong n, fq_ctx_t mod)
```

Set g to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
void _fq_poly_sqrt_series(fq_struct *g, const fq_struct *h, slong n, fq_ctx_t ctx)
```

Set the first n terms of g to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n . Aliasing is not permitted.

```
void fq_poly_sqrt_series(fq_poly_t g, const fq_poly_t h, slong n, fq_ctx_t ctx)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
int _fq_poly_sqrt(fq_struct *s, const fq_struct *p, slong n, fq_ctx_t mod)
```

If (p, n) is a perfect square, sets $(s, n / 2 + 1)$ to a square root of p and returns 1. Otherwise returns 0.

```
int fq_poly_sqrt(fq_poly_t s, const fq_poly_t p, fq_ctx_t mod)
```

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

11.6.20 Evaluation

```
void _fq_poly_evaluate_fq(fq_t rop, const fq_struct *op, slong len, const fq_t a, const fq_ctx_t ctx)
```

Sets `rop` to (op, len) evaluated at a .

Supports zero padding. There are no restrictions on `len`, that is, `len` is allowed to be zero, too.

```
void fq_poly_evaluate_fq(fq_t rop, const fq_poly_t f, const fq_t a, const fq_ctx_t ctx)
```

Sets `rop` to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

11.6.21 Composition

```
void _fq_poly_compose(fq_struct *rop, const fq_struct *op1, slong len1, const fq_struct *op2, slong len2, const fq_ctx_t ctx)
```

Sets `rop` to the composition of $(op1, len1)$ and $(op2, len2)$.

Assumes that `rop` has space for $(len1-1)*(len2-1) + 1$ coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_poly_compose(fq_poly_t rop, const fq_poly_t op1, const fq_poly_t op2, const fq_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f , g , and h , respectively, sets $f(t) = g(h(t))$.

```
void _fq_poly_compose_mod_horner(fq_struct *res, const fq_struct *f, slong lenf, const fq_struct *g, const fq_struct *h, slong lenh, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_poly_compose_mod_horner(fq_poly_t res, const fq_poly_t f, const fq_poly_t g, const fq_poly_t h, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fq_poly_compose_mod_horner_preinv(fq_struct *res, const fq_struct *f, slong lenf, const fq_struct *g, const fq_struct *h, slong lenh, const fq_struct *hinv, slong lenhiv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_poly_compose_mod_horner_preinv(fq_poly_t res, const fq_poly_t f, const fq_poly_t g, const fq_poly_t h, const fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is Horner's rule.

```
void _fq_poly_compose_mod_brent_kung(fq_struct *res, const fq_struct *f, slong lenf, const fq_struct *g, const fq_struct *h, slong lenh, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g

is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_poly_compose_mod_brent_kung(fq_poly_t res, const fq_poly_t f, const fq_poly_t g, const
    fq_poly_t h, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_poly_compose_mod_brent_kung_preinv(fq_struct *res, const fq_struct *f, slong lenf, const
    fq_struct *g, const fq_struct *h, slong lenh, const
    fq_struct *hinv, slong lenhiv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_poly_compose_mod_brent_kung_preinv(fq_poly_t res, const fq_poly_t f, const fq_poly_t g,
    const fq_poly_t h, const fq_poly_t hinv, const
    fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_poly_compose_mod(fq_struct *res, const fq_struct *f, slong lenf, const fq_struct *g, const
    fq_struct *h, slong lenh, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fq_poly_compose_mod(fq_poly_t res, const fq_poly_t f, const fq_poly_t g, const fq_poly_t h,
    const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero.

```
void _fq_poly_compose_mod_preinv(fq_struct *res, const fq_struct *f, slong lenf, const fq_struct *g,
    const fq_struct *h, slong lenh, const fq_struct *hinv, slong
    lenhiv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

```
void fq_poly_compose_mod_preinv(fq_poly_t res, const fq_poly_t f, const fq_poly_t g, const
    fq_poly_t h, const fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h .

```
void _fq_poly_reduce_matrix_mod_poly(fq_mat_t A, const fq_mat_t B, const fq_poly_t f, const
    fq_ctx_t ctx)
```

Sets the i th row of A to the reduction of the i th row of B modulo f for $i = 1, \dots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fq_poly_precompute_matrix(fq_mat_t A, const fq_struct *f, const fq_struct *g, slong leng,
    const fq_struct *ginv, slong lenginv, const fq_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g and g to be nonzero.


```
void fq_poly_precompute_matrix(fq_mat_t A, const fq_poly_t f, const fq_poly_t g, const
                             fq_poly_t ginv, const fq_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require $ginv$ to be the inverse of the reverse of g .

```
void _fq_poly_compose_mod_brent_kung_precomp_preinv(fq_struct *res, const fq_struct *f, slong
                                                    lenf, const fq_mat_t A, const fq_struct
                                                    *h, slong lenh, const fq_struct *hinv,
                                                    slong lenhinv, const fq_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero. We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h . Furthermore, we require **hinv** to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_poly_compose_mod_brent_kung_precomp_preinv(fq_poly_t res, const fq_poly_t f, const
                                                    fq_mat_t A, const fq_poly_t h, const
                                                    fq_poly_t hinv, const fq_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h . Furthermore, we require **hinv** to be the inverse of the reverse of h . This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h .

11.6.22 Output

```
int _fq_poly_fprint_pretty(FILE *file, const fq_struct *poly, slong len, const char *x, const
                           fq_ctx_t ctx)
```

Prints the pretty representation of **(poly, len)** to the stream **file**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_fprint_pretty(FILE *file, const fq_poly_t poly, const char *x, const fq_ctx_t ctx)
```

Prints the pretty representation of **poly** to the stream **file**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_poly_print_pretty(const fq_struct *poly, slong len, const char *x, const fq_ctx_t ctx)
```

Prints the pretty representation of **(poly, len)** to **stdout**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_print_pretty(const fq_poly_t poly, const char *x, const fq_ctx_t ctx)
```

Prints the pretty representation of **poly** to **stdout**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_poly_fprint(FILE *file, const fq_struct *poly, slong len, const fq_ctx_t ctx)
```

Prints the pretty representation of **(poly, len)** to the stream **file**.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_poly_fprint(FILE *file, const fq_poly_t poly, const fq_ctx_t ctx)
```

Prints the pretty representation of **poly** to the stream **file**.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_poly_print**(const *fq_struct* *poly, *slong* len, const *fq_ctx_t* ctx)

Prints the pretty representation of (poly, len) to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_poly_print**(const *fq_poly_t* poly, const *fq_ctx_t* ctx)

Prints the representation of poly to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

char ***fq_poly_get_str**(const *fq_struct* *poly, *slong* len, const *fq_ctx_t* ctx)

Returns the plain FLINT string representation of the polynomial (poly, len).

char ***fq_poly_get_str**(const *fq_poly_t* poly, const *fq_ctx_t* ctx)

Returns the plain FLINT string representation of the polynomial poly.

char ***fq_poly_get_str_pretty**(const *fq_struct* *poly, *slong* len, const char *x, const *fq_ctx_t* ctx)

Returns a pretty representation of the polynomial (poly, len) using the null-terminated string x as the variable name.

char ***fq_poly_get_str_pretty**(const *fq_poly_t* poly, const char *x, const *fq_ctx_t* ctx)

Returns a pretty representation of the polynomial poly using the null-terminated string x as the variable name

11.6.23 Inflation and deflation

void **fq_poly_inflate**(*fq_poly_t* result, const *fq_poly_t* input, *ulong* inflation, const *fq_ctx_t* ctx)

Sets result to the inflated polynomial $p(x^n)$ where p is given by input and n is given by inflation.

void **fq_poly_deflate**(*fq_poly_t* result, const *fq_poly_t* input, *ulong* deflation, const *fq_ctx_t* ctx)

Sets result to the deflated polynomial $p(x^{1/n})$ where p is given by input and n is given by deflation. Requires $n > 0$.

ulong **fq_poly_deflation**(const *fq_poly_t* input, const *fq_ctx_t* ctx)

Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 if input is a constant polynomial.

11.7 fq_default_poly.h – univariate polynomials over finite fields

11.7.1 Types, macros and constants

type **fq_default_poly_t**

11.7.2 Memory management

void **fq_default_poly_init**(*fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)

Initialises poly for use, with context ctx, and setting its length to zero. A corresponding call to *fq_default_poly_clear()* must be made after finishing with the *fq_default_poly_t* to free the memory used by the polynomial.

void **fq_default_poly_init2**(*fq_default_poly_t* poly, *slong* alloc, const *fq_default_ctx_t* ctx)

Initialises poly with space for at least alloc coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to *fq_default_poly_clear()* must be made after finishing with the *fq_default_poly_t* to free the memory used by the polynomial.

void **fq_default_poly_realloc**(*fq_default_poly_t* poly, *slong* alloc, const *fq_default_ctx_t* ctx)
 Reallocates the given polynomial to have space for *alloc* coefficients. If *alloc* is zero the polynomial is cleared and then reinitialised. If the current length is greater than *alloc* the polynomial is first truncated to length *alloc*.

void **fq_default_poly_fit_length**(*fq_default_poly_t* poly, *slong* len, const *fq_default_ctx_t* ctx)
 If *len* is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least *len* coefficients. No data is lost when calling this function.

The function efficiently deals with the case where *fit_length* is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void **fq_default_poly_clear**(*fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)
 Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void **_fq_default_poly_set_length**(*fq_default_poly_t* poly, *slong* len, const *fq_default_ctx_t* ctx)
 Set the length of *poly* to *len*.

void **fq_default_poly_truncate**(*fq_default_poly_t* poly, *slong* newlen, const *fq_default_ctx_t* ctx)
 Truncates the polynomial to length at most *n*.

void **fq_default_poly_set_trunc**(*fq_default_poly_t* poly1, *fq_default_poly_t* poly2, *slong* newlen, const *fq_default_ctx_t* ctx)
 Sets *poly1* to *poly2* truncated to length *n*.

void **fq_default_poly_reverse**(*fq_default_poly_t* output, const *fq_default_poly_t* input, *slong* m, const *fq_default_ctx_t* ctx)
 Sets *output* to the reverse of *input*, thinking of it as a polynomial of length *m*, notionally zero-padded if necessary). The length *m* must be non-negative, but there are no other restrictions. The output polynomial will be set to length *m* and then normalised.

11.7.3 Polynomial parameters

slong **fq_default_poly_degree**(const *fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)
 Returns the degree of the polynomial *poly*.

slong **fq_default_poly_length**(const *fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)
 Returns the length of the polynomial *poly*.

11.7.4 Randomisation

void **fq_default_poly_randtest**(*fq_default_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_default_ctx_t* ctx)
 Sets *f* to a random polynomial of length at most *len* with entries in the field described by *ctx*.

void **fq_default_poly_randtest_not_zero**(*fq_default_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_default_ctx_t* ctx)
 Same as *fq_default_poly_randtest* but guarantees that the polynomial is not zero.

void **fq_default_poly_randtest_monic**(*fq_default_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_default_ctx_t* ctx)
 Sets *f* to a random monic polynomial of length *len* with entries in the field described by *ctx*.

void **fq_default_poly_randtest_irreducible**(*fq_default_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_default_ctx_t* ctx)
 Sets *f* to a random monic, irreducible polynomial of length *len* with entries in the field described by *ctx*.

11.7.5 Assignment and basic manipulation

void **fq_default_poly_set**(*fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, const *fq_default_ctx_t* ctx)

Sets the polynomial poly1 to the polynomial poly2.

void **fq_default_poly_set_fq_default**(*fq_default_poly_t* poly, const *fq_default_t* c, const *fq_default_ctx_t* ctx)

Sets the polynomial poly to c.

void **fq_default_poly_swap**(*fq_default_poly_t* op1, *fq_default_poly_t* op2, const *fq_default_ctx_t* ctx)

Swaps the two polynomials op1 and op2.

void **fq_default_poly_zero**(*fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)

Sets poly to the zero polynomial.

void **fq_default_poly_one**(*fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)

Sets poly to the constant polynomial 1.

void **fq_default_poly_gen**(*fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)

Sets poly to the polynomial x .

void **fq_default_poly_make_monic**(*fq_default_poly_t* rop, const *fq_default_poly_t* op, const *fq_default_ctx_t* ctx)

Sets rop to op, normed to have leading coefficient 1.

void **fq_default_poly_set_nmod_poly**(*fq_default_poly_t* rop, const *nmod_poly_t* op, const *fq_default_ctx_t* ctx)

Sets the polynomial rop to the polynomial op.

void **fq_default_poly_set_fmpz_mod_poly**(*fq_default_poly_t* rop, const *fmpz_mod_poly_t* op, const *fq_default_ctx_t* ctx)

Sets the polynomial rop to the polynomial op.

void **fq_default_poly_set_fmpz_poly**(*fq_default_poly_t* rop, const *fmpz_poly_t* op, const *fq_default_ctx_t* ctx)

Sets the polynomial rop to the polynomial op.

11.7.6 Getting and setting coefficients

void **fq_default_poly_get_coeff**(*fq_default_t* x, const *fq_default_poly_t* poly, *slong* n, const *fq_default_ctx_t* ctx)

Sets x to the coefficient of X^n in poly.

void **fq_default_poly_set_coeff**(*fq_default_poly_t* poly, *slong* n, const *fq_default_t* x, const *fq_default_ctx_t* ctx)

Sets the coefficient of X^n in poly to x .

void **fq_default_poly_set_coeff_fmpz**(*fq_default_poly_t* poly, *slong* n, const *fmpz_t* x, const *fq_default_ctx_t* ctx)

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

11.7.7 Comparison

int **fq_default_poly_equal**(const *fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, const *fq_default_ctx_t* ctx)

Returns nonzero if the two polynomials **poly1** and **poly2** are equal, otherwise returns zero.

int **fq_default_poly_equal_trunc**(const *fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, *slong* n, const *fq_default_ctx_t* ctx)

Notionally truncate **poly1** and **poly2** to length *n* and return nonzero if they are equal, otherwise return zero.

int **fq_default_poly_is_zero**(const *fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)

Returns whether the polynomial **poly** is the zero polynomial.

int **fq_default_poly_is_one**(const *fq_default_poly_t* op, const *fq_default_ctx_t* ctx)

Returns whether the polynomial **poly** is equal to the constant polynomial 1.

int **fq_default_poly_is_gen**(const *fq_default_poly_t* op, const *fq_default_ctx_t* ctx)

Returns whether the polynomial **poly** is equal to the polynomial *x*.

int **fq_default_poly_is_unit**(const *fq_default_poly_t* op, const *fq_default_ctx_t* ctx)

Returns whether the polynomial **poly** is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

int **fq_default_poly_equal_fq_default**(const *fq_default_poly_t* poly, const *fq_default_t* c, const *fq_default_ctx_t* ctx)

Returns whether the polynomial **poly** is equal the (constant) $\mathbf{F}_q$ element *c*

11.7.8 Addition and subtraction

void **fq_default_poly_add**(*fq_default_poly_t* res, const *fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, const *fq_default_ctx_t* ctx)

Sets **res** to the sum of **poly1** and **poly2**.

void **fq_default_poly_add_si**(*fq_default_poly_t* res, const *fq_default_poly_t* poly1, *slong* c, const *fq_default_ctx_t* ctx)

Sets **res** to the sum of **poly1** and *c*.

void **fq_default_poly_add_series**(*fq_default_poly_t* res, const *fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, *slong* n, const *fq_default_ctx_t* ctx)

Notionally truncate **poly1** and **poly2** to length *n* and set **res** to the sum.

void **fq_default_poly_sub**(*fq_default_poly_t* res, const *fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, const *fq_default_ctx_t* ctx)

Sets **res** to the difference of **poly1** and **poly2**.

void **fq_default_poly_sub_series**(*fq_default_poly_t* res, const *fq_default_poly_t* poly1, const *fq_default_poly_t* poly2, *slong* n, const *fq_default_ctx_t* ctx)

Notionally truncate **poly1** and **poly2** to length *n* and set **res** to the difference.

void **fq_default_poly_neg**(*fq_default_poly_t* res, const *fq_default_poly_t* poly, const *fq_default_ctx_t* ctx)

Sets **res** to the additive inverse of **poly**.

11.7.9 Scalar multiplication and division

```
void fq_default_poly_scalar_mul_fq_default(fq_default_poly_t rop, const fq_default_poly_t op,  
                                           const fq_default_t x, const fq_default_ctx_t ctx)
```

Sets `rop` to the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void fq_default_poly_scalar_addmul_fq_default(fq_default_poly_t rop, const fq_default_poly_t  
                                             op, const fq_default_t x, const fq_default_ctx_t  
                                             ctx)
```

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void fq_default_poly_scalar_submul_fq_default(fq_default_poly_t rop, const fq_default_poly_t  
                                             op, const fq_default_t x, const fq_default_ctx_t  
                                             ctx)
```

Subtracts from `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void fq_default_poly_scalar_div_fq_default(fq_default_poly_t rop, const fq_default_poly_t op,  
                                           const fq_default_t x, const fq_default_ctx_t ctx)
```

Sets `rop` to the quotient of `op` by the scalar `x`, in the context defined by `ctx`. An exception is raised if `x` is zero.

11.7.10 Multiplication

```
void fq_default_poly_mul(fq_default_poly_t rop, const fq_default_poly_t op1, const  
                        fq_default_poly_t op2, const fq_default_ctx_t ctx)
```

Sets `rop` to the product of `op1` and `op2`, choosing an appropriate algorithm.

```
void fq_default_poly_mullo(fq_default_poly_t rop, const fq_default_poly_t op1, const  
                           fq_default_poly_t op2, slong n, const fq_default_ctx_t ctx)
```

Sets `rop` to the lowest `n` coefficients of the product of `op1` and `op2`.

```
void fq_default_poly_mulhigh(fq_default_poly_t res, const fq_default_poly_t poly1, const  
                             fq_default_poly_t poly2, slong start, const fq_default_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced.

```
void fq_default_poly_mulmod(fq_default_poly_t res, const fq_default_poly_t poly1, const  
                            fq_default_poly_t poly2, const fq_default_poly_t f, const  
                            fq_default_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

11.7.11 Squaring

```
void fq_default_poly_sqr(fq_default_poly_t rop, const fq_default_poly_t op, const  
                        fq_default_ctx_t ctx)
```

Sets `rop` to the square of `op`, choosing an appropriate algorithm.

11.7.12 Powering

void **fq_default_poly_pow**(*fq_default_poly_t* rop, const *fq_default_poly_t* op, *ulong* e, const *fq_default_ctx_t* ctx)

Computes $\text{rop} = \text{op}^e$. If e is zero, returns one, so that in particular $0^0 = 1$.

void **fq_default_poly_powmod_ui_binexp**(*fq_default_poly_t* res, const *fq_default_poly_t* poly, *ulong* e, const *fq_default_poly_t* f, const *fq_default_ctx_t* ctx)

Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require $e \geq 0$.

void **fq_default_poly_powmod_fmpz_binexp**(*fq_default_poly_t* res, const *fq_default_poly_t* poly, const *fmpz_t* e, const *fq_default_poly_t* f, const *fq_default_ctx_t* ctx)

Sets *res* to *poly* raised to the power *e* modulo *f*, using binary exponentiation. We require $e \geq 0$.

void **fq_default_poly_pow_trunc**(*fq_default_poly_t* res, const *fq_default_poly_t* poly, *ulong* e, *slong* trunc, const *fq_default_ctx_t* ctx)

Sets *res* to the low *trunc* coefficients of *poly* to the power *e*. This is equivalent to doing a powering followed by a truncation.

11.7.13 Shifting

void **fq_default_poly_shift_left**(*fq_default_poly_t* rop, const *fq_default_poly_t* op, *slong* n, const *fq_default_ctx_t* ctx)

Sets *rop* to *op* shifted left by *n* coeffs. Zero coefficients are inserted.

void **fq_default_poly_shift_right**(*fq_default_poly_t* rop, const *fq_default_poly_t* op, *slong* n, const *fq_default_ctx_t* ctx)

Sets *rop* to *op* shifted right by *n* coefficients. If *n* is equal to or greater than the current length of *op*, *rop* is set to the zero polynomial.

11.7.14 Norms

slong **fq_default_poly_hamming_weight**(const *fq_default_poly_t* op, const *fq_default_ctx_t* ctx)

Returns the number of non-zero entries in the polynomial *op*.

11.7.15 Euclidean division

void **fq_default_poly_divrem**(*fq_default_poly_t* Q, *fq_default_poly_t* R, const *fq_default_poly_t* A, const *fq_default_poly_t* B, const *fq_default_ctx_t* ctx)

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

void **fq_default_poly_rem**(*fq_default_poly_t* R, const *fq_default_poly_t* A, const *fq_default_poly_t* B, const *fq_default_ctx_t* ctx)

Sets *R* to the remainder of the division of *A* by *B* in the context described by *ctx*.

void **fq_default_poly_inv_series**(*fq_default_poly_t* Qinv, const *fq_default_poly_t* Q, *slong* n, const *fq_default_ctx_t* ctx)

Given *Q* find *Qinv* such that $Q * Qinv$ is 1 modulo x^n . The constant coefficient of *Q* must be invertible modulo the modulus of *Q*. An exception is raised if this is not the case or if $n = 0$.


```
void fq_default_poly_div_series(fq_default_poly_t Q, const fq_default_poly_t A, const
                             fq_default_poly_t B, slong n, const fq_default_ctx_t ctx)
```

Set Q to the quotient of the series A by B , thinking of the series as though they were of length n . We assume that the bottom coefficient of B is invertible.

11.7.16 Greatest common divisor

```
void fq_default_poly_gcd(fq_default_poly_t rop, const fq_default_poly_t op1, const
                       fq_default_poly_t op2, const fq_default_ctx_t ctx)
```

Sets `rop` to the greatest common divisor of `op1` and `op2`, using either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

```
void fq_default_poly_xgcd(fq_default_poly_t G, fq_default_poly_t S, fq_default_poly_t T, const
                        fq_default_poly_t A, const fq_default_poly_t B, const fq_default_ctx_t
                        ctx)
```

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S \cdot A + T \cdot B = G$. The length of S will be at most `lenB` and the length of T will be at most `lenA`.

11.7.17 Divisibility testing

```
int fq_default_poly_divides(fq_default_poly_t Q, const fq_default_poly_t A, const
                          fq_default_poly_t B, const fq_default_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

11.7.18 Derivative

```
void fq_default_poly_derivative(fq_default_poly_t rop, const fq_default_poly_t op, const
                              fq_default_ctx_t ctx)
```

Sets `rop` to the derivative of `op`.

11.7.19 Square root

```
void fq_default_poly_invsqrt_series(fq_default_poly_t g, const fq_default_poly_t h, slong n,
                                   fq_default_ctx_t ctx)
```

Set g to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
void fq_default_poly_sqrt_series(fq_default_poly_t g, const fq_default_poly_t h, slong n,
                                fq_default_ctx_t ctx)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
int fq_default_poly_sqrt(fq_default_poly_t s, const fq_default_poly_t p, fq_default_ctx_t mod)
```

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

11.7.20 Evaluation

```
void fq_default_poly_evaluate_fq_default(fq_default_t rop, const fq_default_poly_t f, const
fq_default_t a, const fq_default_ctx_t ctx)
```

Sets `rop` to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

11.7.21 Composition

```
void fq_default_poly_compose(fq_default_poly_t rop, const fq_default_poly_t op1, const
fq_default_poly_t op2, const fq_default_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f , g , and h , respectively, sets $f(t) = g(h(t))$.

```
void fq_default_poly_compose_mod(fq_default_poly_t res, const fq_default_poly_t f, const
fq_default_poly_t g, const fq_default_poly_t h, const
fq_default_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero.

11.7.22 Output

```
int fq_default_poly_fprint_pretty(FILE *file, const fq_default_poly_t poly, const char *x, const
fq_default_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_default_poly_print_pretty(const fq_default_poly_t poly, const char *x, const
fq_default_ctx_t ctx)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_default_poly_fprint(FILE *file, const fq_default_poly_t poly, const fq_default_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_default_poly_print(const fq_default_poly_t poly, const fq_default_ctx_t ctx)
```

Prints the representation of `poly` to `stdout`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
char *fq_default_poly_get_str(const fq_default_poly_t poly, const fq_default_ctx_t ctx)
```

Returns the plain FLINT string representation of the polynomial `poly`.

```
char *fq_default_poly_get_str_pretty(const fq_default_poly_t poly, const char *x, const
fq_default_ctx_t ctx)
```

Returns a pretty representation of the polynomial `poly` using the null-terminated string `x` as the variable name

11.7.23 Inflation and deflation

void **fq_default_poly_inflate**(*fq_default_poly_t* result, const *fq_default_poly_t* input, *ulong* inflation, const *fq_default_ctx_t* ctx)

Sets **result** to the inflated polynomial $p(x^n)$ where p is given by **input** and n is given by **inflation**.

void **fq_default_poly_deflate**(*fq_default_poly_t* result, const *fq_default_poly_t* input, *ulong* deflation, const *fq_default_ctx_t* ctx)

Sets **result** to the deflated polynomial $p(x^{1/n})$ where p is given by **input** and n is given by **deflation**. Requires $n > 0$.

ulong **fq_default_poly_deflation**(const *fq_default_poly_t* input, const *fq_default_ctx_t* ctx)

Returns the largest integer by which **input** can be deflated. As special cases, returns 0 if **input** is the zero polynomial and 1 if **input** is a constant polynomial.

11.8 fq_poly_factor.h – factorisation of univariate polynomials over finite fields

11.8.1 Types, macros and constants

type **fq_poly_factor_struct**

type **fq_poly_factor_t**

11.8.2 Memory Management

void **fq_poly_factor_init**(*fq_poly_factor_t* fac, const *fq_ctx_t* ctx)

Initialises **fac** for use. An *fq_poly_factor_t* represents a polynomial in factorised form as a product of polynomials with associated exponents.

void **fq_poly_factor_clear**(*fq_poly_factor_t* fac, const *fq_ctx_t* ctx)

Frees all memory associated with **fac**.

void **fq_poly_factor_realloc**(*fq_poly_factor_t* fac, *slong* alloc, const *fq_ctx_t* ctx)

Reallocates the factor structure to provide space for precisely **alloc** factors.

void **fq_poly_factor_fit_length**(*fq_poly_factor_t* fac, *slong* len, const *fq_ctx_t* ctx)

Ensures that the factor structure has space for at least **len** factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.8.3 Basic Operations

void **fq_poly_factor_set**(*fq_poly_factor_t* res, const *fq_poly_factor_t* fac, const *fq_ctx_t* ctx)

Sets **res** to the same factorisation as **fac**.

void **fq_poly_factor_print_pretty**(const *fq_poly_factor_t* fac, const char *var, const *fq_ctx_t* ctx)

Pretty-prints the entries of **fac** to standard output.

void **fq_poly_factor_print**(const *fq_poly_factor_t* fac, const *fq_ctx_t* ctx)

Prints the entries of **fac** to standard output.

```
void fq_poly_factor_insert(fq_poly_factor_t fac, const fq_poly_t poly, slong exp, const fq_ctx_t ctx)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void fq_poly_factor_concat(fq_poly_factor_t res, const fq_poly_factor_t fac, const fq_ctx_t ctx)
```

Concatenates two factorisations.

This is equivalent to calling `fq_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void fq_poly_factor_pow(fq_poly_factor_t fac, slong exp, const fq_ctx_t ctx)
```

Raises `fac` to the power `exp`.

```
ulong fq_poly_remove(fq_poly_t f, const fq_poly_t p, const fq_ctx_t ctx)
```

Removes the highest possible power of `p` from `f` and returns the exponent.

11.8.4 Irreducibility Testing

```
int fq_poly_is_irreducible(const fq_poly_t f, const fq_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int fq_poly_is_irreducible_ddf(const fq_poly_t f, const fq_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

```
int fq_poly_is_irreducible_ben_or(const fq_poly_t f, const fq_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

```
int _fq_poly_is_squarefree(const fq_struct *f, slong len, const fq_ctx_t ctx)
```

Returns 1 if `(f, len)` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

```
int fq_poly_is_squarefree(const fq_poly_t f, const fq_ctx_t ctx)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.8.5 Factorisation

```
int fq_poly_factor_equal_deg_prob(fq_poly_t factor, flint_rand_t state, const fq_poly_t pol, slong d, const fq_ctx_t ctx)
```

Probabilistic equal degree factorisation of `pol` into irreducible factors of degree `d`. If it passes, a factor is placed in `factor` and 1 is returned, otherwise 0 is returned and the value of `factor` is undetermined.

Requires that `pol` be monic, non-constant and squarefree.

```
void fq_poly_factor_equal_deg(fq_poly_factor_t factors, const fq_poly_t pol, slong d, const fq_ctx_t ctx)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in `factors`. Requires that `pol` be monic, non-constant and squarefree.

```
void fq_poly_factor_split_single(fq_poly_t linfactor, const fq_poly_t input, const fq_ctx_t ctx)
```

Assuming `input` is a product of factors all of degree 1, finds a single linear factor of `input` and places it in `linfactor`. Requires that `input` be monic and non-constant.

```
void fq_poly_factor_distinct_deg(fq_poly_factor_t res, const fq_poly_t poly, slong *const *degs,
                                const fq_ctx_t ctx)
```

Factorises a monic non-constant squarefree polynomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d . Factors are stored in `res`, associated powers of irreducible polynomials are stored in `degs` in the same order as factors.

Requires that `degs` have enough space for irreducible polynomials' powers (maximum space required is $n * \text{sizeof}(\text{slong})$).

```
void fq_poly_factor_squarefree(fq_poly_factor_t res, const fq_poly_t f, const fq_ctx_t ctx)
    Sets res to a squarefree factorization of f.
```

```
void fq_poly_factor(fq_poly_factor_t res, fq_t lead, const fq_poly_t f, const fq_ctx_t ctx)
    Factorises a non-constant polynomial f into monic irreducible factors choosing the best algorithm for given modulo and degree. The output lead is set to the leading coefficient of  $f$  upon return. Choice of algorithm is based on heuristic measurements.
```

```
void fq_poly_factor_cantor_zassenhaus(fq_poly_factor_t res, const fq_poly_t f, const fq_ctx_t
                                      ctx)
    Factorises a non-constant polynomial f into monic irreducible factors using the Cantor-Zassenhaus algorithm.
```

```
void fq_poly_factor_kaltofen_shoup(fq_poly_factor_t res, const fq_poly_t poly, const fq_ctx_t
                                   ctx)
    Factorises a non-constant polynomial f into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step.
```

```
void fq_poly_factor_berlekamp(fq_poly_factor_t factors, const fq_poly_t f, const fq_ctx_t ctx)
    Factorises a non-constant polynomial f into monic irreducible factors using the Berlekamp algorithm.
```

```
void fq_poly_factor_with_berlekamp(fq_poly_factor_t res, fq_t leading_coeff, const fq_poly_t f,
                                   const fq_ctx_t ctx)
    Factorises a general polynomial f into monic irreducible factors and sets leading_coeff to the leading coefficient of f, or 0 if f is the zero polynomial.
```

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp factorisation on all the individual square-free factors.

```
void fq_poly_factor_with_cantor_zassenhaus(fq_poly_factor_t res, fq_t leading_coeff, const
                                           fq_poly_t f, const fq_ctx_t ctx)
    Factorises a general polynomial f into monic irreducible factors and sets leading_coeff to the leading coefficient of f, or 0 if f is the zero polynomial.
```

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
void fq_poly_factor_with_kaltofen_shoup(fq_poly_factor_t res, fq_t leading_coeff, const
                                         fq_poly_t f, const fq_ctx_t ctx)
    Factorises a general polynomial f into monic irreducible factors and sets leading_coeff to the leading coefficient of f, or 0 if f is the zero polynomial.
```

This function first checks for small special cases, deflates `f` if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
void fq_poly_iterated_frobenius_preinv(fq_poly_t *rop, slong n, const fq_poly_t v, const
fq_poly_t vinv, const fq_ctx_t ctx)
```

Sets `rop[i]` to be $x^{q^i} \bmod v$ for $0 \leq i < n$.

It is required that `vinv` is the inverse of the reverse of `v` mod x^{lenv} .

11.8.6 Root Finding

```
void fq_poly_roots(fq_poly_factor_t r, const fq_poly_t f, int with_multiplicity, const fq_ctx_t ctx)
```

Fill `r` with factors of the form $x - r_i$ where the r_i are the distinct roots of a nonzero f in F_q . If `with_multiplicity` is zero, the exponent e_i of the factor $x - r_i$ is 1. Otherwise, it is the largest e_i such that $(x - r_i)^{e_i}$ divides f . This function throws if f is zero, but is otherwise always successful.

11.9 fq_default_poly_factor.h – factorisation of univariate polynomials over finite fields

11.9.1 Types, macros and constants

```
type fq_default_poly_factor_t
```

11.9.2 Memory Management

```
void fq_default_poly_factor_init(fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)
```

Initialises `fac` for use. An `fq_default_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

```
void fq_default_poly_factor_clear(fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)
```

Frees all memory associated with `fac`.

```
void fq_default_poly_factor_realloc(fq_default_poly_factor_t fac, slong alloc, const
fq_default_ctx_t ctx)
```

Reallocates the factor structure to provide space for precisely `alloc` factors.

```
void fq_default_poly_factor_fit_length(fq_default_poly_factor_t fac, slong len, const
fq_default_ctx_t ctx)
```

Ensures that the factor structure has space for at least `len` factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.9.3 Basic Operations

```
void fq_default_poly_factor_set(fq_default_poly_factor_t res, const fq_default_poly_factor_t
fac, const fq_default_ctx_t ctx)
```

Sets `res` to the same factorisation as `fac`.

```
void fq_default_poly_factor_print_pretty(const fq_default_poly_factor_t fac, const char *var,
const fq_default_ctx_t ctx)
```

Pretty-prints the entries of `fac` to standard output.

```
void fq_default_poly_factor_print(const fq_default_poly_factor_t fac, const fq_default_ctx_t
ctx)
```

Prints the entries of `fac` to standard output.

```
void fq_default_poly_factor_insert(fq_default_poly_factor_t fac, const fq_default_poly_t poly,
                                  slong exp, const fq_default_ctx_t ctx)
```

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

```
void fq_default_poly_factor_concat(fq_default_poly_factor_t res, const fq_default_poly_factor_t
                                  fac, const fq_default_ctx_t ctx)
```

Concatenates two factorisations.

This is equivalent to calling `fq_default_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

```
void fq_default_poly_factor_pow(fq_default_poly_factor_t fac, slong exp, const fq_default_ctx_t
                                ctx)
```

Raises `fac` to the power `exp`.

```
ulong fq_default_poly_remove(fq_default_poly_t f, const fq_default_poly_t p, const
                              fq_default_ctx_t ctx)
```

Removes the highest possible power of `p` from `f` and returns the exponent.

```
slong fq_default_poly_factor_length(fq_default_poly_factor_t fac, const fq_default_ctx_t ctx)
```

Return the number of factors, not including the unit.

```
void fq_default_poly_factor_get_poly(fq_default_poly_t poly, const fq_default_poly_factor_t
                                     fac, slong i, const fq_default_ctx_t ctx)
```

Set `poly` to factor `i` of `fac` (numbering starts at zero).

```
slong fq_default_poly_factor_exp(fq_default_poly_factor_t fac, slong i, const fq_default_ctx_t
                                 ctx)
```

Return the exponent of factor `i` of `fac`.

11.9.4 Irreducibility Testing

```
int fq_default_poly_is_irreducible(const fq_default_poly_t f, const fq_default_ctx_t ctx)
```

Returns 1 if the polynomial `f` is irreducible, otherwise returns 0.

```
int fq_default_poly_is_squarefree(const fq_default_poly_t f, const fq_default_ctx_t ctx)
```

Returns 1 if `f` is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.9.5 Factorisation

```
void fq_default_poly_factor_equal_deg(fq_default_poly_factor_t factors, const fq_default_poly_t
                                      pol, slong d, const fq_default_ctx_t ctx)
```

Assuming `pol` is a product of irreducible factors all of degree `d`, finds all those factors and places them in `factors`. Requires that `pol` be monic, non-constant and squarefree.

```
void fq_default_poly_factor_split_single(fq_default_poly_t linfactor, const fq_default_poly_t
                                         input, const fq_default_ctx_t ctx)
```

Assuming `input` is a product of factors all of degree 1, finds a single linear factor of `input` and places it in `linfactor`. Requires that `input` be monic and non-constant.


```
void fq_default_poly_factor_distinct_deg(fq_default_poly_factor_t res, const fq_default_poly_t
poly, slong *const *degs, const fq_default_ctx_t ctx)
```

Factorises a monic non-constant squarefree polynomial `poly` of degree n into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of `poly` of degree d . Factors are stored in `res`, associated powers of irreducible polynomials are stored in `degs` in the same order as factors.

Requires that `degs` have enough space for irreducible polynomials' powers (maximum space required is $n * \text{sizeof}(\text{slong})$).

```
void fq_default_poly_factor_squarefree(fq_default_poly_factor_t res, const fq_default_poly_t f,
const fq_default_ctx_t ctx)
```

Sets `res` to a squarefree factorization of `f`.

```
void fq_default_poly_factor(fq_default_poly_factor_t res, fq_default_t lead, const
fq_default_poly_t f, const fq_default_ctx_t ctx)
```

Factorises a non-constant polynomial `f` into monic irreducible factors choosing the best algorithm for given modulo and degree. The output `lead` is set to the leading coefficient of f upon return. Choice of algorithm is based on heuristic measurements.

11.9.6 Root Finding

```
void fq_default_poly_roots(fq_default_poly_factor_t r, const fq_default_poly_t f, int
with_multiplicity, const fq_default_ctx_t ctx)
```

Fill `r` with factors of the form $x - r_i$ where the r_i are the distinct roots of a nonzero f in F_q . If `with_multiplicity` is zero, the exponent e_i of the factor $x - r_i$ is 1. Otherwise, it is the largest e_i such that $(x - r_i)^{e_i}$ divides f . This function throws if f is zero, but is otherwise always successful.

11.10 fq_embed.h – Computing isomorphisms and embeddings of finite fields

```
void fq_embed_gens(fq_t gen_sub, fq_t gen_sup, fmpz_mod_poly_t minpoly, const fq_ctx_t
sub_ctx, const fq_ctx_t sup_ctx)
```

Given two contexts `sub_ctx` and `sup_ctx`, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute:

- an element `gen_sub` in `sub_ctx` such that `gen_sub` generates the finite field defined by `sub_ctx`,
- its minimal polynomial `minpoly`,
- a root `gen_sup` of `minpoly` inside the field defined by `sup_ctx`.

These data uniquely define an embedding of `sub_ctx` into `sup_ctx`.

```
void _fq_embed_gens_naive(fq_t gen_sub, fq_t gen_sup, fmpz_mod_poly_t minpoly, const
fq_ctx_t sub_ctx, const fq_ctx_t sup_ctx)
```

Given two contexts `sub_ctx` and `sup_ctx`, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute an embedding of `sub_ctx` into `sup_ctx` defined as follows:

- `gen_sub` is the canonical generator of `sup_ctx` (i.e., the class of X),
- `minpoly` is the defining polynomial of `sub_ctx`,
- `gen_sup` is a root of `minpoly` inside the field defined by `sup_ctx`.

```
void fq_embed_matrices(fmpz_mod_mat_t embed, fmpz_mod_mat_t project, const fq_t gen_sub,
                      const fq_ctx_t sub_ctx, const fq_t gen_sup, const fq_ctx_t sup_ctx, const
                      fmpz_mod_poly_t gen_minpoly)
```

Given:

- two contexts `sub_ctx` and `sup_ctx`, of respective degrees m and n , such that m divides n ;
- a generator `gen_sub` of `sub_ctx`, its minimal polynomial `gen_minpoly`, and a root `gen_sup` of `gen_minpoly` in `sup_ctx`, as returned by `fq_embed_gens()`;

Compute:

- the $n \times m$ matrix `embed` mapping `gen_sub` to `gen_sup`, and all their powers accordingly;
- an $m \times n$ matrix `project` such that `project` $\times$ `embed` is the $m \times m$ identity matrix.

```
void fq_embed_trace_matrix(fmpz_mod_mat_t res, const fmpz_mod_mat_t basis, const fq_ctx_t
                          sub_ctx, const fq_ctx_t sup_ctx)
```

Given:

- two contexts `sub_ctx` and `sup_ctx`, of degrees m and n , such that m divides n ;
- an $n \times m$ matrix `basis` that maps `sub_ctx` to an isomorphic subfield in `sup_ctx`;

Compute the $m \times n$ matrix of the trace from `sup_ctx` to `sub_ctx`.

This matrix is computed as

```
embed_dual_to_mono_matrix(_, sub_ctx)  $\times$  basist  $\times$  embed_mono_to_dual_matrix(_,
sup_ctx).
```

Note: if $m = n$, `basis` represents a Frobenius, and the result is its inverse matrix.

```
void fq_embed_composition_matrix(fmpz_mod_mat_t matrix, const fq_t gen, const fq_ctx_t ctx)
Compute the composition matrix of gen.
```

For an element $a \in \mathbf{F}_{p^n}$, its composition matrix is the matrix whose columns are $a^0, a^1, \dots, a^{n-1}$.

```
void fq_embed_composition_matrix_sub(fmpz_mod_mat_t matrix, const fq_t gen, const fq_ctx_t
                                   ctx, slong trunc)
```

Compute the *composition matrix* of `gen`, truncated to `trunc` columns.

```
void fq_embed_mul_matrix(fmpz_mod_mat_t matrix, const fq_t gen, const fq_ctx_t ctx)
Compute the multiplication matrix of gen.
```

For an element a in $\mathbf{F}_{p^n} = \mathbf{F}_p[x]$, its multiplication matrix is the matrix whose columns are $a, ax, \dots, ax^{n-1}$.

```
void fq_embed_mono_to_dual_matrix(fmpz_mod_mat_t res, const fq_ctx_t ctx)
Compute the change of basis matrix from the monomial basis of ctx to its dual basis.
```

```
void fq_embed_dual_to_mono_matrix(fmpz_mod_mat_t res, const fq_ctx_t ctx)
Compute the change of basis matrix from the dual basis of ctx to its monomial basis.
```

```
void fq_modulus_pow_series_inv(fmpz_mod_poly_t res, const fq_ctx_t ctx, slong trunc)
Compute the power series inverse of the reverse of the modulus of ctx up to  $O(x^{\text{trunc}})$ .
```

```
void fq_modulus_derivative_inv(fq_t m_prime, fq_t m_prime_inv, const fq_ctx_t ctx)
Compute the derivative m_prime of the modulus of ctx as an element of ctx, and its inverse
m_prime_inv.
```

11.11 fq_nmod.h – finite fields (word-size characteristic)

We represent an element of the finite field $\mathbf{F}_{p^n} \cong \mathbf{F}_p[X]/(f(X))$, where $f(X) \in \mathbf{F}_p[X]$ is a monic, irreducible polynomial of degree n , as a polynomial in $\mathbf{F}_p[X]$ of degree less than n . The underlying data structure is an `nmod_poly_t`.

The default choice for $f(X)$ is the Conway polynomial for the pair (p, n) , enabled by Frank Lübeck's data base of Conway polynomials using the `_nmod_poly_conway()` function. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

11.11.1 Types, macros and constants

type `fq_nmod_ctx_struct`

type `fq_nmod_ctx_t`

type `fq_nmod_struct`

type `fq_nmod_t`

11.11.2 Context Management

void `fq_nmod_ctx_init_ui`(`fq_nmod_ctx_t` ctx, *ulong* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name `var` for the generator. By default, it will try use a Conway polynomial; if one is not available, a minimal weight irreducible polynomial will be used.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

void `fq_nmod_ctx_init_minimal_weight_ui`(`fq_nmod_ctx_t` ctx, *ulong* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name `var` for the generator, choosing a modulus polynomial with minimal number of nonzero terms for efficient arithmetic.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

int `_fq_nmod_ctx_init_conway_ui`(`fq_nmod_ctx_t` ctx, *ulong* p, *slong* d, const char *var)

Attempts to initialise the context for prime p and extension degree d , with name `var` for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

void `fq_nmod_ctx_init_conway_ui`(`fq_nmod_ctx_t` ctx, *ulong* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name `var` for the generator using a Conway polynomial for the modulus.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

void **fq_nmod_ctx_init_modulus**(*fq_nmod_ctx_t* ctx, const *nmod_poly_t* modulus, const char *var)

Initialises the context for given **modulus** with name **var** for the generator.

Assumes that **modulus** is an irreducible polynomial over $\mathbf{F}_p$.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_nmod_ctx_init_randtest**(*fq_nmod_ctx_t* ctx, *flint_rand_t* state, int type)

Initialises **ctx** to a random finite field, where the prime and degree is set according to **type**. To see what prime and degrees may be output, see **type** in *_nmod_poly_conway_rand()*.

void **fq_nmod_ctx_init_randtest_reducible**(*fq_nmod_ctx_t* ctx, *flint_rand_t* state, int type)

Initializes **ctx** to a random extension of a word-sized prime field, where the prime and degree is set according to **type**. If **type** is 0 the prime and degree may be large, else if **type** is 1 the degree is small but the prime may be large, else if **type** is 2 the prime is small but the degree may be large, else if **type** is 3 both prime and degree are small.

The modulus may or may not be irreducible.

void **fq_nmod_ctx_clear**(*fq_nmod_ctx_t* ctx)

Clears all memory that has been allocated as part of the context.

const *nmod_poly_struct* ***fq_nmod_ctx_modulus**(const *fq_nmod_ctx_t* ctx)

Returns a pointer to the modulus in the context.

slong **fq_nmod_ctx_degree**(const *fq_nmod_ctx_t* ctx)

Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

ulong **fq_nmod_ctx_prime**(const *fq_nmod_ctx_t* ctx)

Returns the prime p of the context.

void **fq_nmod_ctx_order**(*fmpz_t* f, const *fq_nmod_ctx_t* ctx)

Sets f to be the size of the finite field.

int **fq_nmod_ctx_fprint**(FILE *file, const *fq_nmod_ctx_t* ctx)

Prints the context information to **file**. Returns 1 for a success and a negative number for an error.

void **fq_nmod_ctx_print**(const *fq_nmod_ctx_t* ctx)

Prints the context information to **stdout**.

11.11.3 Memory management

void **fq_nmod_init**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Initialises the element **rop**, setting its value to 0. Currently, the behaviour is identical to **fq_nmod_init2**, as it also ensures **rop** has enough space for it to be an element of **ctx**, this may change in the future.

void **fq_nmod_init2**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Initialises **rop** with at least enough space for it to be an element of **ctx** and sets it to 0.

void **fq_nmod_clear**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Clears the element **rop**.

void **_fq_nmod_sparse_reduce**(*ulong* *R, *slong* lenR, const *fq_nmod_ctx_t* ctx)

Reduces (R, lenR) modulo the polynomial f given by the modulus of **ctx**.

void **_fq_nmod_dense_reduce**(*ulong* *R, *slong* lenR, const *fq_nmod_ctx_t* ctx)

Reduces (R, lenR) modulo the polynomial f given by the modulus of **ctx** using Newton division.

void **_fq_nmod_reduce**(*ulong* *r, *slong* lenR, const *fq_nmod_ctx_t* ctx)

Reduces (R, lenR) modulo the polynomial f given by the modulus of **ctx**. Does either sparse or dense reduction based on **ctx->sparse_modulus**.

void **fq_nmod_reduce**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Reduces the polynomial *rop* as an element of $\mathbf{F}_p[X]/(f(X))$.

11.11.4 Basic arithmetic

void **fq_nmod_add**(*fq_nmod_t* rop, const *fq_nmod_t* op1, const *fq_nmod_t* op2, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the sum of *op1* and *op2*.

void **fq_nmod_sub**(*fq_nmod_t* rop, const *fq_nmod_t* op1, const *fq_nmod_t* op2, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the difference of *op1* and *op2*.

void **fq_nmod_sub_one**(*fq_nmod_t* rop, const *fq_nmod_t* op1, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the difference of *op1* and 1.

void **fq_nmod_neg**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the negative of *op*.

void **fq_nmod_mul**(*fq_nmod_t* rop, const *fq_nmod_t* op1, const *fq_nmod_t* op2, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the product of *op1* and *op2*, reducing the output in the given context.

void **fq_nmod_mul_fmpz**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *fmpz_t* x, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the product of *op* and *x*, reducing the output in the given context.

void **fq_nmod_mul_si**(*fq_nmod_t* rop, const *fq_nmod_t* op, *slong* x, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the product of *op* and *x*, reducing the output in the given context.

void **fq_nmod_mul_ui**(*fq_nmod_t* rop, const *fq_nmod_t* op, *ulong* x, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the product of *op* and *x*, reducing the output in the given context.

void **fq_nmod_sqr**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the square of *op*, reducing the output in the given context.

void **_fq_nmod_inv**(*nn_ptr* *rop, *nn_srcptr* *op, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets (*rop*, *d*) to the inverse of the non-zero element (*op*, *len*).

void **fq_nmod_inv**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the inverse of the non-zero element *op*.

void **fq_nmod_gcdinv**(*fq_nmod_t* f, *fq_nmod_t* inv, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Sets *inv* to be the inverse of *op* modulo the modulus of *ctx*. If *op* is not invertible, then *f* is set to a factor of the modulus; otherwise, it is set to one.

void **_fq_nmod_pow**(*ulong* *rop, const *ulong* *op, *slong* len, const *fmpz_t* e, const *fq_nmod_ctx_t* ctx)

Sets (*rop*, $2*d-1$) to (*op*, *len*) raised to the power *e*, reduced modulo $f(X)$, the modulus of *ctx*.

Assumes that $e \geq 0$ and that *len* is positive and at most *d*.

Although we require that *rop* provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree *d*.

Does not support aliasing.

void **fq_nmod_pow**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *fmpz_t* e, const *fq_nmod_ctx_t* ctx)

Sets *rop* to *op* raised to the power *e*.

Currently assumes that $e \geq 0$.

Note that for any input *op*, *rop* is set to 1 whenever $e = 0$.

void **fq_nmod_pow_ui**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *ulong* e, const *fq_nmod_ctx_t* ctx)

Sets rop to op raised to the power e .

Currently assumes that $e \geq 0$.

Note that for any input op, rop is set to 1 whenever $e = 0$.

11.11.5 Roots

int **fq_nmod_sqrt**(*fq_nmod_t* rop, const *fq_nmod_t* op1, const *fq_nmod_ctx_t* ctx)

Sets rop to the square root of op1 if it is a square, and return 1, otherwise return 0.

void **fq_nmod_pth_root**(*fq_nmod_t* rop, const *fq_nmod_t* op1, const *fq_nmod_ctx_t* ctx)

Sets rop to a p^{th} root of op1. Currently, this computes the root by raising op1 to p^{d-1} where d is the degree of the extension.

int **fq_nmod_is_square**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Return 1 if op is a square.

11.11.6 Output

int **fq_nmod_fprint_pretty**(FILE *file, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Prints a pretty representation of op to file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

void **fq_nmod_print_pretty**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Prints a pretty representation of op to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_nmod_fprint**(FILE *file, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Prints a representation of op to file.

For further details on the representation used, see **nmod_poly_fprint()**.

void **fq_nmod_print**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Prints a representation of op to stdout.

For further details on the representation used, see **nmod_poly_print()**.

char ***fq_nmod_get_str**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns the plain FLINT string representation of the element op.

char ***fq_nmod_get_str_pretty**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns a pretty representation of the element op using the null-terminated string x as the variable name.

11.11.7 Randomisation

void **fq_nmod_randtest**(*fq_nmod_t* rop, *flint_rand_t* state, const *fq_nmod_ctx_t* ctx)

Generates a random element of $\mathbf{F}_q$.

void **fq_nmod_randtest_not_zero**(*fq_nmod_t* rop, *flint_rand_t* state, const *fq_nmod_ctx_t* ctx)

Generates a random non-zero element of $\mathbf{F}_q$.

void **fq_nmod_randtest_dense**(*fq_nmod_t* rop, *flint_rand_t* state, const *fq_nmod_ctx_t* ctx)

Generates a random element of $\mathbf{F}_q$ which has an underlying polynomial with dense coefficients.

void **fq_nmod_rand**(*fq_nmod_t* rop, *flint_rand_t* state, const *fq_nmod_ctx_t* ctx)

Generates a high quality random element of $\mathbf{F}_q$.

void **fq_nmod_rand_not_zero**(*fq_nmod_t* rop, *flint_rand_t* state, const *fq_nmod_ctx_t* ctx)

Generates a high quality non-zero random element of $\mathbf{F}_q$.

11.11.8 Assignments and conversions

void **fq_nmod_set**(*fq_nmod_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Sets rop to op.

void **fq_nmod_set_si**(*fq_nmod_t* rop, const *slong* x, const *fq_nmod_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_nmod_set_ui**(*fq_nmod_t* rop, const *ulong* x, const *fq_nmod_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_nmod_set_fmpz**(*fq_nmod_t* rop, const *fmpz_t* x, const *fq_nmod_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_nmod_swap**(*fq_nmod_t* op1, *fq_nmod_t* op2, const *fq_nmod_ctx_t* ctx)

Swaps the two elements op1 and op2.

void **fq_nmod_zero**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Sets rop to zero.

void **fq_nmod_one**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Sets rop to one, reduced in the given context.

void **fq_nmod_gen**(*fq_nmod_t* rop, const *fq_nmod_ctx_t* ctx)

Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int **fq_nmod_get_fmpz**(*fmpz_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

If op has a lift to the integers, return 1 and set rop to the lift in $[0, p)$. Otherwise, return 0 and leave rop undefined.

void **fq_nmod_get_nmod_poly**(*nmod_poly_t* a, const *fq_nmod_t* b, const *fq_nmod_ctx_t* ctx)

Set a to a representative of b in ctx. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void **fq_nmod_set_nmod_poly**(*fq_nmod_t* a, const *nmod_poly_t* b, const *fq_nmod_ctx_t* ctx)

Set a to the element in ctx with representative b. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void **fq_nmod_get_nmod_mat**(*nmod_mat_t* col, const *fq_nmod_t* a, const *fq_nmod_ctx_t* ctx)

Convert a to a column vector of length `degree(ctx)`.

void **fq_nmod_set_nmod_mat**(*fq_nmod_t* a, const *nmod_mat_t* col, const *fq_nmod_ctx_t* ctx)

Convert a column vector col of length `degree(ctx)` to an element of ctx.

11.11.9 Comparison

int **fq_nmod_is_zero**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns whether *op* is equal to zero.

int **fq_nmod_is_one**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns whether *op* is equal to one.

int **fq_nmod_equal**(const *fq_nmod_t* op1, const *fq_nmod_t* op2, const *fq_nmod_ctx_t* ctx)

Returns whether *op1* and *op2* are equal.

int **fq_nmod_is_invertible**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns whether *op* is an invertible element.

int **fq_nmod_is_invertible_f**(*fq_nmod_t* f, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns whether *op* is an invertible element. If it is not, then *f* is set to a factor of the modulus.

int **fq_nmod_cmp**(const *fq_nmod_t* a, const *fq_nmod_t* b, const *fq_nmod_ctx_t* ctx)

Return 1 (resp. -1, or 0) if *a* is after (resp. before, same as) *b* in some arbitrary but fixed total ordering of the elements.

11.11.10 Special functions

void **_fq_nmod_trace**(*fmpz_t* rop, const *ulong* *op, *ulong* len, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the trace of the non-zero element (*op*, *len*) in $\mathbf{F}_q$.

void **fq_nmod_trace**(*fmpz_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the trace of *op*.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

void **_fq_nmod_norm**(*fmpz_t* rop, const *ulong* *op, *ulong* len, const *fq_nmod_ctx_t* ctx)

Sets *rop* to the norm of the non-zero element (*op*, *len*) in $\mathbf{F}_q$.

void **fq_nmod_norm**(*fmpz_t* rop, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Computes the norm of *op*.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbf{F}_p}(\mathbf{F}_q)$.

Algorithm selection is automatic depending on the input.

void **_fq_nmod_frobenius**(*ulong* *rop, const *ulong* *op, *ulong* len, *ulong* e, const *fq_nmod_ctx_t* ctx)

Sets (*rop*, *2d-1*) to the image of (*op*, *len*) under the Frobenius operator raised to the *e*-th power, assuming that neither *op* nor *e* are zero.

void **fq_nmod_frobenius**(*fq_nmod_t* rop, const *fq_nmod_t* op, *ulong* e, const *fq_nmod_ctx_t* ctx)

Evaluates the homomorphism Σ^e at *op*.

Recall that $\mathbf{F}_q/\mathbf{F}_p$ is Galois with Galois group $\langle \sigma \rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma: x \mapsto x^p$.

int **fq_nmod_multiplicative_order**(*fmpz_t* *ord, const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Computes the order of *op* as an element of the multiplicative group of *ctx*.

Returns 0 if *op* is 0, otherwise it returns 1 if *op* is a generator of the multiplicative group, and -1 if it is not.

This function can also be used to check primitivity of a generator of a finite field whose defining polynomial is not primitive.

int **fq_nmod_is_primitive**(const *fq_nmod_t* op, const *fq_nmod_ctx_t* ctx)

Returns whether op is primitive, i.e., whether it is a generator of the multiplicative group of ctx.

11.11.11 Bit packing

void **fq_nmod_bit_pack**(*fmpz_t* f, const *fq_nmod_t* op, *flint_bitcnt_t* bit_size, const *fq_nmod_ctx_t* ctx)

Packs op into bitfields of size bit_size, writing the result to f.

void **fq_nmod_bit_unpack**(*fq_nmod_t* rop, const *fmpz_t* f, *flint_bitcnt_t* bit_size, const *fq_nmod_ctx_t* ctx)

Unpacks into rop the element with coefficients packed into fields of size bit_size as represented by the integer f.

11.12 fq_nmod_vec.h – vectors over finite fields (word-size characteristic)

11.12.1 Memory management

fq_nmod_struct ***_fq_nmod_vec_init**(*slong* len, const *fq_nmod_ctx_t* ctx)

Returns an initialised vector of fq_nmod's of given length.

void **_fq_nmod_vec_clear**(*fq_nmod_struct* *vec, *slong* len, const *fq_nmod_ctx_t* ctx)

Clears the entries of (vec, len) and frees the space allocated for vec.

11.12.2 Randomisation

void **_fq_nmod_vec_randtest**(*fq_nmod_struct* *f, *flint_rand_t* state, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets the entries of a vector of the given length to elements of the finite field.

11.12.3 Input and output

int **_fq_nmod_vec_fprint**(FILE *file, const *fq_nmod_struct* *vec, *slong* len, const *fq_nmod_ctx_t* ctx)

Prints the vector of given length to the stream file. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fq_nmod_vec_print**(const *fq_nmod_struct* *vec, *slong* len, const *fq_nmod_ctx_t* ctx)

Prints the vector of given length to stdout.

For further details, see _fq_nmod_vec_fprint().

11.12.4 Assignment and basic manipulation

```
void _fq_nmod_vec_set(fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const  
                    fq_nmod_ctx_t ctx)
```

Makes a copy of (vec2, len2) into vec1.

```
void _fq_nmod_vec_swap(fq_nmod_struct *vec1, fq_nmod_struct *vec2, slong len2, const  
                     fq_nmod_ctx_t ctx)
```

Swaps the elements in (vec1, len2) and (vec2, len2).

```
void _fq_nmod_vec_zero(fq_nmod_struct *vec, slong len, const fq_nmod_ctx_t ctx)
```

Zeros the entries of (vec, len).

```
void _fq_nmod_vec_neg(fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len2, const  
                    fq_nmod_ctx_t ctx)
```

Negates (vec2, len2) and places it into vec1.

11.12.5 Comparison

```
int _fq_nmod_vec_equal(const fq_nmod_struct *vec1, const fq_nmod_struct *vec2, slong len, const  
                     fq_nmod_ctx_t ctx)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```
int _fq_nmod_vec_is_zero(const fq_nmod_struct *vec, slong len, const fq_nmod_ctx_t ctx)
```

Returns 1 if (vec, len) is zero, and 0 otherwise.

11.12.6 Addition and subtraction

```
void _fq_nmod_vec_add(fq_nmod_struct *res, const fq_nmod_struct *vec1, const fq_nmod_struct  
                    *vec2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

```
void _fq_nmod_vec_sub(fq_nmod_struct *res, const fq_nmod_struct *vec1, const fq_nmod_struct  
                    *vec2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets (res, len2) to (vec1, len2) minus (vec2, len2).

11.12.7 Scalar multiplication and division

```
void _fq_nmod_vec_scalar_addmul_fq_nmod(fq_nmod_struct *vec1, const fq_nmod_struct *vec2,  
                                       slong len2, const fq_nmod_t c, const fq_nmod_ctx_t  
                                       ctx)
```

Adds (vec2, len2) times *c* to (vec1, len2), where *c* is a *fq_nmod_t*.

```
void _fq_nmod_vec_scalar_submul_fq_nmod(fq_nmod_struct *vec1, const fq_nmod_struct *vec2,  
                                       slong len2, const fq_nmod_t c, const fq_nmod_ctx_t  
                                       ctx)
```

Subtracts (vec2, len2) times *c* from (vec1, len2), where *c* is a *fq_nmod_t*.

11.12.8 Dot products

```
void _fq_nmod_vec_dot(fq_nmod_t res, const fq_nmod_struct *vec1, const fq_nmod_struct *vec2,
                     slong len2, const fq_nmod_ctx_t ctx)
```

Sets `res` to the dot product of `(vec1, len)` and `(vec2, len)`.

11.13 `fq_nmod_mat.h` – matrices over finite fields (word-size characteristic)

11.13.1 Types, macros and constants

```
type fq_nmod_mat_struct
```

```
type fq_nmod_mat_t
```

11.13.2 Memory management

```
void fq_nmod_mat_init(fq_nmod_mat_t mat, slong rows, slong cols, const fq_nmod_ctx_t ctx)
```

Initialises `mat` to a rows-by-cols matrix with coefficients in $\mathbf{F}_q$ given by `ctx`. All elements are set to zero.

```
void fq_nmod_mat_init_set(fq_nmod_mat_t mat, const fq_nmod_mat_t src, const fq_nmod_ctx_t ctx)
```

Initialises `mat` and sets its dimensions and elements to those of `src`.

```
void fq_nmod_mat_clear(fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an *fq_nmod_mat_t* object.

```
void fq_nmod_mat_set(fq_nmod_mat_t mat, const fq_nmod_mat_t src, const fq_nmod_ctx_t ctx)
```

Sets `mat` to a copy of `src`. It is assumed that `mat` and `src` have identical dimensions.

11.13.3 Basic properties and manipulation

```
fq_nmod_struct *fq_nmod_mat_entry(const fq_nmod_mat_t mat, slong i, slong j)
```

Directly accesses the entry in `mat` in row `i` and column `j`, indexed from zero. No bounds checking is performed.

```
void fq_nmod_mat_entry_set(fq_nmod_mat_t mat, slong i, slong j, const fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Sets the entry in `mat` in row `i` and column `j` to `x`.

```
slong fq_nmod_mat_nrows(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns the number of rows in `mat`.

```
slong fq_nmod_mat_ncols(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns the number of columns in `mat`.

```
void fq_nmod_mat_swap(fq_nmod_mat_t mat1, fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
```

Swaps two matrices. The dimensions of `mat1` and `mat2` are allowed to be different.

```
void fq_nmod_mat_swap_entrywise(fq_nmod_mat_t mat1, fq_nmod_mat_t mat2, const
                                fq_nmod_ctx_t ctx)
```

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

```
void fq_nmod_mat_zero(fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Sets all entries of `mat` to 0.

```
void fq_nmod_mat_one(fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Sets all diagonal entries of `mat` to 1 and all other entries to 0.

```
void fq_nmod_mat_swap_rows(fq_nmod_mat_t mat, slong *perm, slong r, slong s, const
                            fq_nmod_ctx_t ctx)
```

Swaps rows `r` and `s` of `mat`. If `perm` is non-NULL, the permutation of the rows will also be applied to `perm`.

```
void fq_nmod_mat_swap_cols(fq_nmod_mat_t mat, slong *perm, slong r, slong s, const
                            fq_nmod_ctx_t ctx)
```

Swaps columns `r` and `s` of `mat`. If `perm` is non-NULL, the permutation of the columns will also be applied to `perm`.

```
void fq_nmod_mat_invert_rows(fq_nmod_mat_t mat, slong *perm, const fq_nmod_ctx_t ctx)
```

Swaps rows `i` and `r - i` of `mat` for $0 \leq i < r/2$, where `r` is the number of rows of `mat`. If `perm` is non-NULL, the permutation of the rows will also be applied to `perm`.

```
void fq_nmod_mat_invert_cols(fq_nmod_mat_t mat, slong *perm, const fq_nmod_ctx_t ctx)
```

Swaps columns `i` and `c - i` of `mat` for $0 \leq i < c/2$, where `c` is the number of columns of `mat`. If `perm` is non-NULL, the permutation of the columns will also be applied to `perm`.

11.13.4 Conversions

```
void fq_nmod_mat_set_nmod_mat(fq_nmod_mat_t mat1, const nmod_mat_t mat2, const
                               fq_nmod_ctx_t ctx)
```

Sets the matrix `mat1` to the matrix `mat2`.

```
void fq_nmod_mat_set_fmpz_mod_mat(fq_nmod_mat_t mat1, const fmpz_mod_mat_t mat2, const
                                   fq_nmod_ctx_t ctx)
```

Sets the matrix `mat1` to the matrix `mat2`.

11.13.5 Concatenate

```
void fq_nmod_mat_concat_vertical(fq_nmod_mat_t res, const fq_nmod_mat_t mat1, const
                                 fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
```

Sets `res` to vertical concatenation of (`mat1`, `mat2`) in that order. Matrix dimensions : `mat1` : $m \times n$, `mat2` : $k \times n$, `res` : $(m + k) \times n$.

```
void fq_nmod_mat_concat_horizontal(fq_nmod_mat_t res, const fq_nmod_mat_t mat1, const
                                   fq_nmod_mat_t mat2, const fq_nmod_ctx_t ctx)
```

Sets `res` to horizontal concatenation of (`mat1`, `mat2`) in that order. Matrix dimensions : `mat1` : $m \times n$, `mat2` : $m \times k$, `res` : $m \times (n + k)$.

11.13.6 Printing

int `fq_nmod_mat_print_pretty`(const *fq_nmod_mat_t* mat, const *fq_nmod_ctx_t* ctx)

Pretty-prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

int `fq_nmod_mat_fprint_pretty`(FILE *file, const *fq_nmod_mat_t* mat, const *fq_nmod_ctx_t* ctx)

Pretty-prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int `fq_nmod_mat_print`(const *fq_nmod_mat_t* mat, const *fq_nmod_ctx_t* ctx)

Prints `mat` to `stdout`. A header is printed followed by the rows enclosed in brackets.

int `fq_nmod_mat_fprint`(FILE *file, const *fq_nmod_mat_t* mat, const *fq_nmod_ctx_t* ctx)

Prints `mat` to `file`. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.13.7 Window

void `fq_nmod_mat_window_init`(*fq_nmod_mat_t* window, const *fq_nmod_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2, const *fq_nmod_ctx_t* ctx)

Initializes the matrix `window` to be an $r2 - r1$ by $c2 - c1$ submatrix of `mat` whose (0,0) entry is the (r1, c1) entry of `mat`. The memory for the elements of `window` is shared with `mat`.

void `fq_nmod_mat_window_clear`(*fq_nmod_mat_t* window, const *fq_nmod_ctx_t* ctx)

Clears the matrix `window` and releases any memory that it uses. Note that the memory to the underlying matrix that `window` points to is not freed.

11.13.8 Random matrix generation

void `fq_nmod_mat_randtest`(*fq_nmod_mat_t* mat, *flint_rand_t* state, const *fq_nmod_ctx_t* ctx)

Sets the elements of `mat` to random elements of $\mathbf{F}_q$, given by `ctx`.

int `fq_nmod_mat_randpermdiag`(*fq_nmod_mat_t* mat, *flint_rand_t* state, *fq_nmod_struct* *diag, *slong* n, const *fq_nmod_ctx_t* ctx)

Sets `mat` to a random permutation of the diagonal matrix with n leading entries given by the vector `diag`. It is assumed that the main diagonal of `mat` has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.

void `fq_nmod_mat_randrank`(*fq_nmod_mat_t* mat, *flint_rand_t* state, *slong* rank, const *fq_nmod_ctx_t* ctx)

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbf{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_nmod_mat_randops()`.

void `fq_nmod_mat_randops`(*fq_nmod_mat_t* mat, *flint_rand_t* state, *slong* count, const *fq_nmod_ctx_t* ctx)

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

void `fq_nmod_mat_randtril`(*fq_nmod_mat_t* mat, *flint_rand_t* state, int unit, const *fq_nmod_ctx_t* ctx)

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```
void fq_nmod_mat_randtriu(fq_nmod_mat_t mat, flint_rand_t state, int unit, const
                          fq_nmod_ctx_t ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

11.13.9 Comparison

```
int fq_nmod_mat_equal(const fq_nmod_mat_t mat1, const fq_nmod_mat_t mat2, const
                      fq_nmod_ctx_t ctx)
```

Returns nonzero if `mat1` and `mat2` have the same dimensions and elements, and zero otherwise.

```
int fq_nmod_mat_is_zero(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.

```
int fq_nmod_mat_is_one(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero except the diagonal entries which must be one, otherwise returns zero.

```
int fq_nmod_mat_is_empty(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fq_nmod_mat_is_square(const fq_nmod_mat_t mat, const fq_nmod_ctx_t ctx)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

11.13.10 Transpose

```
void fq_nmod_mat_transpose(fq_nmod_mat_t B, const fq_nmod_mat_t A, const fq_nmod_ctx_t
                           ctx)
```

Sets B to A^T , the transpose of A . Dimensions must be compatible. Aliasing is allowed for square matrices.

11.13.11 Addition and subtraction

```
void fq_nmod_mat_add(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const
                     fq_nmod_ctx_t ctx)
```

Computes $C = A + B$. Dimensions must be identical.

```
void fq_nmod_mat_sub(fq_nmod_mat_t C, const fq_nmod_mat_t A, const fq_nmod_mat_t B, const
                     fq_nmod_ctx_t ctx)
```

Computes $C = A - B$. Dimensions must be identical.

```
void fq_nmod_mat_neg(fq_nmod_mat_t A, const fq_nmod_mat_t B, const fq_nmod_ctx_t ctx)
```

Sets $B = -A$. Dimensions must be identical.

11.13.12 Matrix multiplication

void `fq_nmod_mat_mul`(*fq_nmod_mat_t* C, const *fq_nmod_mat_t* A, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. Aliasing is allowed. This function automatically chooses between classical and KS multiplication.

void `fq_nmod_mat_mul_classical`(*fq_nmod_mat_t* C, const *fq_nmod_mat_t* A, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses classical matrix multiplication.

void `fq_nmod_mat_mul_KS`(*fq_nmod_mat_t* C, const *fq_nmod_mat_t* A, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses Kronecker substitution to perform the multiplication over the integers.

void `fq_nmod_mat_submul`(*fq_nmod_mat_t* D, const *fq_nmod_mat_t* C, const *fq_nmod_mat_t* A, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B .

void `fq_nmod_mat_mul_vec`(*fq_nmod_struct* *c, const *fq_nmod_mat_t* A, const *fq_nmod_struct* *b, *slong* blen, const *fq_nmod_ctx_t* ctx)

void `fq_nmod_mat_mul_vec_ptr`(*fq_nmod_struct* *const *c, const *fq_nmod_mat_t* A, const *fq_nmod_struct* *const *b, *slong* blen, const *fq_nmod_ctx_t* ctx)

Compute a matrix-vector product of A and $(b, blen)$ and store the result in c . The vector $(b, blen)$ is either truncated or zero-extended to the number of columns of A . The number entries written to c is always equal to the number of rows of A .

void `fq_nmod_mat_vec_mul`(*fq_nmod_struct* *c, const *fq_nmod_struct* *a, *slong* alen, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

void `fq_nmod_mat_vec_mul_ptr`(*fq_nmod_struct* *const *c, const *fq_nmod_struct* *const *a, *slong* alen, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Compute a vector-matrix product of $(a, alen)$ and B and store the result in c . The vector $(a, alen)$ is either truncated or zero-extended to the number of rows of B . The number entries written to c is always equal to the number of columns of B .

11.13.13 Inverse

int `fq_nmod_mat_inv`(*fq_nmod_mat_t* B, *fq_nmod_mat_t* A, const *fq_nmod_ctx_t* ctx)

Sets $B = A^{-1}$ and returns 1 if A is invertible. If A is singular, returns 0 and sets the elements of B to undefined values.

A and B must be square matrices with the same dimensions.

11.13.14 LU decomposition

slong `fq_nmod_mat_lu`(*slong* *P, *fq_nmod_mat_t* A, int rank_check, const *fq_nmod_ctx_t* ctx)

Computes a generalised LU decomposition $PLU = A$ of a given matrix A , returning the rank of A .

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for `rank_check` is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

11.13.15 Reduced row echelon form

slong **fq_nmod_mat_rref**(*fq_nmod_mat_t* B, const *fq_nmod_mat_t* A, const *fq_nmod_ctx_t* ctx)

Puts B in reduced row echelon form and returns the rank of A .

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

slong **fq_nmod_mat_reduce_row**(*fq_nmod_mat_t* A, *slong* *P, *slong* *L, *slong* n, const *fq_nmod_ctx_t* ctx)

Reduce row n of the matrix A , assuming the prior rows are in Gauss form. However those rows may not be in order. The entry i of the array P is the row of A which has a pivot in the i -th column. If no such row exists, the entry of P will be -1 . The function returns the column in which the n -th row has a pivot after reduction. This will always be chosen to be the first available column for a pivot from the left. This information is also updated in P . Entry i of the array L contains the number of possibly nonzero columns of A row i . This speeds up reduction in the case that A is chambered on the right. Otherwise the entries of L can all be set to the number of columns of A . We require the entries of L to be monotonic increasing.

11.13.16 Triangular solving

void **fq_nmod_mat_solve_tril**(*fq_nmod_mat_t* X, const *fq_nmod_mat_t* L, const *fq_nmod_mat_t* B, int unit, const *fq_nmod_ctx_t* ctx)

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If `unit` = 1, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

void **fq_nmod_mat_solve_triu**(*fq_nmod_mat_t* X, const *fq_nmod_mat_t* U, const *fq_nmod_mat_t* B, int unit, const *fq_nmod_ctx_t* ctx)

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If `unit` = 1, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

11.13.17 Solving

int **fq_nmod_mat_solve**(*fq_nmod_mat_t* X, const *fq_nmod_mat_t* A, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

The matrix A must be square.

int **fq_nmod_mat_can_solve**(*fq_nmod_mat_t* X, const *fq_nmod_mat_t* A, const *fq_nmod_mat_t* B, const *fq_nmod_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$ over F_q .

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of X to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of A and it may be singular.

11.13.18 Transforms

void **fq_nmod_mat_similarity**(*fq_nmod_mat_t* M, *slong* r, *fq_nmod_t* d, const *fq_nmod_ctx_t* ctx)
 Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value d is required to be reduced modulo the modulus of the entries in the matrix.

11.13.19 Characteristic polynomial

void **fq_nmod_mat_charpoly**(*fq_nmod_poly_t* p, const *fq_nmod_mat_t* M, const *fq_nmod_ctx_t* ctx)

Compute the characteristic polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.13.20 Minimal polynomial

void **fq_nmod_mat_minpoly**(*fq_nmod_poly_t* p, const *fq_nmod_mat_t* M, const *fq_nmod_ctx_t* ctx)

Compute the minimal polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.14 fq_nmod_poly.h – univariate polynomials over finite fields (word-size characteristic)

We represent a polynomial in $\mathbf{F}_q[X]$ as a **struct** which includes an array **coeffs** with the coefficients, as well as the length **length** and the number **alloc** of coefficients for which memory has been allocated.

As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $\mathbf{F}_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

11.14.1 Types, macros and constants

type **fq_nmod_poly_struct**

type **fq_nmod_poly_t**

11.14.2 Memory management

void **fq_nmod_poly_init**(*fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Initialises **poly** for use, with context **ctx**, and setting its length to zero. A corresponding call to **fq_nmod_poly_clear()** must be made after finishing with the **fq_nmod_poly_t** to free the memory used by the polynomial.

void **fq_nmod_poly_init2**(*fq_nmod_poly_t* poly, *slong* alloc, const *fq_nmod_ctx_t* ctx)

Initialises **poly** with space for at least **alloc** coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to **fq_nmod_poly_clear()** must be made after finishing with the **fq_nmod_poly_t** to free the memory used by the polynomial.

void **fq_nmod_poly_realloc**(*fq_nmod_poly_t* poly, *slong* alloc, const *fq_nmod_ctx_t* ctx)

Reallocates the given polynomial to have space for **alloc** coefficients. If **alloc** is zero the polynomial is cleared and then reinitialised. If the current length is greater than **alloc** the polynomial is first truncated to length **alloc**.

void **fq_nmod_poly_fit_length**(*fq_nmod_poly_t* poly, *slong* len, const *fq_nmod_ctx_t* ctx)

If **len** is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least **len** coefficients. No data is lost when calling this function.

The function efficiently deals with the case where **fit_length** is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void **_fq_nmod_poly_set_length**(*fq_nmod_poly_t* poly, *slong* newlen, const *fq_nmod_ctx_t* ctx)

Sets the coefficients of **poly** beyond **len** to zero and sets the length of **poly** to **len**.

void **fq_nmod_poly_clear**(*fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void **_fq_nmod_poly_normalise**(*fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Sets the length of **poly** so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **_fq_nmod_poly_normalise2**(const *fq_nmod_struct* *poly, *slong* *length, const *fq_nmod_ctx_t* ctx)

Sets the length **length** of (**poly**, **length**) so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **fq_nmod_poly_truncate**(*fq_nmod_poly_t* poly, *slong* newlen, const *fq_nmod_ctx_t* ctx)

Truncates the polynomial to length at most **n**.

void **fq_nmod_poly_set_trunc**(*fq_nmod_poly_t* poly1, *fq_nmod_poly_t* poly2, *slong* newlen, const *fq_nmod_ctx_t* ctx)

Sets **poly1** to **poly2** truncated to length **n**.

void **_fq_nmod_poly_reverse**(*fq_nmod_struct* *output, const *fq_nmod_struct* *input, *slong* len, *slong* m, const *fq_nmod_ctx_t* ctx)

Sets **output** to the reverse of **input**, which is of length **len**, but thinking of it as a polynomial of length **m**, notionally zero-padded if necessary. The length **m** must be non-negative, but there are no other restrictions. The polynomial **output** must have space for **m** coefficients.

void **fq_nmod_poly_reverse**(*fq_nmod_poly_t* output, const *fq_nmod_poly_t* input, *slong* m, const *fq_nmod_ctx_t* ctx)

Sets **output** to the reverse of **input**, thinking of it as a polynomial of length **m**, notionally zero-padded if necessary). The length **m** must be non-negative, but there are no other restrictions. The output polynomial will be set to length **m** and then normalised.

11.14.3 Polynomial parameters

slong **fq_nmod_poly_degree**(const *fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Returns the degree of the polynomial **poly**.

slong **fq_nmod_poly_length**(const *fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Returns the length of the polynomial **poly**.

fq_nmod_struct ***fq_nmod_poly_lead**(const *fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Returns a pointer to the leading coefficient of **poly**, or NULL if **poly** is the zero polynomial.

11.14.4 Randomisation

void **fq_nmod_poly_randtest**(*fq_nmod_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets *f* to a random polynomial of length at most *len* with entries in the field described by *ctx*.

void **fq_nmod_poly_randtest_not_zero**(*fq_nmod_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_nmod_ctx_t* ctx)

Same as **fq_nmod_poly_randtest** but guarantees that the polynomial is not zero.

void **fq_nmod_poly_randtest_monic**(*fq_nmod_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets *f* to a random monic polynomial of length *len* with entries in the field described by *ctx*.

void **fq_nmod_poly_randtest_irreducible**(*fq_nmod_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets *f* to a random monic, irreducible polynomial of length *len* with entries in the field described by *ctx*.

11.14.5 Assignment and basic manipulation

void **_fq_nmod_poly_set**(*fq_nmod_struct* *rop, const *fq_nmod_struct* *op, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets (rop, len) to (op, len).

void **fq_nmod_poly_set**(*fq_nmod_poly_t* poly1, const *fq_nmod_poly_t* poly2, const *fq_nmod_ctx_t* ctx)

Sets the polynomial poly1 to the polynomial poly2.

void **fq_nmod_poly_set_fq_nmod**(*fq_nmod_poly_t* poly, const *fq_nmod_t* c, const *fq_nmod_ctx_t* ctx)

Sets the polynomial poly to c.

void **fq_nmod_poly_set_fmpz_mod_poly**(*fq_nmod_poly_t* rop, const *fmpz_mod_poly_t* op, const *fq_nmod_ctx_t* ctx)

Sets the polynomial rop to the polynomial op

void **fq_nmod_poly_set_nmod_poly**(*fq_nmod_poly_t* rop, const *nmod_poly_t* op, const *fq_nmod_ctx_t* ctx)

Sets the polynomial rop to the polynomial op

void **fq_nmod_poly_swap**(*fq_nmod_poly_t* op1, *fq_nmod_poly_t* op2, const *fq_nmod_ctx_t* ctx)

Swaps the two polynomials op1 and op2.

void **_fq_nmod_poly_zero**(*fq_nmod_struct* *rop, *slong* len, const *fq_nmod_ctx_t* ctx)

Sets (rop, len) to the zero polynomial.

void **fq_nmod_poly_zero**(*fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Sets poly to the zero polynomial.

void **fq_nmod_poly_one**(*fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Sets poly to the constant polynomial 1.

void **fq_nmod_poly_gen**(*fq_nmod_poly_t* poly, const *fq_nmod_ctx_t* ctx)

Sets poly to the polynomial *x*.

void **fq_nmod_poly_make_monic**(*fq_nmod_poly_t* rop, const *fq_nmod_poly_t* op, const *fq_nmod_ctx_t* ctx)

Sets rop to op, normed to have leading coefficient 1.

```
void _fq_nmod_poly_make_monic(fq_nmod_struct *rop, const fq_nmod_struct *op, slong length,
                             const fq_nmod_ctx_t ctx)
```

Sets `rop` to `(op, length)`, normed to have leading coefficient 1. Assumes that `rop` has enough space for the polynomial, assumes that `op` is not zero (and thus has an invertible leading coefficient).

11.14.6 Getting and setting coefficients

```
void fq_nmod_poly_get_coeff(fq_nmod_t x, const fq_nmod_poly_t poly, slong n, const
                             fq_nmod_ctx_t ctx)
```

Sets x to the coefficient of X^n in `poly`.

```
void fq_nmod_poly_set_coeff(fq_nmod_poly_t poly, slong n, const fq_nmod_t x, const
                             fq_nmod_ctx_t ctx)
```

Sets the coefficient of X^n in `poly` to x .

```
void fq_nmod_poly_set_coeff_fmpz(fq_nmod_poly_t poly, slong n, const fmpz_t x, const
                                 fq_nmod_ctx_t ctx)
```

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

11.14.7 Comparison

```
int fq_nmod_poly_equal(const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const
                       fq_nmod_ctx_t ctx)
```

Returns nonzero if the two polynomials `poly1` and `poly2` are equal, otherwise return zero.

```
int fq_nmod_poly_equal_trunc(const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong n,
                             const fq_nmod_ctx_t ctx)
```

Notionally truncate `poly1` and `poly2` to length n and return nonzero if they are equal, otherwise return zero.

```
int fq_nmod_poly_is_zero(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is the zero polynomial.

```
int fq_nmod_poly_is_one(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the constant polynomial 1.

```
int fq_nmod_poly_is_gen(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the polynomial x .

```
int fq_nmod_poly_is_unit(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

```
int fq_nmod_poly_equal_fq_nmod(const fq_nmod_poly_t poly, const fq_nmod_t c, const
                                fq_nmod_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal the (constant) $\mathbf{F}_q$ element `c`

11.14.8 Addition and subtraction

`void _fq_nmod_poly_add(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, const fq_nmod_ctx_t ctx)`

Sets `res` to the sum of `(poly1,len1)` and `(poly2,len2)`.

`void fq_nmod_poly_add(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)`

Sets `res` to the sum of `poly1` and `poly2`.

`void fq_nmod_poly_add_si(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, slong c, const fq_nmod_ctx_t ctx)`

Sets `res` to the sum of `poly1` and `c`.

`void fq_nmod_poly_add_series(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong n, const fq_nmod_ctx_t ctx)`

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the sum.

`void _fq_nmod_poly_sub(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const fq_nmod_struct *poly2, slong len2, const fq_nmod_ctx_t ctx)`

Sets `res` to the difference of `(poly1,len1)` and `(poly2,len2)`.

`void fq_nmod_poly_sub(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, const fq_nmod_ctx_t ctx)`

Sets `res` to the difference of `poly1` and `poly2`.

`void fq_nmod_poly_sub_series(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const fq_nmod_poly_t poly2, slong n, const fq_nmod_ctx_t ctx)`

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the difference.

`void _fq_nmod_poly_neg(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t ctx)`

Sets `rop` to the additive inverse of `(poly,len)`.

`void fq_nmod_poly_neg(fq_nmod_poly_t res, const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)`

Sets `res` to the additive inverse of `poly`.

11.14.9 Scalar multiplication and division

`void _fq_nmod_poly_scalar_mul_fq_nmod(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_t x, const fq_nmod_ctx_t ctx)`

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

`void fq_nmod_poly_scalar_mul_fq_nmod(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_t x, const fq_nmod_ctx_t ctx)`

Sets `rop` to the product of `op` by the scalar `x`, in the context defined by `ctx`.

`void _fq_nmod_poly_scalar_addmul_fq_nmod(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const fq_nmod_t x, const fq_nmod_ctx_t ctx)`

Adds to `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

`void fq_nmod_poly_scalar_addmul_fq_nmod(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_t x, const fq_nmod_ctx_t ctx)`

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.


```
void _fq_nmod_poly_scalar_submul_fq_nmod(fq_nmod_struct *rop, const fq_nmod_struct *op,
                                          slong len, const fq_nmod_t x, const fq_nmod_ctx_t
                                          ctx)
```

Subtracts from (rop, len) the product of (op, len) by the scalar x , in the context defined by ctx . In particular, assumes the same length for op and rop .

```
void fq_nmod_poly_scalar_submul_fq_nmod(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const
                                          fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Subtracts from rop the product of op by the scalar x , in the context defined by ctx .

```
void _fq_nmod_poly_scalar_div_fq(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len,
                                  const fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Sets (rop, len) to the quotient of (op, len) by the scalar x , in the context defined by ctx . An exception is raised if x is zero.

```
void fq_nmod_poly_scalar_div_fq(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const
                                  fq_nmod_t x, const fq_nmod_ctx_t ctx)
```

Sets rop to the quotient of op by the scalar x , in the context defined by ctx . An exception is raised if x is zero.

11.14.10 Multiplication

```
void _fq_nmod_poly_mul_classical(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1,
                                  const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t
                                  ctx)
```

Sets $(rop, len1 + len2 - 1)$ to the product of $(op1, len1)$ and $(op2, len2)$, assuming that $len1$ is at least $len2$ and neither is zero.

Permits zero padding. Does not support aliasing of rop with either $op1$ or $op2$.

```
void fq_nmod_poly_mul_classical(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                                  fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of $op1$ and $op2$ using classical polynomial multiplication.

```
void _fq_nmod_poly_mul_reorder(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1,
                                const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets $(rop, len1 + len2 - 1)$ to the product of $(op1, len1)$ and $(op2, len2)$, assuming that $len1$ and $len2$ are non-zero.

Permits zero padding. Supports aliasing.

```
void fq_nmod_poly_mul_reorder(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                                fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of $op1$ and $op2$, reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Suppose $\mathbf{F}_q = \mathbf{F}_p[X]/(f(X))$ and recall that elements of $\mathbf{F}_q$ are internally represented by elements of type `fmpz_poly`. For small degree extensions but polynomials in $\mathbf{F}_q[Y]$ of large degree n , we change the representation to

$$\begin{aligned} g(Y) &= \sum_{i=0}^n a_i(X) Y^i \\ &= \sum_{j=0}^d \sum_{i=0}^n \text{Coeff}(a_i(X), j) Y^i. \end{aligned}$$

This allows us to use a poor algorithm (such as classical multiplication) in the X -direction and leverage the existing fast integer multiplication routines in the Y -direction where the polynomial degree n is large.

```
void _fq_nmod_poly_mul_univariate(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1,
                                const fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t
                                ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and makes no assumptions on len1 and len2. Supports aliasing.

```
void fq_nmod_poly_mul_univariate(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                                fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of op1 and op2 using a bivariate to univariate transformation and reducing this problem to multiplying two univariate polynomials.

```
void _fq_nmod_poly_mul_KS(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const
                          fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

```
void fq_nmod_poly_mul_KS(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t
                          op2, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of op1 and op2 using Kronecker substitution, that is, by encoding each coefficient in F_q as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_nmod_poly_mul(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const
                       fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_nmod_poly_mul(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t
                      op2, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of op1 and op2, choosing an appropriate algorithm.

```
void _fq_nmod_poly_mullow_classical(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong
                                   len1, const fq_nmod_struct *op2, slong len2, slong n, const
                                   fq_nmod_ctx_t ctx)
```

Sets (rop, n) to the first n coefficients of (op1, len1) multiplied by (op2, len2).

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Assumes neither len1 nor len2 is zero.

```
void fq_nmod_poly_mullow_classical(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                                   fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of op1 and op2, computed using the classical or schoolbook method.

```
void _fq_nmod_poly_mullow_univariate(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong
                                     len1, const fq_nmod_struct *op2, slong len2, slong n, const
                                     fq_nmod_ctx_t ctx)
```

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) and (op2, len2), computed using a bivariate to univariate transformation.

Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between rop, op1 and op2.

```
void fq_nmod_poly_mullow_univariate(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                                    fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)
```

Sets rop to the lowest n coefficients of the product of poly1 and poly2, computed using a bivariate to univariate transformation.

```
void _fq_nmod_poly_mullo_KS(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const
                             fq_nmod_struct *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)
```

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) and (op2, len2).

Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between rop, op1 and op2.

```
void fq_nmod_poly_mullo_KS(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                             fq_nmod_poly_t op2, slong n, const fq_nmod_ctx_t ctx)
```

Sets rop to the product of op1 and op2.

```
void _fq_nmod_poly_mullo(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const
                          fq_nmod_struct *op2, slong len2, slong n, const fq_nmod_ctx_t ctx)
```

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) and (op2, len2).

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fq_nmod_poly_mullo(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const fq_nmod_poly_t
                        op2, slong n, const fq_nmod_ctx_t ctx)
```

Sets rop to the lowest n coefficients of the product of op1 and op2.

```
void _fq_nmod_poly_mulhigh_classical(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong
                                     len1, const fq_nmod_struct *poly2, slong len2, slong start,
                                     const fq_nmod_ctx_t ctx)
```

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that $\text{len1} \geq \text{len2} > 0$. Aliasing of inputs and output is not permitted. Algorithm is classical multiplication.

```
void fq_nmod_poly_mulhigh_classical(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const
                                    fq_nmod_poly_t poly2, slong start, const fq_nmod_ctx_t
                                    ctx)
```

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Algorithm is classical multiplication.

```
void _fq_nmod_poly_mulhigh(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const
                           fq_nmod_struct *poly2, slong len2, slong start, fq_nmod_ctx_t ctx)
```

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced. Assumes that $\text{len1} \geq \text{len2} > 0$. Aliasing of inputs and output is not permitted.

```
void fq_nmod_poly_mulhigh(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const
                           fq_nmod_poly_t poly2, slong start, const fq_nmod_ctx_t ctx)
```

Computes the product of poly1 and poly2 and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but reduced.

```
void _fq_nmod_poly_mulmod(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1, const
                          fq_nmod_struct *poly2, slong len2, const fq_nmod_struct *f, slong lenf,
                          const fq_nmod_ctx_t ctx)
```

Sets res to the remainder of the product of poly1 and poly2 upon polynomial division by f.

It is required that $\text{len1} + \text{len2} - \text{lenf} > 0$, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use _fq_nmod_poly_mul instead.

Aliasing of f and res is not permitted.

```
void fq_nmod_poly_mulmod(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const
                          fq_nmod_poly_t poly2, const fq_nmod_poly_t f, const fq_nmod_ctx_t
                          ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fq_nmod_poly_mulmod_preinv(fq_nmod_struct *res, const fq_nmod_struct *poly1, slong len1,
                                const fq_nmod_struct *poly2, slong len2, const fq_nmod_struct
                                *f, slong lenf, const fq_nmod_struct *finv, slong lenfinv, const
                                fq_nmod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f mod xlenf`.

Aliasing of `res` with any of the inputs is not permitted.

```
void fq_nmod_poly_mulmod_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly1, const
                                fq_nmod_poly_t poly2, const fq_nmod_poly_t f, const
                                fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

11.14.11 Squaring

```
void _fq_nmod_poly_sqr_classical(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len,
                                const fq_nmod_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op, len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_nmod_poly_sqr_classical(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const
                                fq_nmod_ctx_t ctx)
```

Sets `rop` to the square of `op` using classical polynomial multiplication.

```
void _fq_nmod_poly_sqr_KS(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const
                           fq_nmod_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_nmod_poly_sqr_KS(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t
                           ctx)
```

Sets `rop` to the square `op` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_nmod_poly_sqr(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const
                       fq_nmod_ctx_t ctx)
```

Sets `(rop, 2 * len - 1)` to the square of `(op, len)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_nmod_poly_sqr(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Sets `rop` to the square of `op`, choosing an appropriate algorithm.

11.14.12 Powering

void `_fq_nmod_poly_pow`(*fq_nmod_struct* *rop, const *fq_nmod_struct* *op, *slong* len, *ulong* e, const *fq_nmod_ctx_t* ctx)

Sets `rop = ope`, assuming that `e`, `len > 0` and that `rop` has space for `e*(len - 1) + 1` coefficients. Does not support aliasing.

void `fq_nmod_poly_pow`(*fq_nmod_poly_t* rop, const *fq_nmod_poly_t* op, *ulong* e, const *fq_nmod_ctx_t* ctx)

Computes `rop = ope`. If `e` is zero, returns one, so that in particular `00 = 1`.

void `_fq_nmod_poly_powmod_ui_binexp`(*fq_nmod_struct* *res, const *fq_nmod_struct* *poly, *ulong* e, const *fq_nmod_struct* *f, *slong* lenf, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

void `fq_nmod_poly_powmod_ui_binexp`(*fq_nmod_poly_t* res, const *fq_nmod_poly_t* poly, *ulong* e, const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

void `_fq_nmod_poly_powmod_ui_binexp_preinv`(*fq_nmod_struct* *res, const *fq_nmod_struct* *poly, *ulong* e, const *fq_nmod_struct* *f, *slong* lenf, const *fq_nmod_struct* *finv, *slong* lenfinv, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

void `fq_nmod_poly_powmod_ui_binexp_preinv`(*fq_nmod_poly_t* res, const *fq_nmod_poly_t* poly, *ulong* e, const *fq_nmod_poly_t* f, const *fq_nmod_poly_t* finv, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

void `_fq_nmod_poly_powmod_fmpz_binexp`(*fq_nmod_struct* *res, const *fq_nmod_struct* *poly, const *fmpz_t* e, const *fq_nmod_struct* *f, *slong* lenf, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

void `fq_nmod_poly_powmod_fmpz_binexp`(*fq_nmod_poly_t* res, const *fq_nmod_poly_t* poly, const *fmpz_t* e, const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

void `_fq_nmod_poly_powmod_fmpz_binexp_preinv`(*fq_nmod_struct* *res, const *fq_nmod_struct* *poly, const *fmpz_t* e, const *fq_nmod_struct* *f, *slong* lenf, const *fq_nmod_struct* *finv, *slong* lenfinv, const *fq_nmod_ctx_t* ctx)

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_fmpz_binexp_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly,
                                             const fmpz_t e, const fq_nmod_poly_t f, const
                                             fq_nmod_poly_t finv, const fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_nmod_poly_powmod_fmpz_sliding_preinv(fq_nmod_struct *res, const fq_nmod_struct
                                              *poly, const fmpz_t e, ulong k, const
                                              fq_nmod_struct *f, slong lenf, const
                                              fq_nmod_struct *finv, slong lenfinv, const
                                              fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an “optimum” size will be selected automatically base on `e`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_fmpz_sliding_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t poly,
                                             const fmpz_t e, ulong k, const fq_nmod_poly_t f,
                                             const fq_nmod_poly_t finv, const
                                             fq_nmod_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an “optimum” size will be selected automatically base on `e`.

```
void _fq_nmod_poly_powmod_x_fmpz_preinv(fq_nmod_struct *res, const fmpz_t e, const
                                        fq_nmod_struct *f, slong lenf, const fq_nmod_struct
                                        *finv, slong lenfinv, const fq_nmod_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_nmod_poly_powmod_x_fmpz_preinv(fq_nmod_poly_t res, const fmpz_t e, const
                                        fq_nmod_poly_t f, const fq_nmod_poly_t finv, const
                                        fq_nmod_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_nmod_poly_pow_trunc_binexp(fq_nmod_struct *res, const fq_nmod_struct *poly, ulong e,
                                    slong trunc, const fq_nmod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void fq_nmod_poly_pow_trunc_binexp(fq_nmod_poly_t res, const fq_nmod_poly_t poly, ulong e,
                                   slong trunc, const fq_nmod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.


```
void _fq_nmod_poly_pow_trunc(fq_nmod_struct *res, const fq_nmod_struct *poly, ulong e, slong
                             trunc, const fq_nmod_ctx_t mod)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc` > 0 and that `e` > 1 . Aliasing is not permitted.

```
void fq_nmod_poly_pow_trunc(fq_nmod_poly_t res, const fq_nmod_poly_t poly, ulong e, slong
                             trunc, const fq_nmod_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.

11.14.13 Shifting

```
void _fq_nmod_poly_shift_left(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, slong
                              n, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len + n)` to `(op, len)` shifted left by n coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and n are positive, and that `rop` fits `len + n` elements. Supports aliasing between `rop` and `op`.

```
void fq_nmod_poly_shift_left(fq_nmod_poly_t rop, const fq_nmod_poly_t op, slong n, const
                             fq_nmod_ctx_t ctx)
```

Sets `rop` to `op` shifted left by n coeffs. Zero coefficients are inserted.

```
void _fq_nmod_poly_shift_right(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, slong
                               n, const fq_nmod_ctx_t ctx)
```

Sets `(rop, len - n)` to `(op, len)` shifted right by n coefficients.

Assumes that `len` and n are positive, that `len` $> n$, and that `rop` fits `len - n` elements. Supports aliasing between `rop` and `op`, although in this case the top coefficients of `op` are not set to zero.

```
void fq_nmod_poly_shift_right(fq_nmod_poly_t rop, const fq_nmod_poly_t op, slong n, const
                              fq_nmod_ctx_t ctx)
```

Sets `rop` to `op` shifted right by n coefficients. If n is equal to or greater than the current length of `op`, `rop` is set to the zero polynomial.

11.14.14 Norms

```
slong _fq_nmod_poly_hamming_weight(const fq_nmod_struct *op, slong len, const fq_nmod_ctx_t
                                    ctx)
```

Returns the number of non-zero entries in `(op, len)`.

```
slong fq_nmod_poly_hamming_weight(const fq_nmod_poly_t op, const fq_nmod_ctx_t ctx)
```

Returns the number of non-zero entries in the polynomial `op`.

11.14.15 Euclidean division

```
void _fq_nmod_poly_divrem(fq_nmod_struct *Q, fq_nmod_struct *R, const fq_nmod_struct *A,
                          slong lenA, const fq_nmod_struct *B, slong lenB, const fq_nmod_t
                          invB, const fq_nmod_ctx_t ctx)
```

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (R, lenA) such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that `invB` is its inverse.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.


```
void fq_nmod_poly_divrem(fq_nmod_poly_t Q, fq_nmod_poly_t R, const fq_nmod_poly_t A, const
                        fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

```
void fq_nmod_poly_divrem_f(fq_nmod_t f, fq_nmod_poly_t Q, fq_nmod_poly_t R, const
                          fq_nmod_poly_t A, const fq_nmod_poly_t B, const fq_nmod_ctx_t
                          ctx)
```

Either finds a non-trivial factor f of the modulus of `ctx`, or computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

If the leading coefficient of B is invertible, the division with remainder operation is carried out, Q and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of the modulus and Q and R are not touched.

Assumes that B is non-zero.

```
void _fq_nmod_poly_rem(fq_nmod_struct *R, const fq_nmod_struct *A, slong lenA, const
                     fq_nmod_struct *B, slong lenB, const fq_nmod_t invB, const
                     fq_nmod_ctx_t ctx)
```

Sets R to the remainder of the division of $(A, \text{len}A)$ by $(B, \text{len}B)$. Assumes that the leading coefficient of $(B, \text{len}B)$ is invertible and that `invB` is its inverse.

```
void fq_nmod_poly_rem(fq_nmod_poly_t R, const fq_nmod_poly_t A, const fq_nmod_poly_t B,
                    const fq_nmod_ctx_t ctx)
```

Sets R to the remainder of the division of A by B in the context described by `ctx`.

```
void _fq_nmod_poly_div(fq_nmod_struct *Q, const fq_nmod_struct *A, slong lenA, const
                     fq_nmod_struct *B, slong lenB, const fq_nmod_t invB, const
                     fq_nmod_ctx_t ctx)
```

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(Q, \text{len}A - \text{len}B + 1)$.

Allows zero-padding in A but not in B . Assumes that the leading coefficient of B is a unit.

```
void fq_nmod_poly_div(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const fq_nmod_poly_t B,
                    const fq_nmod_ctx_t ctx)
```

Notionally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q . If $\text{len}(B) = 0$ an exception is raised.

```
void _fq_nmod_poly_div_newton_n_preinv(fq_nmod_struct *Q, const fq_nmod_struct *A, slong
                                      lenA, const fq_nmod_struct *B, slong lenB, const
                                      fq_nmod_struct *Binv, slong lenBinv, const
                                      fq_nmod_ctx_t ctx)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than `lenB`, where A is of length `lenA` and B is of length `lenB`, but return only Q .

We require that Q have space for `lenA - lenB + 1` coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fq_nmod_poly_div_newton_n_preinv(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const
                                      fq_nmod_poly_t B, const fq_nmod_poly_t Binv, const
                                      fq_nmod_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q .

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \cdot \text{length of } B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_struct *Q, fq_nmod_struct *R, const
                                         fq_nmod_struct *A, slong lenA, const
                                         fq_nmod_struct *B, slong lenB, const
                                         fq_nmod_struct *Binv, slong lenBinv, const
                                         fq_nmod_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$. The algorithm used is to call `div_newton_preinv()` and then multiply out and compute the remainder.

```
void fq_nmod_poly_divrem_newton_n_preinv(fq_nmod_poly_t Q, fq_nmod_poly_t R, const
                                         fq_nmod_poly_t A, const fq_nmod_poly_t B, const
                                         fq_nmod_poly_t Binv, const fq_nmod_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume B_{inv} is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \cdot \text{length of } B - 2$.

The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

```
void _fq_nmod_poly_inv_series_newton(fq_nmod_struct *Qinv, const fq_nmod_struct *Q, slong n,
                                     const fq_nmod_t cinv, const fq_nmod_ctx_t ctx)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial Q_{inv} of length n such that $Q * Q_{\text{inv}}$ is 1 modulo x^n . Requires $n > 0$. This function can be viewed as inverting a power series via Newton iteration.

```
void fq_nmod_poly_inv_series_newton(fq_nmod_poly_t Qinv, const fq_nmod_poly_t Q, slong n,
                                    const fq_nmod_ctx_t ctx)
```

Given Q find Q_{inv} such that $Q * Q_{\text{inv}}$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series via Newton iteration.

```
void _fq_nmod_poly_inv_series(fq_nmod_struct *Qinv, const fq_nmod_struct *Q, slong n, const
                              fq_nmod_t cinv, const fq_nmod_ctx_t ctx)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial Q_{inv} of length n such that $Q * Q_{\text{inv}}$ is 1 modulo x^n . Requires $n > 0$.

```
void fq_nmod_poly_inv_series(fq_nmod_poly_t Qinv, const fq_nmod_poly_t Q, slong n, const
                              fq_nmod_ctx_t ctx)
```

Given Q find Q_{inv} such that $Q * Q_{\text{inv}}$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$.

```
void _fq_nmod_poly_div_series(fq_nmod_struct *Q, const fq_nmod_struct *A, slong Alen, const
                              fq_nmod_struct *B, slong Blen, slong n, const fq_nmod_ctx_t ctx)
```

Set (Q, n) to the quotient of the series (A, Alen) and (B, Blen) assuming $\text{Alen}, \text{Blen} \leq n$. We assume the bottom coefficient of B is invertible.

```
void fq_nmod_poly_div_series(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const
                              fq_nmod_poly_t B, slong n, fq_nmod_ctx_t ctx)
```

Set Q to the quotient of the series A by B , thinking of the series as though they were of length n . We assume that the bottom coefficient of B is invertible.

11.14.16 Greatest common divisor

void **fq_nmod_poly_gcd**(*fq_nmod_poly_t* rop, const *fq_nmod_poly_t* op1, const *fq_nmod_poly_t* op2, const *fq_nmod_ctx_t* ctx)

Sets **rop** to the greatest common divisor of **op1** and **op2**, using the either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

slong **_fq_nmod_poly_gcd**(*fq_nmod_struct* *G, const *fq_nmod_struct* *A, *slong* lenA, const *fq_nmod_struct* *B, *slong* lenB, const *fq_nmod_ctx_t* ctx)

Computes the GCD of A of length **lenA** and B of length **lenB**, where **lenA** $\geq$ **lenB** $>$ 0 and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for **lenB** coefficients.

slong **_fq_nmod_poly_gcd_euclidean_f**(*fq_nmod_t* f, *fq_nmod_struct* *G, const *fq_nmod_struct* *A, *slong* lenA, const *fq_nmod_struct* *B, *slong* lenB, const *fq_nmod_ctx_t* ctx)

Either sets $f = 1$ and G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length, or sets f to a non-trivial factor of the modulus of **ctx** and leaves the contents of the vector $(G, \text{len}B)$ undefined.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

void **fq_nmod_poly_gcd_euclidean_f**(*fq_nmod_t* f, *fq_nmod_poly_t* G, const *fq_nmod_poly_t* A, const *fq_nmod_poly_t* B, const *fq_nmod_ctx_t* ctx)

Either sets $f = 1$ and G to the greatest common divisor of A and B or sets f to a factor of the modulus of **ctx**.

slong **_fq_nmod_poly_xgcd**(*fq_nmod_struct* *G, *fq_nmod_struct* *S, *fq_nmod_struct* *T, const *fq_nmod_struct* *A, *slong* lenA, const *fq_nmod_struct* *B, *slong* lenB, const *fq_nmod_ctx_t* ctx)

Computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

void **fq_nmod_poly_xgcd**(*fq_nmod_poly_t* G, *fq_nmod_poly_t* S, *fq_nmod_poly_t* T, const *fq_nmod_poly_t* A, const *fq_nmod_poly_t* B, const *fq_nmod_ctx_t* ctx)

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S \cdot A + T \cdot B = G$. The length of S will be at most **lenB** and the length of T will be at most **lenA**.

slong **_fq_nmod_poly_xgcd_euclidean_f**(*fq_nmod_t* f, *fq_nmod_struct* *G, *fq_nmod_struct* *S, *fq_nmod_struct* *T, const *fq_nmod_struct* *A, *slong* lenA, const *fq_nmod_struct* *B, *slong* lenB, const *fq_nmod_ctx_t* ctx)

Either sets $f = 1$ and computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$; otherwise, sets f to a non-trivial factor of the modulus of **ctx** and leaves G , S , and T undefined. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fq_nmod_poly_xgcd_euclidean_f(fq_nmod_t f, fq_nmod_poly_t G, fq_nmod_poly_t S,
                                   fq_nmod_poly_t T, const fq_nmod_poly_t A, const
                                   fq_nmod_poly_t B, const fq_nmod_ctx_t ctx)
```

Either sets $f = 1$ and computes the GCD of A and B or sets f to a non-trivial factor of the modulus of ctx .

If the GCD is computed, polynomials S and T are computed such that $S \cdot A + T \cdot B = G$; otherwise, they are undefined. The length of S will be at most $\text{len}B$ and the length of T will be at most $\text{len}A$.

The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

11.14.17 Divisibility testing

```
int _fq_nmod_poly_divides(fq_nmod_struct *Q, const fq_nmod_struct *A, slong lenA, const
                          fq_nmod_struct *B, slong lenB, const fq_nmod_t invB, const
                          fq_nmod_ctx_t ctx)
```

Returns 1 if $(B, \text{len}B)$ divides $(A, \text{len}A)$ exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\text{len}(A) \geq \text{len}(B) > 0$ and that Q has space for $\text{len}(A) - \text{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fq_nmod_poly_divides(fq_nmod_poly_t Q, const fq_nmod_poly_t A, const fq_nmod_poly_t B,
                        const fq_nmod_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

11.14.18 Derivative

```
void _fq_nmod_poly_derivative(fq_nmod_struct *rop, const fq_nmod_struct *op, slong len, const
                              fq_nmod_ctx_t ctx)
```

Sets $(\text{rop}, \text{len} - 1)$ to the derivative of (op, len) . Also handles the cases where len is 0 or 1 correctly. Supports aliasing of rop and op .

```
void fq_nmod_poly_derivative(fq_nmod_poly_t rop, const fq_nmod_poly_t op, const
                              fq_nmod_ctx_t ctx)
```

Sets rop to the derivative of op .

11.14.19 Square root

```
void _fq_nmod_poly_invsqrt_series(fq_nmod_struct *g, const fq_nmod_struct *h, slong n,
                                fq_nmod_ctx_t mod)
```

Set the first n terms of g to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n . Aliasing is not permitted.

```
void fq_nmod_poly_invsqrt_series(fq_nmod_poly_t g, const fq_nmod_poly_t h, slong n,
                                fq_nmod_ctx_t ctx)
```

Set g to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
void _fq_nmod_poly_sqrt_series(fq_nmod_struct *g, const fq_nmod_struct *h, slong n,
                              fq_nmod_ctx_t ctx)
```

Set the first n terms of g to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n . Aliasing is not permitted.

```
void fq_nmod_poly_sqrt_series(fq_nmod_poly_t g, const fq_nmod_poly_t h, slong n,
                              fq_nmod_ctx_t ctx)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
int _fq_nmod_poly_sqrt(fq_nmod_struct *s, const fq_nmod_struct *p, slong n, fq_nmod_ctx_t
                      mod)
```

If (p, n) is a perfect square, sets $(s, n / 2 + 1)$ to a square root of p and returns 1. Otherwise returns 0.

```
int fq_nmod_poly_sqrt(fq_nmod_poly_t s, const fq_nmod_poly_t p, fq_nmod_ctx_t mod)
```

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

11.14.20 Evaluation

```
void _fq_nmod_poly_evaluate_fq_nmod(fq_nmod_t rop, const fq_nmod_struct *op, slong len, const
                                    fq_nmod_t a, const fq_nmod_ctx_t ctx)
```

Sets rop to (op, len) evaluated at a .

Supports zero padding. There are no restrictions on len , that is, len is allowed to be zero, too.

```
void fq_nmod_poly_evaluate_fq_nmod(fq_nmod_t rop, const fq_nmod_poly_t f, const fq_nmod_t a,
                                   const fq_nmod_ctx_t ctx)
```

Sets rop to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

11.14.21 Composition

```
void _fq_nmod_poly_compose(fq_nmod_struct *rop, const fq_nmod_struct *op1, slong len1, const
                           fq_nmod_struct *op2, slong len2, const fq_nmod_ctx_t ctx)
```

Sets rop to the composition of $(op1, len1)$ and $(op2, len2)$.

Assumes that rop has space for $(len1-1)*(len2-1) + 1$ coefficients. Assumes that $op1$ and $op2$ are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_nmod_poly_compose(fq_nmod_poly_t rop, const fq_nmod_poly_t op1, const
                          fq_nmod_poly_t op2, const fq_nmod_ctx_t ctx)
```

Sets rop to the composition of $op1$ and $op2$. To be precise about the order of composition, denoting rop , $op1$, and $op2$ by f , g , and h , respectively, sets $f(t) = g(h(t))$.

```
void _fq_nmod_poly_compose_mod_horner(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf,
                                       const fq_nmod_struct *g, const fq_nmod_struct *h, slong
                                       lenh, const fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_nmod_poly_compose_mod_horner(fq_nmod_poly_t res, const fq_nmod_poly_t f, const
                                     fq_nmod_poly_t g, const fq_nmod_poly_t h, const
                                     fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fq_nmod_poly_compose_mod_horner_preinv(fq_nmod_struct *res, const fq_nmod_struct *f,
                                             slong lenf, const fq_nmod_struct *g, const
                                             fq_nmod_struct *h, slong lenh, const
                                             fq_nmod_struct *hinv, slong lenhiv, const
                                             fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require **hinv** to be the inverse of the reverse of **h**. The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_nmod_poly_compose_mod_horner_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t f,
                                             const fq_nmod_poly_t g, const fq_nmod_poly_t
                                             h, const fq_nmod_poly_t hinv, const
                                             fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require **hinv** to be the inverse of the reverse of **h**. The algorithm used is Horner's rule.

```
void _fq_nmod_poly_compose_mod_brent_kung(fq_nmod_struct *res, const fq_nmod_struct *f, slong
lenf, const fq_nmod_struct *g, const fq_nmod_struct
*h, slong lenh, const fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_nmod_poly_compose_mod_brent_kung(fq_nmod_poly_t res, const fq_nmod_poly_t f, const
fq_nmod_poly_t g, const fq_nmod_poly_t h, const
fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_nmod_poly_compose_mod_brent_kung_preinv(fq_nmod_struct *res, const fq_nmod_struct
*f, slong lenf, const fq_nmod_struct *g,
const fq_nmod_struct *h, slong lenh, const
fq_nmod_struct *hinv, slong lenhiv, const
fq_nmod_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require **hinv** to be the inverse of the reverse of **h**. The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.


```
void fq_nmod_poly_compose_mod_brent_kung_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t
                                                f, const fq_nmod_poly_t g, const
                                                fq_nmod_poly_t h, const fq_nmod_poly_t
                                                hinv, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_nmod_poly_compose_mod(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf, const
                               fq_nmod_struct *g, const fq_nmod_struct *h, slong lenh, const
                               fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fq_nmod_poly_compose_mod(fq_nmod_poly_t res, const fq_nmod_poly_t f, const
                               fq_nmod_poly_t g, const fq_nmod_poly_t h, const fq_nmod_ctx_t
                               ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero.

```
void _fq_nmod_poly_compose_mod_preinv(fq_nmod_struct *res, const fq_nmod_struct *f, slong lenf,
                                       const fq_nmod_struct *g, const fq_nmod_struct *h, slong
                                       lenh, const fq_nmod_struct *hinv, slong lenhinv, const
                                       fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

```
void fq_nmod_poly_compose_mod_preinv(fq_nmod_poly_t res, const fq_nmod_poly_t f, const
                                       fq_nmod_poly_t g, const fq_nmod_poly_t h, const
                                       fq_nmod_poly_t hinv, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h .

```
void _fq_nmod_poly_reduce_matrix_mod_poly(fq_nmod_mat_t A, const fq_nmod_mat_t B, const
                                           fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Sets the i th row of A to the reduction of the i th row of B modulo f for $i = 1, \dots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fq_nmod_poly_precompute_matrix(fq_nmod_mat_t A, const fq_nmod_struct *f, const
                                      fq_nmod_struct *g, slong leng, const fq_nmod_struct
                                      *ginv, slong lenginv, const fq_nmod_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g and g to be nonzero.

```
void fq_nmod_poly_precompute_matrix(fq_nmod_mat_t A, const fq_nmod_poly_t f, const
                                     fq_nmod_poly_t g, const fq_nmod_poly_t ginv, const
                                     fq_nmod_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g .

```
void _fq_nmod_poly_compose_mod_brent_kung_precomp_preinv(fq_nmod_struct *res, const
                                                           fq_nmod_struct *f, slong lenf,
                                                           const fq_nmod_mat_t A, const
                                                           fq_nmod_struct *h, slong lenh,
                                                           const fq_nmod_struct *hinv, slong
                                                           lenhinv, const fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. We require that the i th

row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_nmod_poly_compose_mod_brent_kung_precomp_preinv(fq_nmod_poly_t res, const
                                                         fq_nmod_poly_t f, const
                                                         fq_nmod_mat_t A, const
                                                         fq_nmod_poly_t h, const
                                                         fq_nmod_poly_t hinv, const
                                                         fq_nmod_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h .

11.14.22 Output

```
int _fq_nmod_poly_fprint_pretty(FILE *file, const fq_nmod_struct *poly, slong len, const char *x,
                                const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_fprint_pretty(FILE *file, const fq_nmod_poly_t poly, const char *x, const
                                fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_nmod_poly_print_pretty(const fq_nmod_struct *poly, slong len, const char *x, const
                                fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `(poly, len)` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_print_pretty(const fq_nmod_poly_t poly, const char *x, const fq_nmod_ctx_t
                                ctx)
```

Prints the pretty representation of `poly` to `stdout`, using the string `x` to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_nmod_poly_fprint(FILE *file, const fq_nmod_struct *poly, slong len, const fq_nmod_ctx_t
                                ctx)
```

Prints the pretty representation of `(poly, len)` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_nmod_poly_fprint(FILE *file, const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)
```

Prints the pretty representation of `poly` to the stream `file`.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int fq_nmod_poly_print(const fq_nmod_struct *poly, slong len, const fq_nmod_ctx_t ctx)`
Prints the pretty representation of (poly, len) to `stdout`.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

`int fq_nmod_poly_print(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)`
Prints the representation of poly to `stdout`.
In case of success, returns a positive value. In case of failure, returns a non-positive value.

`char *fq_nmod_poly_get_str(const fq_nmod_struct *poly, slong len, const fq_nmod_ctx_t ctx)`
Returns the plain FLINT string representation of the polynomial (poly, len).

`char *fq_nmod_poly_get_str(const fq_nmod_poly_t poly, const fq_nmod_ctx_t ctx)`
Returns the plain FLINT string representation of the polynomial poly.

`char *fq_nmod_poly_get_str_pretty(const fq_nmod_struct *poly, slong len, const char *x, const fq_nmod_ctx_t ctx)`
Returns a pretty representation of the polynomial (poly, len) using the null-terminated string x as the variable name.

`char *fq_nmod_poly_get_str_pretty(const fq_nmod_poly_t poly, const char *x, const fq_nmod_ctx_t ctx)`
Returns a pretty representation of the polynomial poly using the null-terminated string x as the variable name

11.14.23 Inflation and deflation

`void fq_nmod_poly_inflate(fq_nmod_poly_t result, const fq_nmod_poly_t input, ulong inflation, const fq_nmod_ctx_t ctx)`
Sets `result` to the inflated polynomial $p(x^n)$ where p is given by `input` and n is given by `inflation`.

`void fq_nmod_poly_deflate(fq_nmod_poly_t result, const fq_nmod_poly_t input, ulong deflation, const fq_nmod_ctx_t ctx)`
Sets `result` to the deflated polynomial $p(x^{1/n})$ where p is given by `input` and n is given by `deflation`. Requires $n > 0$.

ulong `fq_nmod_poly_deflation(const fq_nmod_poly_t input, const fq_nmod_ctx_t ctx)`
Returns the largest integer by which `input` can be deflated. As special cases, returns 0 if `input` is the zero polynomial and 1 if `input` is a constant polynomial.

11.15 fq_nmod_poly_factor.h – factorisation of univariate polynomials over finite fields (word-size characteristic)

11.15.1 Types, macros and constants

`type fq_nmod_poly_factor_struct`

`type fq_nmod_poly_factor_t`

11.15.2 Memory Management

void **fq_nmod_poly_factor_init**(*fq_nmod_poly_factor_t* fac, const *fq_nmod_ctx_t* ctx)

Initialises *fac* for use. An *fq_nmod_poly_factor_t* represents a polynomial in factorised form as a product of polynomials with associated exponents.

void **fq_nmod_poly_factor_clear**(*fq_nmod_poly_factor_t* fac, const *fq_nmod_ctx_t* ctx)

Frees all memory associated with *fac*.

void **fq_nmod_poly_factor_realloc**(*fq_nmod_poly_factor_t* fac, *slong* alloc, const *fq_nmod_ctx_t* ctx)

Reallocates the factor structure to provide space for precisely *alloc* factors.

void **fq_nmod_poly_factor_fit_length**(*fq_nmod_poly_factor_t* fac, *slong* len, const *fq_nmod_ctx_t* ctx)

Ensures that the factor structure has space for at least *len* factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.15.3 Basic Operations

void **fq_nmod_poly_factor_set**(*fq_nmod_poly_factor_t* res, const *fq_nmod_poly_factor_t* fac, const *fq_nmod_ctx_t* ctx)

Sets *res* to the same factorisation as *fac*.

void **fq_nmod_poly_factor_print_pretty**(const *fq_nmod_poly_factor_t* fac, const char *var, const *fq_nmod_ctx_t* ctx)

Pretty-prints the entries of *fac* to standard output.

void **fq_nmod_poly_factor_print**(const *fq_nmod_poly_factor_t* fac, const *fq_nmod_ctx_t* ctx)

Prints the entries of *fac* to standard output.

void **fq_nmod_poly_factor_insert**(*fq_nmod_poly_factor_t* fac, const *fq_nmod_poly_t* poly, *slong* exp, const *fq_nmod_ctx_t* ctx)

Inserts the factor *poly* with multiplicity *exp* into the factorisation *fac*.

If *fac* already contains *poly*, then *exp* simply gets added to the exponent of the existing entry.

void **fq_nmod_poly_factor_concat**(*fq_nmod_poly_factor_t* res, const *fq_nmod_poly_factor_t* fac, const *fq_nmod_ctx_t* ctx)

Concatenates two factorisations.

This is equivalent to calling *fq_nmod_poly_factor_insert()* repeatedly with the individual factors of *fac*.

Does not support aliasing between *res* and *fac*.

void **fq_nmod_poly_factor_pow**(*fq_nmod_poly_factor_t* fac, *slong* exp, const *fq_nmod_ctx_t* ctx)

Raises *fac* to the power *exp*.

ulong **fq_nmod_poly_remove**(*fq_nmod_poly_t* f, const *fq_nmod_poly_t* p, const *fq_nmod_ctx_t* ctx)

Removes the highest possible power of *p* from *f* and returns the exponent.

11.15.4 Irreducibility Testing

int `fq_nmod_poly_is_irreducible`(const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Returns 1 if the polynomial *f* is irreducible, otherwise returns 0.

int `fq_nmod_poly_is_irreducible_ddf`(const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int `fq_nmod_poly_is_irreducible_ben_or`(const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

int `_fq_nmod_poly_is_squarefree`(const *fq_nmod_struct* *f, *slong* len, const *fq_nmod_ctx_t* ctx)

Returns 1 if (*f*, *len*) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int `fq_nmod_poly_is_squarefree`(const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Returns 1 if *f* is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.15.5 Factorisation

int `fq_nmod_poly_factor_equal_deg_prob`(*fq_nmod_poly_t* factor, *flint_rand_t* state, const *fq_nmod_poly_t* pol, *slong* d, const *fq_nmod_ctx_t* ctx)

Probabilistic equal degree factorisation of *pol* into irreducible factors of degree *d*. If it passes, a factor is placed in *factor* and 1 is returned, otherwise 0 is returned and the value of *factor* is undetermined.

Requires that *pol* be monic, non-constant and squarefree.

void `fq_nmod_poly_factor_equal_deg`(*fq_nmod_poly_factor_t* factors, const *fq_nmod_poly_t* pol, *slong* d, const *fq_nmod_ctx_t* ctx)

Assuming *pol* is a product of irreducible factors all of degree *d*, finds all those factors and places them in *factors*. Requires that *pol* be monic, non-constant and squarefree.

void `fq_nmod_poly_factor_split_single`(*fq_nmod_poly_t* linfactor, const *fq_nmod_poly_t* input, const *fq_nmod_ctx_t* ctx)

Assuming *input* is a product of factors all of degree 1, finds a single linear factor of *input* and places it in *linfactor*. Requires that *input* be monic and non-constant.

void `fq_nmod_poly_factor_distinct_deg`(*fq_nmod_poly_factor_t* res, const *fq_nmod_poly_t* poly, *slong* *const *degs, const *fq_nmod_ctx_t* ctx)

Factorises a monic non-constant squarefree polynomial *poly* of degree *n* into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of *poly* of degree *d*. Factors are stored in *res*, associated powers of irreducible polynomials are stored in *degs* in the same order as factors.

Requires that *degs* have enough space for irreducible polynomials' powers (maximum space required is $n * \text{sizeof}(\text{slong})$).

void `fq_nmod_poly_factor_squarefree`(*fq_nmod_poly_factor_t* res, const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Sets *res* to a squarefree factorization of *f*.

void `fq_nmod_poly_factor`(*fq_nmod_poly_factor_t* res, *fq_nmod_t* lead, const *fq_nmod_poly_t* f, const *fq_nmod_ctx_t* ctx)

Factorises a non-constant polynomial *f* into monic irreducible factors choosing the best algorithm for given modulo and degree. The output *lead* is set to the leading coefficient of *f* upon return. Choice of algorithm is based on heuristic measurements.

```
void fq_nmod_poly_factor_cantor_zassenhaus(fq_nmod_poly_factor_t res, const fq_nmod_poly_t
                                           f, const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void fq_nmod_poly_factor_kaltofen_shoup(fq_nmod_poly_factor_t res, const fq_nmod_poly_t
                                         poly, const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step.

```
void fq_nmod_poly_factor_berlekamp(fq_nmod_poly_factor_t factors, const fq_nmod_poly_t f,
                                    const fq_nmod_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors using the Berlekamp algorithm.

```
void fq_nmod_poly_factor_with_berlekamp(fq_nmod_poly_factor_t res, fq_nmod_t leading_coeff,
                                         const fq_nmod_poly_t f, const fq_nmod_ctx_t ctx)
```

Factorises a general polynomial f into monic irreducible factors and sets `leading_coeff` to the leading coefficient of f , or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

```
void fq_nmod_poly_factor_with_cantor_zassenhaus(fq_nmod_poly_factor_t res, fq_nmod_t
                                                leading_coeff, const fq_nmod_poly_t f, const
                                                fq_nmod_ctx_t ctx)
```

Factorises a general polynomial f into monic irreducible factors and sets `leading_coeff` to the leading coefficient of f , or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
void fq_nmod_poly_factor_with_kaltofen_shoup(fq_nmod_poly_factor_t res, fq_nmod_t
                                              leading_coeff, const fq_nmod_poly_t f, const
                                              fq_nmod_ctx_t ctx)
```

Factorises a general polynomial f into monic irreducible factors and sets `leading_coeff` to the leading coefficient of f , or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
void fq_nmod_poly_iterated_frobenius_preinv(fq_nmod_poly_t *rop, slong n, const
                                             fq_nmod_poly_t v, const fq_nmod_poly_t vinv,
                                             const fq_nmod_ctx_t ctx)
```

Sets `rop[i]` to be $x^{q^i} \bmod v$ for $0 \leq i < n$.

It is required that `vinv` is the inverse of the reverse of $v \bmod x^{\text{len}v}$.

11.15.6 Root Finding

void **fq_nmod_poly_roots**(*fq_nmod_poly_factor_t* r, const *fq_nmod_poly_t* f, int with_multiplicity, const *fq_nmod_ctx_t* ctx)

Fill *r* with factors of the form $x - r_i$ where the r_i are the distinct roots of a nonzero f in F_q . If *with_multiplicity* is zero, the exponent e_i of the factor $x - r_i$ is 1. Otherwise, it is the largest e_i such that $(x - r_i)^{e_i}$ divides f . This function throws if f is zero, but is otherwise always successful.

11.16 fq_nmod_embed.h – Computing isomorphisms and embeddings of finite fields

void **fq_nmod_embed_gens**(*fq_nmod_t* gen_sub, *fq_nmod_t* gen_sup, *nmod_poly_t* minpoly, const *fq_nmod_ctx_t* sub_ctx, const *fq_nmod_ctx_t* sup_ctx)

Given two contexts *sub_ctx* and *sup_ctx*, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute:

- an element *gen_sub* in *sub_ctx* such that *gen_sub* generates the finite field defined by *sub_ctx*,
- its minimal polynomial *minpoly*,
- a root *gen_sup* of *minpoly* inside the field defined by *sup_ctx*.

These data uniquely define an embedding of *sub_ctx* into *sup_ctx*.

void **_fq_nmod_embed_gens_naive**(*fq_nmod_t* gen_sub, *fq_nmod_t* gen_sup, *nmod_poly_t* minpoly, const *fq_nmod_ctx_t* sub_ctx, const *fq_nmod_ctx_t* sup_ctx)

Given two contexts *sub_ctx* and *sup_ctx*, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute an embedding of *sub_ctx* into *sup_ctx* defined as follows:

- *gen_sub* is the canonical generator of *sub_ctx* (i.e., the class of X),
- *minpoly* is the defining polynomial of *sub_ctx*,
- *gen_sup* is a root of *minpoly* inside the field defined by *sup_ctx*.

void **fq_nmod_embed_matrices**(*nmod_mat_t* embed, *nmod_mat_t* project, const *fq_nmod_t* gen_sub, const *fq_nmod_ctx_t* sub_ctx, const *fq_nmod_t* gen_sup, const *fq_nmod_ctx_t* sup_ctx, const *nmod_poly_t* gen_minpoly)

Given:

- two contexts *sub_ctx* and *sup_ctx*, of respective degrees m and n , such that m divides n ;
- a generator *gen_sub* of *sub_ctx*, its minimal polynomial *gen_minpoly*, and a root *gen_sup* of *gen_minpoly* in *sup_ctx*, as returned by `fq_nmod_embed_gens`;

Compute:

- the $n \times m$ matrix *embed* mapping *gen_sub* to *gen_sup*, and all their powers accordingly;
- an $m \times n$ matrix *project* such that *project* $\times$ *embed* is the $m \times m$ identity matrix.

void **fq_nmod_embed_trace_matrix**(*nmod_mat_t* res, const *nmod_mat_t* basis, const *fq_nmod_ctx_t* sub_ctx, const *fq_nmod_ctx_t* sup_ctx)

Given:

- two contexts *sub_ctx* and *sup_ctx*, of degrees m and n , such that m divides n ;
- an $n \times m$ matrix *basis* that maps *sub_ctx* to an isomorphic subfield in *sup_ctx*;

Compute the $m \times n$ matrix of the trace from *sup_ctx* to *sub_ctx*.

This matrix is computed as

```
embed_dual_to_mono_matrix(_, sub_ctx) × basist × embed_mono_to_dual_matrix(_,
sup_ctx)}.
```

Note: if $m = n$, `basis` represents a Frobenius, and the result is its inverse matrix.

```
void fq_nmod_embed_composition_matrix(nmod_mat_t matrix, const fq_nmod_t gen, const
fq_nmod_ctx_t ctx)
```

Compute the *composition matrix* of `gen`.

For an element $a \in \mathbf{F}_{p^n}$, its composition matrix is the matrix whose columns are $a^0, a^1, \dots, a^{n-1}$.

```
void fq_nmod_embed_composition_matrix_sub(nmod_mat_t matrix, const fq_nmod_t gen, const
fq_nmod_ctx_t ctx, slong trunc)
```

Compute the *composition matrix* of `gen`, truncated to `trunc` columns.

```
void fq_nmod_embed_mul_matrix(nmod_mat_t matrix, const fq_nmod_t gen, const fq_nmod_ctx_t
ctx)
```

Compute the *multiplication matrix* of `gen`.

For an element a in $\mathbf{F}_{p^n} = \mathbf{F}_p[x]$, its multiplication matrix is the matrix whose columns are $a, ax, \dots, ax^{n-1}$.

```
void fq_nmod_embed_mono_to_dual_matrix(nmod_mat_t res, const fq_nmod_ctx_t ctx)
```

Compute the change of basis matrix from the monomial basis of `ctx` to its dual basis.

```
void fq_nmod_embed_dual_to_mono_matrix(nmod_mat_t res, const fq_nmod_ctx_t ctx)
```

Compute the change of basis matrix from the dual basis of `ctx` to its monomial basis.

```
void fq_nmod_modulus_pow_series_inv(nmod_poly_t res, const fq_nmod_ctx_t ctx, slong trunc)
```

Compute the power series inverse of the reverse of the modulus of `ctx` up to $O(x^{\text{trunc}})$.

```
void fq_nmod_modulus_derivative_inv(fq_nmod_t m_prime, fq_nmod_t m_prime_inv, const
fq_nmod_ctx_t ctx)
```

Compute the derivative `m_prime` of the modulus of `ctx` as an element of `ctx`, and its inverse `m_prime_inv`.

11.17 fq_nmod_mpoly.h – multivariate polynomials over finite fields of word-sized characteristic

The exponents follow the `mpoly` interface. No references to the coefficients are available.

11.17.1 Types, macros and constants

```
type fq_nmod_mpoly_struct
```

A structure holding a multivariate polynomial over a finite field of word-sized characteristic.

```
type fq_nmod_mpoly_t
```

An array of length 1 of `fq_nmod_mpoly_struct`.

```
type fq_nmod_mpoly_ctx_struct
```

Context structure representing the parent ring of an `fq_nmod_mpoly`.

```
type fq_nmod_mpoly_ctx_t
```

An array of length 1 of `fq_nmod_mpoly_ctx_struct`.

11.17.2 Context object

void **fq_nmod_mpoly_ctx_init**(*fq_nmod_mpoly_ctx_t* ctx, *slong* nvars, const *ordering_t* ord, const *fq_nmod_ctx_t* fqctx)

Initialise a context object for a polynomial ring with the given number of variables and the given ordering. It will have coefficients in the finite field *fqctx*. The possibilities for the ordering are ORD_LEX, ORD_DEGLEX and ORD_DEGREVLEX.

slong **fq_nmod_mpoly_ctx_nvars**(const *fq_nmod_mpoly_ctx_t* ctx)

Return the number of variables used to initialize the context.

ordering_t **fq_nmod_mpoly_ctx_ord**(const *fq_nmod_mpoly_ctx_t* ctx)

Return the ordering used to initialize the context.

void **fq_nmod_mpoly_ctx_clear**(*fq_nmod_mpoly_ctx_t* ctx)

Release any space allocated by an *ctx*.

11.17.3 Memory management

void **fq_nmod_mpoly_init**(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Initialise *A* for use with the given an initialised context object. Its value is set to zero.

void **fq_nmod_mpoly_init2**(*fq_nmod_mpoly_t* A, *slong* alloc, const *fq_nmod_mpoly_ctx_t* ctx)

Initialise *A* for use with the given an initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and at least MPOLY_MIN_BITS bits for the exponents.

void **fq_nmod_mpoly_init3**(*fq_nmod_mpoly_t* A, *slong* alloc, *flint_bitcnt_t* bits, const *fq_nmod_mpoly_ctx_t* ctx)

Initialise *A* for use with the given an initialised context object. Its value is set to zero. It is allocated with space for *alloc* terms and *bits* bits for the exponents.

void **fq_nmod_mpoly_fit_length**(*fq_nmod_mpoly_t* A, *slong* len, const *fq_nmod_mpoly_ctx_t* ctx)

Ensure that *A* has space for at least *len* terms.

void **fq_nmod_mpoly_realloc**(*fq_nmod_mpoly_t* A, *slong* alloc, const *fq_nmod_mpoly_ctx_t* ctx)

Reallocate *A* to have space for *alloc* terms. Assumes the current length of the polynomial is not greater than *alloc*.

void **fq_nmod_mpoly_clear**(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Release any space allocated for *A*.

11.17.4 Input/Output

The variable strings in *x* start with the variable of most significance at index 0. If *x* is NULL, the variables are named *x1*, *x2*, etc.

char ***fq_nmod_mpoly_get_str_pretty**(const *fq_nmod_mpoly_t* A, const char ***x*, const *fq_nmod_mpoly_ctx_t* ctx)

Return a string, which the user is responsible for cleaning up, representing *A*, given an array of variable strings *x*.

int **fq_nmod_mpoly_fprint_pretty**(FILE **file*, const *fq_nmod_mpoly_t* A, const char ***x*, const *fq_nmod_mpoly_ctx_t* ctx)

Print a string representing *A* to *file*.

int **fq_nmod_mpoly_print_pretty**(const *fq_nmod_mpoly_t* A, const char ***x*, const *fq_nmod_mpoly_ctx_t* ctx)

Print a string representing *A* to *stdout*.

```
int fq_nmod_mpoly_set_str_pretty(fq_nmod_mpoly_t A, const char *str, const char **x, const
                                fq_nmod_mpoly_ctx_t ctx)
```

Set A to the polynomial in the null-terminates string str given an array x of variable strings. If parsing str fails, A is set to zero, and -1 is returned. Otherwise, 0 is returned. The operations $+$, $-$, $*$, and $/$ are permitted along with integers and the variables in x . The character $\wedge$ must be immediately followed by the (integer) exponent. If any division is not exact, parsing fails.

11.17.5 Basic manipulation

```
void fq_nmod_mpoly_gen(fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to the variable of index var , where $var = 0$ corresponds to the variable with the most significance with respect to the ordering.

```
int fq_nmod_mpoly_is_gen(const fq_nmod_mpoly_t A, slong var, const fq_nmod_mpoly_ctx_t ctx)
```

If $var \geq 0$, return 1 if A is equal to the var -th generator, otherwise return 0 . If $var < 0$, return 1 if the polynomial is equal to any generator, otherwise return 0 .

```
void fq_nmod_mpoly_set(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const
                      fq_nmod_mpoly_ctx_t ctx)
```

Set A to B .

```
int fq_nmod_mpoly_equal(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const
                       fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if A is equal to B , else return 0 .

```
void fq_nmod_mpoly_swap(fq_nmod_mpoly_t A, fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t
                       ctx)
```

Efficiently swap A and B .

11.17.6 Constants

```
int fq_nmod_mpoly_is_fq_nmod(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if A is a constant, else return 0 .

```
void fq_nmod_mpoly_get_fq_nmod(fq_nmod_t c, const fq_nmod_mpoly_t A, const
                              fq_nmod_mpoly_ctx_t ctx)
```

Assuming that A is a constant, set c to this constant. This function throws if A is not a constant.

```
void fq_nmod_mpoly_set_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_t c, const
                              fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_set_ui(fq_nmod_mpoly_t A, ulong c, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to the constant c .

```
void fq_nmod_mpoly_set_fq_nmod_gen(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to the constant given by `fq_nmod_gen()`.

```
void fq_nmod_mpoly_zero(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to the constant 0 .

```
void fq_nmod_mpoly_one(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to the constant 1 .

```
int fq_nmod_mpoly_equal_fq_nmod(const fq_nmod_mpoly_t A, const fq_nmod_t c, const
                                fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if A is equal to the constant c , else return 0 .

int `fq_nmod_mpoly_is_zero`(const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is the constant 0, else return 0.

int `fq_nmod_mpoly_is_one`(const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Return 1 if *A* is the constant 1, else return 0.

11.17.7 Degrees

int `fq_nmod_mpoly_degrees_fit_si`(const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Return 1 if the degrees of *A* with respect to each variable fit into an `slong`, otherwise return 0.

void `fq_nmod_mpoly_degrees_fmpz`(*fmpz* *degs, const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

void `fq_nmod_mpoly_degrees_si`(*slong* *degs, const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Set *degs* to the degrees of *A* with respect to each variable. If *A* is zero, all degrees are set to -1 .

void `fq_nmod_mpoly_degree_fmpz`(*fmpz_t* deg, const *fq_nmod_mpoly_t* A, *slong* var, const *fq_nmod_mpoly_ctx_t* ctx)

slong `fq_nmod_mpoly_degree_si`(const *fq_nmod_mpoly_t* A, *slong* var, const *fq_nmod_mpoly_ctx_t* ctx)

Either return or set *deg* to the degree of *A* with respect to the variable of index *var*. If *A* is zero, the degree is defined to be -1 .

int `fq_nmod_mpoly_total_degree_fits_si`(const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Return 1 if the total degree of *A* fits into an `slong`, otherwise return 0.

void `fq_nmod_mpoly_total_degree_fmpz`(*fmpz_t* tdeg, const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

slong `fq_nmod_mpoly_total_degree_si`(const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

Either return or set *tdeg* to the total degree of *A*. If *A* is zero, the total degree is defined to be -1 .

void `fq_nmod_mpoly_used_vars`(int *used, const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_ctx_t* ctx)

For each variable index *i*, set *used*[*i*] to nonzero if the variable of index *i* appears in *A* and to zero otherwise.

11.17.8 Coefficients

void `fq_nmod_mpoly_get_coeff_fq_nmod_monomial`(*fq_nmod_t* c, const *fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* M, const *fq_nmod_mpoly_ctx_t* ctx)

Assuming that *M* is a monomial, set *c* to the coefficient of the corresponding monomial in *A*. This function throws if *M* is not a monomial.

void `fq_nmod_mpoly_set_coeff_fq_nmod_monomial`(*fq_nmod_mpoly_t* A, const *fq_nmod_t* c, const *fq_nmod_mpoly_t* M, const *fq_nmod_mpoly_ctx_t* ctx)

Assuming that *M* is a monomial, set the coefficient of the corresponding monomial in *A* to *c*. This function throws if *M* is not a monomial.

void `fq_nmod_mpoly_get_coeff_fq_nmod_fmpz`(*fq_nmod_t* c, const *fq_nmod_mpoly_t* A, *fmpz* *const *exp, const *fq_nmod_mpoly_ctx_t* ctx)

```
void fq_nmod_mpoly_get_coeff_fq_nmod_ui(fq_nmod_t c, const fq_nmod_mpoly_t A, const ulong
                                         *exp, const fq_nmod_mpoly_ctx_t ctx)
```

Set c to the coefficient of the monomial with exponent vector exp .

```
void fq_nmod_mpoly_set_coeff_fq_nmod_fmpz(fq_nmod_mpoly_t A, const fq_nmod_t c, fmpz
                                           *const *exp, const fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_set_coeff_fq_nmod_ui(fq_nmod_mpoly_t A, const fq_nmod_t c, const ulong
                                         *exp, const fq_nmod_mpoly_ctx_t ctx)
```

Set the coefficient of the monomial with exponent exp to c .

```
void fq_nmod_mpoly_get_coeff_vars_ui(fq_nmod_mpoly_t C, const fq_nmod_mpoly_t A, const
                                     slong *vars, const ulong *exps, slong length, const
                                     fq_nmod_mpoly_ctx_t ctx)
```

Set C to the coefficient of A with respect to the variables in $vars$ with powers in the corresponding array $exps$. Both $vars$ and $exps$ point to array of length $length$. It is assumed that $0 < length \leq nvars(A)$ and that the variables in $vars$ are distinct.

11.17.9 Comparison

```
int fq_nmod_mpoly_cmp(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const
                      fq_nmod_mpoly_ctx_t ctx)
```

Return 1 (resp. -1 , or 0) if A is after (resp. before, same as) B in some arbitrary but fixed total ordering of the polynomials. This ordering agrees with the usual ordering of monomials when A and B are both monomials.

11.17.10 Container operations

These functions deal with violations of the internal canonical representation. If a term index is negative or not strictly less than the length of the polynomial, the function will throw.

```
int fq_nmod_mpoly_is_canonical(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if A is in canonical form. Otherwise, return 0. To be in canonical form, all of the terms must have nonzero coefficients, and the terms must be sorted from greatest to least.

```
slong fq_nmod_mpoly_length(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Return the number of terms in A . If the polynomial is in canonical form, this will be the number of nonzero coefficients.

```
void fq_nmod_mpoly_resize(fq_nmod_mpoly_t A, slong new_length, const fq_nmod_mpoly_ctx_t
                           ctx)
```

Set the length of A to new_length . Terms are either deleted from the end, or new zero terms are appended.

```
void fq_nmod_mpoly_get_term_coeff_fq_nmod(fq_nmod_t c, const fq_nmod_mpoly_t A, slong i,
                                           const fq_nmod_mpoly_ctx_t ctx)
```

Set c to the coefficient of the term of index i .

```
void fq_nmod_mpoly_set_term_coeff_ui(fq_nmod_mpoly_t A, slong i, ulong c, const
                                     fq_nmod_mpoly_ctx_t ctx)
```

Set the coefficient of the term of index i to c .

```
int fq_nmod_mpoly_term_exp_fits_si(const fq_nmod_mpoly_t A, slong i, const
                                   fq_nmod_mpoly_ctx_t ctx)
```

```
int fq_nmod_mpoly_term_exp_fits_ui(const fq_nmod_mpoly_t A, slong i, const
                                   fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if all entries of the exponent vector of the term of index i fit into an `slong` (resp. a `ulong`). Otherwise, return 0.

```
void fq_nmod_mpoly_get_term_exp_fmpz(fmpz **exp, const fq_nmod_mpoly_t A, slong i, const
                                     fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_get_term_exp_ui(ulong *exp, const fq_nmod_mpoly_t A, slong i, const
                                    fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_get_term_exp_si(slong *exp, const fq_nmod_mpoly_t A, slong i, const
                                    fq_nmod_mpoly_ctx_t ctx)
```

Set *exp* to the exponent vector of the term of index *i*. The `_ui` (resp. `_si`) version throws if any entry does not fit into a `ulong` (resp. `slong`).

```
ulong fq_nmod_mpoly_get_term_var_exp_ui(const fq_nmod_mpoly_t A, slong i, slong var, const
                                         fq_nmod_mpoly_ctx_t ctx)
```

```
slong fq_nmod_mpoly_get_term_var_exp_si(const fq_nmod_mpoly_t A, slong i, slong var, const
                                         fq_nmod_mpoly_ctx_t ctx)
```

Return the exponent of the variable *var* of the term of index *i*. This function throws if the exponent does not fit into a `ulong` (resp. `slong`).

```
void fq_nmod_mpoly_set_term_exp_fmpz(fq_nmod_mpoly_t A, slong i, fmpz *const *exp, const
                                     fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_set_term_exp_ui(fq_nmod_mpoly_t A, slong i, const ulong *exp, const
                                    fq_nmod_mpoly_ctx_t ctx)
```

Set the exponent of the term of index *i* to *exp*.

```
void fq_nmod_mpoly_get_term(fq_nmod_mpoly_t M, const fq_nmod_mpoly_t A, slong i, const
                            fq_nmod_mpoly_ctx_t ctx)
```

Set *M* to the term of index *i* in *A*.

```
void fq_nmod_mpoly_get_term_monomial(fq_nmod_mpoly_t M, const fq_nmod_mpoly_t A, slong i,
                                     const fq_nmod_mpoly_ctx_t ctx)
```

Set *M* to the monomial of the term of index *i* in *A*. The coefficient of *M* will be one.

```
void fq_nmod_mpoly_push_term_fq_nmod_fmpz(fq_nmod_mpoly_t A, const fq_nmod_t c, fmpz
                                           *const *exp, const fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_push_term_fq_nmod_ffmpz(fq_nmod_mpoly_t A, const fq_nmod_t c, const
                                             fmpz *exp, const fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_push_term_fq_nmod_ui(fq_nmod_mpoly_t A, const fq_nmod_t c, const ulong
                                          *exp, const fq_nmod_mpoly_ctx_t ctx)
```

Append a term to *A* with coefficient *c* and exponent vector *exp*. This function runs in constant average time.

```
void fq_nmod_mpoly_sort_terms(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Sort the terms of *A* into the canonical ordering dictated by the ordering in *ctx*. This function simply reorders the terms: It does not combine like terms, nor does it delete terms with coefficient zero. This function runs in linear time in the bit size of *A*.

```
void fq_nmod_mpoly_combine_like_terms(fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Combine adjacent like terms in *A* and delete terms with coefficient zero. If the terms of *A* were sorted to begin with, the result will be in canonical form. This function runs in linear time in the bit size of *A*.

```
void fq_nmod_mpoly_reverse(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const
                           fq_nmod_mpoly_ctx_t ctx)
```

Set *A* to the reversal of *B*.

11.17.11 Random generation

void `fq_nmod_mpoly_randtest_bound`(*fq_nmod_mpoly_t* A, *flint_rand_t* state, *slong* length, *ulong* exp_bound, const *fq_nmod_mpoly_ctx_t* ctx)

Generate a random polynomial with length up to *length* and exponents in the range $[0, \text{exp\_bound} - 1]$. The exponents of each variable are generated by calls to `n_randint(state, exp_bound)`.

void `fq_nmod_mpoly_randtest_bounds`(*fq_nmod_mpoly_t* A, *flint_rand_t* state, *slong* length, *ulong* *exp_bounds, const *fq_nmod_mpoly_ctx_t* ctx)

Generate a random polynomial with length up to *length* and exponents in the range $[0, \text{exp\_bounds}[i] - 1]$. The exponents of the variable of index *i* are generated by calls to `n_randint(state, exp_bounds[i])`.

void `fq_nmod_mpoly_randtest_bits`(*fq_nmod_mpoly_t* A, *flint_rand_t* state, *slong* length, *ulong* exp_bits, const *fq_nmod_mpoly_ctx_t* ctx)

Generate a random polynomial with length up to *length* and exponents whose packed form does not exceed the given bit count.

11.17.12 Addition/Subtraction

void `fq_nmod_mpoly_add_fq_nmod`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_t* C, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to $B + c$.

void `fq_nmod_mpoly_sub_fq_nmod`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_t* C, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to $B - c$.

void `fq_nmod_mpoly_add`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_mpoly_t* C, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to $B + C$.

void `fq_nmod_mpoly_sub`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_mpoly_t* C, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to $B - C$.

11.17.13 Scalar operations

void `fq_nmod_mpoly_neg`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to $-B$.

void `fq_nmod_mpoly_scalar_mul_fq_nmod`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_t* c, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to $B \times c$.

void `fq_nmod_mpoly_make_monic`(*fq_nmod_mpoly_t* A, const *fq_nmod_mpoly_t* B, const *fq_nmod_mpoly_ctx_t* ctx)

Set *A* to *B* divided by the leading coefficient of *B*. This throws if *B* is zero.

11.17.14 Differentiation

```
void fq_nmod_mpoly_derivative(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, slong var, const
                             fq_nmod_mpoly_ctx_t ctx)
```

Set A to the derivative of B with respect to the variable of index var .

11.17.15 Evaluation

These functions return 0 when the operation would imply unreasonable arithmetic.

```
void fq_nmod_mpoly_evaluate_all_fq_nmod(fq_nmod_t ev, const fq_nmod_mpoly_t A,
                                         fq_nmod_struct *const *vals, const
                                         fq_nmod_mpoly_ctx_t ctx)
```

Set ev the evaluation of A where the variables are replaced by the corresponding elements of the array $vals$.

```
void fq_nmod_mpoly_evaluate_one_fq_nmod(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, slong
                                         var, const fq_nmod_t val, const fq_nmod_mpoly_ctx_t
                                         ctx)
```

Set A to the evaluation of B where the variable of index var is replaced by val .

```
int fq_nmod_mpoly_compose_fq_nmod_poly(fq_nmod_poly_t A, const fq_nmod_mpoly_t B,
                                         fq_nmod_poly_struct *const *C, const
                                         fq_nmod_mpoly_ctx_t ctx)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . The context object of B is $ctxB$. Return 1 for success and 0 for failure.

```
int fq_nmod_mpoly_compose_fq_nmod_mpoly(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B,
                                         fq_nmod_mpoly_struct *const *C, const
                                         fq_nmod_mpoly_ctx_t ctxB, const
                                         fq_nmod_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variables are replaced by the corresponding elements of the array C . Both A and the elements of C have context object $ctxAC$, while B has context object $ctxB$. Neither A nor B is allowed to alias any other polynomial. Return 1 for success and 0 for failure.

```
void fq_nmod_mpoly_compose_fq_nmod_mpoly_gen(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B,
                                              const slong *c, const fq_nmod_mpoly_ctx_t
                                              ctxB, const fq_nmod_mpoly_ctx_t ctxAC)
```

Set A to the evaluation of B where the variable of index i in $ctxB$ is replaced by the variable of index $c[i]$ in $ctxAC$. The length of the array C is the number of variables in $ctxB$. If any $c[i]$ is negative, the corresponding variable of B is replaced by zero. Otherwise, it is expected that $c[i]$ is less than the number of variables in $ctxAC$.

11.17.16 Multiplication

```
void fq_nmod_mpoly_mul(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fq_nmod_mpoly_t
                      C, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to B times C .

11.17.17 Powering

These functions return 0 when the operation would imply unreasonable arithmetic.

```
int fq_nmod_mpoly_pow_fmpz(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, const fmpz_t k,
                           const fq_nmod_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

```
int fq_nmod_mpoly_pow_ui(fq_nmod_mpoly_t A, const fq_nmod_mpoly_t B, ulong k, const
                        fq_nmod_mpoly_ctx_t ctx)
```

Set A to B raised to the k -th power. Return 1 for success and 0 for failure.

11.17.18 Division

```
int fq_nmod_mpoly_divides(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const
                          fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

If A is divisible by B , set Q to the exact quotient and return 1. Otherwise, set Q to zero and return 0.

```
void fq_nmod_mpoly_div(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t
                      B, const fq_nmod_mpoly_ctx_t ctx)
```

Set Q to the quotient of A by B , discarding the remainder.

```
void fq_nmod_mpoly_divrem(fq_nmod_mpoly_t Q, fq_nmod_mpoly_t R, const fq_nmod_mpoly_t A,
                         const fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

Set Q and R to the quotient and remainder of A divided by B .

```
void fq_nmod_mpoly_divrem_ideal(fq_nmod_mpoly_struct **Q, fq_nmod_mpoly_t R, const
                               fq_nmod_mpoly_t A, fq_nmod_mpoly_struct *const *B, slong
                               len, const fq_nmod_mpoly_ctx_t ctx)
```

This function is as per `fq_nmod_mpoly_divrem()` except that it takes an array of divisor polynomials B and it returns an array of quotient polynomials Q . The number of divisor (and hence quotient) polynomials, is given by len .

11.17.19 Greatest Common Divisor

```
void fq_nmod_mpoly_term_content(fq_nmod_mpoly_t M, const fq_nmod_mpoly_t A, const
                               fq_nmod_mpoly_ctx_t ctx)
```

Set M to the GCD of the terms of A . If A is zero, M will be zero. Otherwise, M will be a monomial with coefficient one.

```
int fq_nmod_mpoly_content_vars(fq_nmod_mpoly_t g, const fq_nmod_mpoly_t A, slong *vars,
                              slong vars_length, const fq_nmod_mpoly_ctx_t ctx)
```

Set g to the GCD of the coefficients of A when viewed as a polynomial in the variables $vars$. Return 1 for success and 0 for failure. Upon success, g will be independent of the variables $vars$.

```
int fq_nmod_mpoly_gcd(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const fq_nmod_mpoly_t
                     B, const fq_nmod_mpoly_ctx_t ctx)
```

Try to set G to the monic GCD of A and B . The GCD of zero and zero is defined to be zero. If the return is 1 the function was successful. Otherwise the return is 0 and G is left untouched.

```
int fq_nmod_mpoly_gcd_cofactors(fq_nmod_mpoly_t G, fq_nmod_mpoly_t Abar,
                                fq_nmod_mpoly_t Bbar, const fq_nmod_mpoly_t A, const
                                fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

Do the operation of `fq_nmod_mpoly_gcd()` and also compute $Abar = A/G$ and $Bbar = B/G$ if successful.

```
int fq_nmod_mpoly_gcd_brown(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const
                           fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
int fq_nmod_mpoly_gcd_hensel(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const
                             fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
int fq_nmod_mpoly_gcd_zippel(fq_nmod_mpoly_t G, const fq_nmod_mpoly_t A, const
                             fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

Try to set G to the GCD of A and B using various algorithms.

```
int fq_nmod_mpoly_resultant(fq_nmod_mpoly_t R, const fq_nmod_mpoly_t A, const
                            fq_nmod_mpoly_t B, slong var, const fq_nmod_mpoly_ctx_t ctx)
```

Try to set R to the resultant of A and B with respect to the variable of index var .

```
int fq_nmod_mpoly_discriminant(fq_nmod_mpoly_t D, const fq_nmod_mpoly_t A, slong var, const
                               fq_nmod_mpoly_ctx_t ctx)
```

Try to set D to the discriminant of A with respect to the variable of index var .

11.17.20 Square Root

```
int fq_nmod_mpoly_sqrt(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const
                      fq_nmod_mpoly_ctx_t ctx)
```

If $Q^2 = A$ has a solution, set Q to a solution and return 1, otherwise return 0 and set Q to zero.

```
int fq_nmod_mpoly_is_square(const fq_nmod_mpoly_t A, const fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if A is a perfect square, otherwise return 0.

```
int fq_nmod_mpoly_quadratic_root(fq_nmod_mpoly_t Q, const fq_nmod_mpoly_t A, const
                                  fq_nmod_mpoly_t B, const fq_nmod_mpoly_ctx_t ctx)
```

If $Q^2 + AQ = B$ has a solution, set Q to a solution and return 1, otherwise return 0.

11.17.21 Univariate Functions

An `fq_nmod_mpoly_univar_t` holds a univariate polynomial in some main variable with `fq_nmod_mpoly_t` coefficients in the remaining variables. These functions are useful when one wants to rewrite an element of $\mathbb{F}_q[x_1, \dots, x_m]$ as an element of $(\mathbb{F}_q[x_1, \dots, x_{v-1}, x_{v+1}, \dots, x_m])[x_v]$ and vice versa.

```
void fq_nmod_mpoly_univar_init(fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_ctx_t ctx)
    Initialize  $A$ .
```

```
void fq_nmod_mpoly_univar_clear(fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_ctx_t ctx)
    Clear  $A$ .
```

```
void fq_nmod_mpoly_univar_swap(fq_nmod_mpoly_univar_t A, fq_nmod_mpoly_univar_t B,
                              const fq_nmod_mpoly_ctx_t ctx)
```

Swap A and B .

```
void fq_nmod_mpoly_to_univar(fq_nmod_mpoly_univar_t A, const fq_nmod_mpoly_t B, slong var,
                             const fq_nmod_mpoly_ctx_t ctx)
```

Set A to a univariate form of B by pulling out the variable of index var . The coefficients of A will still belong to the content ctx but will not depend on the variable of index var .

```
void fq_nmod_mpoly_from_univar(fq_nmod_mpoly_t A, const fq_nmod_mpoly_univar_t B, slong
                               var, const fq_nmod_mpoly_ctx_t ctx)
```

Set A to the normal form of B by putting in the variable of index var . This function is undefined if the coefficients of B depend on the variable of index var .

```
int fq_nmod_mpoly_univar_degree_fits_si(const fq_nmod_mpoly_univar_t A, const
                                         fq_nmod_mpoly_ctx_t ctx)
```

Return 1 if the degree of A with respect to the main variable fits an `slong`. Otherwise, return 0.

```
slong fq_nmod_mpoly_univar_length(const fq_nmod_mpoly_univar_t A, const
                                   fq_nmod_mpoly_ctx_t ctx)
```

Return the number of terms in A with respect to the main variable.

```
slong fq_nmod_mpoly_univar_get_term_exp_si(fq_nmod_mpoly_univar_t A, slong i, const
                                             fq_nmod_mpoly_ctx_t ctx)
```

Return the exponent of the term of index i of A .

```
void fq_nmod_mpoly_univar_get_term_coeff(fq_nmod_mpoly_t c, const
                                          fq_nmod_mpoly_univar_t A, slong i, const
                                          fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_univar_swap_term_coeff(fq_nmod_mpoly_t c, fq_nmod_mpoly_univar_t A,
                                            slong i, const fq_nmod_mpoly_ctx_t ctx)
```

Set (resp. swap) c to (resp. with) the coefficient of the term of index i of A .

11.18 fq_nmod_mpoly_factor.h – factorisation of multivariate polynomials over finite fields of word-sized characteristic

11.18.1 Types, macros and constants

```
type fq_nmod_mpoly_factor_struct
```

A struct for holding a factored polynomial. There is a single constant and a product of bases to corresponding exponents.

```
type fq_nmod_mpoly_factor_t
```

An array of length 1 of `fq_nmod_mpoly_factor_struct`.

11.18.2 Memory management

```
void fq_nmod_mpoly_factor_init(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)
```

Initialise f .

```
void fq_nmod_mpoly_factor_clear(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)
```

Clear f .

11.18.3 Basic manipulation

```
void fq_nmod_mpoly_factor_swap(fq_nmod_mpoly_factor_t f, fq_nmod_mpoly_factor_t g, const
                               fq_nmod_mpoly_ctx_t ctx)
```

Efficiently swap f and g .

```
slong fq_nmod_mpoly_factor_length(const fq_nmod_mpoly_factor_t f, const
                                   fq_nmod_mpoly_ctx_t ctx)
```

Return the length of the product in f .

```
void fq_nmod_mpoly_factor_get_constant_fq_nmod(fq_nmod_t c, const fq_nmod_mpoly_factor_t
                                                f, const fq_nmod_mpoly_ctx_t ctx)
```

Set c to the constant of f .

```
void fq_nmod_mpoly_factor_get_base(fq_nmod_mpoly_t p, const fq_nmod_mpoly_factor_t f, slong
                                  i, const fq_nmod_mpoly_ctx_t ctx)
```

```
void fq_nmod_mpoly_factor_swap_base(fq_nmod_mpoly_t p, const fq_nmod_mpoly_factor_t f,
                                     slong i, const fq_nmod_mpoly_ctx_t ctx)
```

Set (resp. swap) B to (resp. with) the base of the term of index i in A .

```
slong fq_nmod_mpoly_factor_get_exp_si(fq_nmod_mpoly_factor_t f, slong i, const
                                       fq_nmod_mpoly_ctx_t ctx)
```

Return the exponent of the term of index i in A . It is assumed to fit an `slong`.

```
void fq_nmod_mpoly_factor_sort(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_ctx_t ctx)
```

Sort the product of f first by exponent and then by base.

11.18.4 Factorisation

A return of 1 indicates that the function was successful. Otherwise, the return is 0 and f is undefined. None of these functions multiply f by A : f is simply set to a factorisation of A , and thus these functions should not depend on the initial value of the output f .

```
int fq_nmod_mpoly_factor_squarefree(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_t A,
                                     const fq_nmod_mpoly_ctx_t ctx)
```

Set f to a factorization of A where the bases are primitive and pairwise relatively prime. If the product of all irreducible factors with a given exponent is desired, it is recommended to call `fq_nmod_mpoly_factor_sort()` and then multiply the bases with the desired exponent.

```
int fq_nmod_mpoly_factor(fq_nmod_mpoly_factor_t f, const fq_nmod_mpoly_t A, const
                         fq_nmod_mpoly_ctx_t ctx)
```

Set f to a factorization of A where the bases are irreducible.

11.19 fq_zech.h – finite fields (Zech logarithm representation)

We represent an element of the finite field as a power of a generator for the multiplicative group of the finite field. In particular, we use a root of $f(x)$, where $f(X) \in \mathbf{F}_p[X]$ is a monic, irreducible polynomial of degree n , as a polynomial in $\mathbf{F}_p[X]$ of degree less than n . The underlying data structure is just an `ulong`.

The default choice for $f(X)$ is the Conway polynomial for the pair (p, n) , enabled by Frank Lübeck's data base of Conway polynomials using the `_nmod_poly_conway()` function. If a Conway polynomial is not available, then a random irreducible polynomial will be chosen for $f(X)$. Additionally, the user is able to supply their own $f(X)$.

We required that the order of the field fits inside of an `ulong`; however, it is recommended that $p^n < 2^{20}$ due to the time and memory needed to compute the Zech logarithm table.

11.19.1 Types, macros and constants

```
type fq_zech_ctx_struct
```

```
type fq_zech_ctx_t
```

```
type fq_zech_struct
```

```
type fq_zech_t
```

11.19.2 Context Management

void **fq_zech_ctx_init_ui**(*fq_zech_ctx_t* ctx, *ulong* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name **var** for the generator. By default, it will try use a Conway polynomial; if one is not available, a random primitive polynomial will be used.

Assumes that p is a prime and $p^d < 2^{\text{FLINT\_BITS}}$.

Assumes that the string **var** is a null-terminated string of length at least one.

int **_fq_zech_ctx_init_conway_ui**(*fq_zech_ctx_t* ctx, *ulong* p, *slong* d, const char *var)

Attempts to initialise the context for prime p and extension degree d , with name **var** for the generator using a Conway polynomial for the modulus.

Returns 1 if the Conway polynomial is in the database for the given size and the initialization is successful; otherwise, returns 0.

Assumes that p is a prime and $p^d < 2^{\text{FLINT\_BITS}}$.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_zech_ctx_init_conway_ui**(*fq_zech_ctx_t* ctx, *ulong* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name **var** for the generator using a Conway polynomial for the modulus.

Assumes that p is a prime and $p^d < 2^{\text{FLINT\_BITS}}$.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_zech_ctx_init_random_ui**(*fq_zech_ctx_t* ctx, *ulong* p, *slong* d, const char *var)

Initialises the context for prime p and extension degree d , with name **var** for the generator using a random primitive polynomial.

Assumes that p is a prime and $p^d < 2^{\text{FLINT\_BITS}}$.

Assumes that the string **var** is a null-terminated string of length at least one.

void **fq_zech_ctx_init_modulus**(*fq_zech_ctx_t* ctx, const *nmod_poly_t* modulus, const char *var)

Initialises the context for given **modulus** with name **var** for the generator.

Assumes that **modulus** is an primitive polynomial over $\mathbf{F}_p$. An exception is raised if a non-primitive modulus is detected.

Assumes that the string **var** is a null-terminated string of length at least one.

int **fq_zech_ctx_init_modulus_check**(*fq_zech_ctx_t* ctx, const *nmod_poly_t* modulus, const char *var)

As per the previous function, but returns 0 if the modulus was not primitive and 1 if the context was successfully initialised with the given modulus. No exception is raised.

void **fq_zech_ctx_init_fq_nmod_ctx**(*fq_zech_ctx_t* ctx, *fq_nmod_ctx_t* ctxn)

Initializes the context **ctx** to be the Zech representation for the finite field given by **ctxn**.

int **fq_zech_ctx_init_fq_nmod_ctx_check**(*fq_zech_ctx_t* ctx, *fq_nmod_ctx_t* ctxn)

As per the previous function but returns 0 if a non-primitive modulus is detected. Returns 0 if the Zech representation was successfully initialised.

void **fq_zech_ctx_init_randtest**(*fq_zech_ctx_t* ctx, *flint_rand_t* state, int type)

Initialises **ctx** to a random finite field, where the prime and degree is set according to **type**. If **type** is 0 the prime and degree may be large, else if **type** is 1 the degree is small but the prime may be large, else if **type** is 2 the prime is small but the degree may be large, else if **type** is 3 both prime and degree are small.

void **fq_zech_ctx_init_randtest_reducible**(*fq_zech_ctx_t* ctx, *flint_rand_t* state, int type)
 Since the Zech logarithm representation does not work with a non-irreducible modulus, this function does the same as *fq_zech_ctx_init_randtest()*.

void **fq_zech_ctx_clear**(*fq_zech_ctx_t* ctx)
 Clears all memory that has been allocated as part of the context.

const *nmod_poly_struct* ***fq_zech_ctx_modulus**(const *fq_zech_ctx_t* ctx)
 Returns a pointer to the modulus in the context.

slong **fq_zech_ctx_degree**(const *fq_zech_ctx_t* ctx)
 Returns the degree of the field extension $[\mathbf{F}_q : \mathbf{F}_p]$, which is equal to $\log_p q$.

ulong **fq_zech_ctx_prime**(const *fq_zech_ctx_t* ctx)
 Returns the prime p of the context.

void **fq_zech_ctx_order**(*fmpz_t* f, const *fq_zech_ctx_t* ctx)
 Sets f to be the size of the finite field.

ulong **fq_zech_ctx_order_ui**(const *fq_zech_ctx_t* ctx)
 Returns the size of the finite field.

int **fq_zech_ctx_fprint**(FILE *file, const *fq_zech_ctx_t* ctx)
 Prints the context information to $\{\texttt{file}\}$. Returns 1 for a success and a negative number for an error.

void **fq_zech_ctx_print**(const *fq_zech_ctx_t* ctx)
 Prints the context information to $\{\texttt{stdout}\}$.

11.19.3 Memory management

void **fq_zech_init**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)
 Initialises the element *rop*, setting its value to 0.

void **fq_zech_init2**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)
 Initialises *poly* with at least enough space for it to be an element of *ctx* and sets it to 0.

void **fq_zech_clear**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)
 Clears the element *rop*.

void **_fq_zech_sparse_reduce**(*nn_ptr* R, *slong* lenR, const *fq_zech_ctx_t* ctx)
 Reduces (R, lenR) modulo the polynomial f given by the modulus of *ctx*.

void **_fq_zech_dense_reduce**(*nn_ptr* R, *slong* lenR, const *fq_zech_ctx_t* ctx)
 Reduces (R, lenR) modulo the polynomial f given by the modulus of *ctx* using Newton division.

void **_fq_zech_reduce**(*nn_ptr* r, *slong* lenR, const *fq_zech_ctx_t* ctx)
 Reduces (R, lenR) modulo the polynomial f given by the modulus of *ctx*. Does either sparse or dense reduction based on *ctx*->*sparse_modulus*.

void **fq_zech_reduce**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)
 Reduces the polynomial *rop* as an element of $\mathbf{F}_p[X]/(f(X))$.

11.19.4 Basic arithmetic

void **fq_zech_add**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_t* op2, const *fq_zech_ctx_t* ctx)

Sets rop to the sum of op1 and op2.

void **fq_zech_sub**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_t* op2, const *fq_zech_ctx_t* ctx)

Sets rop to the difference of op1 and op2.

void **fq_zech_sub_one**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_ctx_t* ctx)

Sets rop to the difference of op1 and 1.

void **fq_zech_neg**(*fq_zech_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to the negative of op.

void **fq_zech_mul**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_t* op2, const *fq_zech_ctx_t* ctx)

Sets rop to the product of op1 and op2, reducing the output in the given context.

void **fq_zech_mul_fmpz**(*fq_zech_t* rop, const *fq_zech_t* op, const *fmpz_t* x, const *fq_zech_ctx_t* ctx)

Sets rop to the product of op and x, reducing the output in the given context.

void **fq_zech_mul_si**(*fq_zech_t* rop, const *fq_zech_t* op, *slong* x, const *fq_zech_ctx_t* ctx)

Sets rop to the product of op and x, reducing the output in the given context.

void **fq_zech_mul_ui**(*fq_zech_t* rop, const *fq_zech_t* op, *ulong* x, const *fq_zech_ctx_t* ctx)

Sets rop to the product of op and x, reducing the output in the given context.

void **fq_zech_sqr**(*fq_zech_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to the square of op, reducing the output in the given context.

void **fq_zech_div**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_t* op2, const *fq_zech_ctx_t* ctx)

Sets rop to the quotient of op1 and op2, reducing the output in the given context.

void **_fq_zech_inv**(*nn_ptr* *rop, *nn_srcptr* *op, *slong* len, const *fq_zech_ctx_t* ctx)

Sets (rop, d) to the inverse of the non-zero element (op, len).

void **fq_zech_inv**(*fq_zech_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to the inverse of the non-zero element op.

void **fq_zech_gcdinv**(*fq_zech_t* f, *fq_zech_t* inv, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets inv to be the inverse of op modulo the modulus of ctx and sets f to one. Since the modulus for ctx is always irreducible, op is always invertible.

void **_fq_zech_pow**(*fmpz* *rop, const *fmpz* *op, *slong* len, const *fmpz_t* e, const *fmpz* *a, const *slong* *j, *slong* lena, const *fmpz_t* p)

Sets (rop, 2*d-1) to (op, len) raised to the power e, reduced modulo $f(X)$, the modulus of ctx.

Assumes that $e \geq 0$ and that len is positive and at most d.

Although we require that rop provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree d.

Does not support aliasing.

void **fq_zech_pow**(*fq_zech_t* rop, const *fq_zech_t* op, const *fmpz_t* e, const *fq_zech_ctx_t* ctx)

Sets rop the op raised to the power e.

Currently assumes that $e \geq 0$.

Note that for any input op, rop is set to 1 whenever $e = 0$.

void **fq_zech_pow_ui**(*fq_zech_t* rop, const *fq_zech_t* op, const *ulong* e, const *fq_zech_ctx_t* ctx)

Sets rop the op raised to the power e .

Currently assumes that $e \geq 0$.

Note that for any input op, rop is set to 1 whenever $e = 0$.

11.19.5 Roots

int **fq_zech_sqrt**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_ctx_t* ctx)

Sets rop to the square root of op1 if it is a square, and return 1, otherwise return 0.

void **fq_zech_pth_root**(*fq_zech_t* rop, const *fq_zech_t* op1, const *fq_zech_ctx_t* ctx)

Sets rop to a p^{th} root root of op1. Currently, this computes the root by raising op1 to p^{d-1} where d is the degree of the extension.

int **fq_zech_is_square**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Return 1 if op is a square.

11.19.6 Output

int **fq_zech_fprint_pretty**(FILE *file, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Prints a pretty representation of op to file.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

void **fq_zech_print_pretty**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Prints a pretty representation of op to stdout.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

int **fq_zech_fprint**(FILE *file, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Prints a representation of op to file.

void **fq_zech_print**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Prints a representation of op to stdout.

char ***fq_zech_get_str**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns the plain FLINT string representation of the element op.

char ***fq_zech_get_str_pretty**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns a pretty representation of the element op using the null-terminated string x as the variable name.

11.19.7 Randomisation

void **fq_zech_randtest**(*fq_zech_t* rop, *flint_rand_t* state, const *fq_zech_ctx_t* ctx)

Generates a random element of $\mathbf{F}_q$.

void **fq_zech_randtest_not_zero**(*fq_zech_t* rop, *flint_rand_t* state, const *fq_zech_ctx_t* ctx)

Generates a random non-zero element of $\mathbf{F}_q$.

void **fq_zech_randtest_dense**(*fq_zech_t* rop, *flint_rand_t* state, const *fq_zech_ctx_t* ctx)

Generates a random element of $\mathbf{F}_q$ which has an underlying polynomial with dense coefficients.

void **fq_zech_rand**(*fq_zech_t* rop, *flint_rand_t* state, const *fq_zech_ctx_t* ctx)

Generates a high quality random element of $\mathbf{F}_q$.

void **fq_zech_rand_not_zero**(*fq_zech_t* rop, *flint_rand_t* state, const *fq_zech_ctx_t* ctx)

Generates a high quality non-zero random element of $\mathbf{F}_q$.

11.19.8 Assignments and conversions

void **fq_zech_set**(*fq_zech_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to op.

void **fq_zech_set_si**(*fq_zech_t* rop, const *slong* x, const *fq_zech_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_zech_set_ui**(*fq_zech_t* rop, const *ulong* x, const *fq_zech_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_zech_set_fmpz**(*fq_zech_t* rop, const *fmpz_t* x, const *fq_zech_ctx_t* ctx)

Sets rop to x, considered as an element of $\mathbf{F}_p$.

void **fq_zech_swap**(*fq_zech_t* op1, *fq_zech_t* op2, const *fq_zech_ctx_t* ctx)

Swaps the two elements op1 and op2.

void **fq_zech_zero**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)

Sets rop to zero.

void **fq_zech_one**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)

Sets rop to one, reduced in the given context.

void **fq_zech_gen**(*fq_zech_t* rop, const *fq_zech_ctx_t* ctx)

Sets rop to a generator for the finite field. There is no guarantee this is a multiplicative generator of the finite field.

int **fq_zech_get_fmpz**(*fmpz_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

If op has a lift to the integers, return 1 and set rop to the lift in $[0, p)$. Otherwise, return 0 and leave rop undefined.

void **fq_zech_get_fq_nmod**(*fq_nmod_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to the *fq_nmod_t* element corresponding to op.

void **fq_zech_set_fq_nmod**(*fq_zech_t* rop, const *fq_nmod_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to the *fq_zech_t* element corresponding to op.

void **fq_zech_get_nmod_poly**(*nmod_poly_t* a, const *fq_zech_t* b, const *fq_zech_ctx_t* ctx)

Set a to a representative of b in ctx. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void **fq_zech_set_nmod_poly**(*fq_zech_t* a, const *nmod_poly_t* b, const *fq_zech_ctx_t* ctx)

Set a to the element in ctx with representative b. The representatives are taken in $(\mathbb{Z}/p\mathbb{Z})[x]/h(x)$ where $h(x)$ is the defining polynomial in ctx.

void **fq_zech_get_nmod_mat**(*nmod_mat_t* col, const *fq_zech_t* a, const *fq_zech_ctx_t* ctx)

Convert a to a column vector of length `degree(ctx)`.

void **fq_zech_set_nmod_mat**(*fq_zech_t* a, const *nmod_mat_t* col, const *fq_zech_ctx_t* ctx)

Convert a column vector col of length `degree(ctx)` to an element of ctx.

11.19.9 Comparison

int **fq_zech_is_zero**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns whether op is equal to zero.

int **fq_zech_is_one**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns whether op is equal to one.

int **fq_zech_equal**(const *fq_zech_t* op1, const *fq_zech_t* op2, const *fq_zech_ctx_t* ctx)

Returns whether op1 and op2 are equal.

int **fq_zech_is_invertible**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns whether op is an invertible element.

int **fq_zech_is_invertible_f**(*fq_zech_t* f, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns whether op is an invertible element. If it is not, then f is set of a factor of the modulus. Since the modulus for an *fq_zech_ctx_t* is always irreducible, then any non-zero op will be invertible.

11.19.10 Special functions

void **fq_zech_trace**(*fmpz_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to the trace of op.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$, where $d = \log_p q$.

void **fq_zech_norm**(*fmpz_t* rop, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Computes the norm of op.

For an element $a \in \mathbf{F}_q$, multiplication by a defines a $\mathbf{F}_p$ -linear map on $\mathbf{F}_q$. We define the norm of a as the determinant of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ then the trace of a is equal to $\prod_{i=0}^{d-1} \Sigma^i(a)$, where $d = \dim_{\mathbf{F}_p}(\mathbf{F}_q)$.

Algorithm selection is automatic depending on the input.

void **fq_zech_frobenius**(*fq_zech_t* rop, const *fq_zech_t* op, *slong* e, const *fq_zech_ctx_t* ctx)

Evaluates the homomorphism Σ^e at op.

Recall that $\mathbf{F}_q/\mathbf{F}_p$ is Galois with Galois group $\langle \sigma \rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma: x \mapsto x^p$.

int **fq_zech_multiplicative_order**(*fmpz_t* *ord, const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Computes the order of op as an element of the multiplicative group of ctx.

Returns 0 if op is 0, otherwise it returns 1 if op is a generator of the multiplicative group, and -1 if it is not.

Note that ctx must already correspond to a finite field defined by a primitive polynomial and so this function cannot be used to check primitivity of the generator, but can be used to check that other elements are primitive.

int **fq_zech_is_primitive**(const *fq_zech_t* op, const *fq_zech_ctx_t* ctx)

Returns whether op is primitive, i.e., whether it is a generator of the multiplicative group of ctx.

11.19.11 Bit packing

void **fq_zech_bit_pack**(*fmpz_t* f, const *fq_zech_t* op, *flint_bitcnt_t* bit_size, const *fq_zech_ctx_t* ctx)

Packs op into bitfields of size bit_size, writing the result to f.

void **fq_zech_bit_unpack**(*fq_zech_t* rop, const *fmpz_t* f, *flint_bitcnt_t* bit_size, const *fq_zech_ctx_t* ctx)

Unpacks into rop the element with coefficients packed into fields of size bit_size as represented by the integer f.

11.20 fq_zech_vec.h – vectors over finite fields (Zech logarithm representation)

11.20.1 Memory management

fq_zech_struct ***_fq_zech_vec_init**(*slong* len, const *fq_zech_ctx_t* ctx)

Returns an initialised vector of **fq_zech**'s of given length.

void **_fq_zech_vec_clear**(*fq_zech_struct* *vec, *slong* len, const *fq_zech_ctx_t* ctx)

Clears the entries of (vec, len) and frees the space allocated for vec.

11.20.2 Randomisation

void **_fq_zech_vec_randtest**(*fq_zech_struct* *f, *flint_rand_t* state, *slong* len, const *fq_zech_ctx_t* ctx)

Sets the entries of a vector of the given length to elements of the finite field.

11.20.3 Input and output

int **_fq_zech_vec_fprint**(FILE *file, const *fq_zech_struct* *vec, *slong* len, const *fq_zech_ctx_t* ctx)

Prints the vector of given length to the stream file. The format is the length followed by two spaces, then a space separated list of coefficients. If the length is zero, only 0 is printed.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fq_zech_vec_print**(const *fq_zech_struct* *vec, *slong* len, const *fq_zech_ctx_t* ctx)

Prints the vector of given length to stdout.

For further details, see **_fq_zech_vec_fprint()**.

11.20.4 Assignment and basic manipulation

void **_fq_zech_vec_set**(*fq_zech_struct* *vec1, const *fq_zech_struct* *vec2, *slong* len2, const *fq_zech_ctx_t* ctx)

Makes a copy of (vec2, len2) into vec1.

void **_fq_zech_vec_swap**(*fq_zech_struct* *vec1, *fq_zech_struct* *vec2, *slong* len2, const *fq_zech_ctx_t* ctx)

Swaps the elements in (vec1, len2) and (vec2, len2).

void **_fq_zech_vec_zero**(*fq_zech_struct* *vec, *slong* len, const *fq_zech_ctx_t* ctx)

Zeros the entries of (vec, len).

```
void _fq_zech_vec_neg(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len2, const  
                    fq_zech_ctx_t ctx)
```

Negates (vec2, len2) and places it into vec1.

11.20.5 Comparison

```
int _fq_zech_vec_equal(const fq_zech_struct *vec1, const fq_zech_struct *vec2, slong len, const  
                     fq_zech_ctx_t ctx)
```

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

```
int _fq_zech_vec_is_zero(const fq_zech_struct *vec, slong len, const fq_zech_ctx_t ctx)
```

Returns 1 if (vec, len) is zero, and 0 otherwise.

11.20.6 Addition and subtraction

```
void _fq_zech_vec_add(fq_zech_struct *res, const fq_zech_struct *vec1, const fq_zech_struct *vec2,  
                    slong len2, const fq_zech_ctx_t ctx)
```

Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

```
void _fq_zech_vec_sub(fq_zech_struct *res, const fq_zech_struct *vec1, const fq_zech_struct *vec2,  
                    slong len2, const fq_zech_ctx_t ctx)
```

Sets (res, len2) to (vec1, len2) minus (vec2, len2).

11.20.7 Scalar multiplication and division

```
void _fq_zech_vec_scalar_addmul_fq_zech(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong  
                                       len2, const fq_zech_t c, const fq_zech_ctx_t ctx)
```

Adds (vec2, len2) times *c* to (vec1, len2), where *c* is a *fq_zech_t*.

```
void _fq_zech_vec_scalar_submul_fq_zech(fq_zech_struct *vec1, const fq_zech_struct *vec2, slong  
                                       len2, const fq_zech_t c, const fq_zech_ctx_t ctx)
```

Subtracts (vec2, len2) times *c* from (vec1, len2), where *c* is a *fq_zech_t*.

11.20.8 Dot products

```
void _fq_zech_vec_dot(fq_zech_t res, const fq_zech_struct *vec1, const fq_zech_struct *vec2, slong  
                    len2, const fq_zech_ctx_t ctx)
```

Sets *res* to the dot product of (vec1, len) and (vec2, len).

11.21 *fq_zech_mat.h* – matrices over finite fields (Zech logarithm representation)

11.21.1 Types, macros and constants

```
type fq_zech_mat_struct
```

```
type fq_zech_mat_t
```

11.21.2 Memory management

void **fq_zech_mat_init**(*fq_zech_mat_t* mat, *slong* rows, *slong* cols, const *fq_zech_ctx_t* ctx)
Initialises *mat* to a rows-by-cols matrix with coefficients in $\mathbf{F}_q$ given by *ctx*. All elements are set to zero.

void **fq_zech_mat_init_set**(*fq_zech_mat_t* mat, const *fq_zech_mat_t* src, const *fq_zech_ctx_t* ctx)
Initialises *mat* and sets its dimensions and elements to those of *src*.

void **fq_zech_mat_clear**(*fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)
Clears the matrix and releases any memory it used. The matrix cannot be used again until it is initialised. This function must be called exactly once when finished using an *fq_zech_mat_t* object.

void **fq_zech_mat_set**(*fq_zech_mat_t* mat, const *fq_zech_mat_t* src, const *fq_zech_ctx_t* ctx)
Sets *mat* to a copy of *src*. It is assumed that *mat* and *src* have identical dimensions.

11.21.3 Basic properties and manipulation

fq_zech_struct ***fq_zech_mat_entry**(const *fq_zech_mat_t* mat, *slong* i, *slong* j)
Directly accesses the entry in *mat* in row *i* and column *j*, indexed from zero. No bounds checking is performed.

void **fq_zech_mat_entry_set**(*fq_zech_mat_t* mat, *slong* i, *slong* j, const *fq_zech_t* x, const *fq_zech_ctx_t* ctx)
Sets the entry in *mat* in row *i* and column *j* to *x*.

slong **fq_zech_mat_nrows**(const *fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)
Returns the number of rows in *mat*.

slong **fq_zech_mat_ncols**(const *fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)
Returns the number of columns in *mat*.

void **fq_zech_mat_swap**(*fq_zech_mat_t* mat1, *fq_zech_mat_t* mat2, const *fq_zech_ctx_t* ctx)
Swaps two matrices. The dimensions of *mat1* and *mat2* are allowed to be different.

void **fq_zech_mat_swap_entrywise**(*fq_zech_mat_t* mat1, *fq_zech_mat_t* mat2, const *fq_zech_ctx_t* ctx)
Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void **fq_zech_mat_zero**(*fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)
Sets all entries of *mat* to 0.

void **fq_zech_mat_one**(*fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)
Sets all diagonal entries of *mat* to 1 and all other entries to 0.

11.21.4 Conversions

void **fq_zech_mat_set_nmod_mat**(*fq_zech_mat_t* mat1, const *nmod_mat_t* mat2, const *fq_zech_ctx_t* ctx)
Sets the matrix *mat1* to the matrix *mat2*.

void **fq_zech_mat_set_fmpz_mod_mat**(*fq_zech_mat_t* mat1, const *fmpz_mod_mat_t* mat2, const *fq_zech_ctx_t* ctx)
Sets the matrix *mat1* to the matrix *mat2*.

11.21.5 Concatenate

void **fq_zech_mat_concat_vertical**(*fq_zech_mat_t* res, const *fq_zech_mat_t* mat1, const *fq_zech_mat_t* mat2, const *fq_zech_ctx_t* ctx)

Sets **res** to vertical concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $k \times n$, **res** : $(m + k) \times n$.

void **fq_zech_mat_concat_horizontal**(*fq_zech_mat_t* res, const *fq_zech_mat_t* mat1, const *fq_zech_mat_t* mat2, const *fq_zech_ctx_t* ctx)

Sets **res** to horizontal concatenation of (**mat1**, **mat2**) in that order. Matrix dimensions : **mat1** : $m \times n$, **mat2** : $m \times k$, **res** : $m \times (n + k)$.

11.21.6 Printing

int **fq_zech_mat_print_pretty**(const *fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)

Pretty-prints **mat** to **stdout**. A header is printed followed by the rows enclosed in brackets.

int **fq_zech_mat_fprint_pretty**(FILE *file, const *fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)

Pretty-prints **mat** to **file**. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_zech_mat_print**(const *fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)

Prints **mat** to **stdout**. A header is printed followed by the rows enclosed in brackets.

int **fq_zech_mat_fprint**(FILE *file, const *fq_zech_mat_t* mat, const *fq_zech_ctx_t* ctx)

Prints **mat** to **file**. A header is printed followed by the rows enclosed in brackets.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

11.21.7 Window

void **fq_zech_mat_window_init**(*fq_zech_mat_t* window, const *fq_zech_mat_t* mat, *slong* r1, *slong* c1, *slong* r2, *slong* c2, const *fq_zech_ctx_t* ctx)

Initializes the matrix **window** to be an $r2 - r1$ by $c2 - c1$ submatrix of **mat** whose (0,0) entry is the (r1, c1) entry of **mat**. The memory for the elements of **window** is shared with **mat**.

void **fq_zech_mat_window_clear**(*fq_zech_mat_t* window, const *fq_zech_ctx_t* ctx)

Clears the matrix **window** and releases any memory that it uses. Note that the memory to the underlying matrix that **window** points to is not freed.

11.21.8 Random matrix generation

void **fq_zech_mat_randtest**(*fq_zech_mat_t* mat, *flint_rand_t* state, const *fq_zech_ctx_t* ctx)

Sets the elements of **mat** to random elements of $\mathbf{F}_q$, given by **ctx**.

int **fq_zech_mat_randpermdiag**(*fq_zech_mat_t* mat, *flint_rand_t* state, *fq_zech_struct* *diag, *slong* n, const *fq_zech_ctx_t* ctx)

Sets **mat** to a random permutation of the diagonal matrix with n leading entries given by the vector **diag**. It is assumed that the main diagonal of **mat** has room for at least n entries.

Returns 0 or 1, depending on whether the permutation is even or odd respectively.


```
void fq_zech_mat_randrank(fq_zech_mat_t mat, flint_rand_t state, slong rank, const
fq_zech_ctx_t ctx)
```

Sets `mat` to a random sparse matrix with the given rank, having exactly as many non-zero elements as the rank, with the non-zero elements being uniformly random elements of $\mathbf{F}_q$.

The matrix can be transformed into a dense matrix with unchanged rank by subsequently calling `fq_zech_mat_randops()`.

```
void fq_zech_mat_randops(fq_zech_mat_t mat, flint_rand_t state, slong count, const
fq_zech_ctx_t ctx)
```

Randomises `mat` by performing elementary row or column operations. More precisely, at most `count` random additions or subtractions of distinct rows and columns will be performed. This leaves the rank (and for square matrices, determinant) unchanged.

```
void fq_zech_mat_randtril(fq_zech_mat_t mat, flint_rand_t state, int unit, const fq_zech_ctx_t
ctx)
```

Sets `mat` to a random lower triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

```
void fq_zech_mat_randtriu(fq_zech_mat_t mat, flint_rand_t state, int unit, const fq_zech_ctx_t
ctx)
```

Sets `mat` to a random upper triangular matrix. If `unit` is 1, it will have ones on the main diagonal, otherwise it will have random nonzero entries on the main diagonal.

11.21.9 Comparison

```
int fq_zech_mat_equal(const fq_zech_mat_t mat1, const fq_zech_mat_t mat2, const fq_zech_ctx_t
ctx)
```

Returns nonzero if `mat1` and `mat2` have the same dimensions and elements, and zero otherwise.

```
int fq_zech_mat_is_zero(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero, and otherwise returns zero.

```
int fq_zech_mat_is_one(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Returns a non-zero value if all entries `mat` are zero except the diagonal entries which must be one, otherwise returns zero.

```
int fq_zech_mat_is_empty(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Returns a non-zero value if the number of rows or the number of columns in `mat` is zero, and otherwise returns zero.

```
int fq_zech_mat_is_square(const fq_zech_mat_t mat, const fq_zech_ctx_t ctx)
```

Returns a non-zero value if the number of rows is equal to the number of columns in `mat`, and otherwise returns zero.

11.21.10 Transpose

```
void fq_zech_mat_transpose(fq_zech_mat_t B, const fq_zech_mat_t A, const fq_zech_ctx_t ctx)
```

Sets B to A^T , the transpose of A . Dimensions must be compatible. Aliasing is allowed for square matrices.

11.21.11 Addition and subtraction

void **fq_zech_mat_add**(*fq_zech_mat_t* C, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Computes $C = A + B$. Dimensions must be identical.

void **fq_zech_mat_sub**(*fq_zech_mat_t* C, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Computes $C = A - B$. Dimensions must be identical.

void **fq_zech_mat_neg**(*fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Sets $B = -A$. Dimensions must be identical.

11.21.12 Matrix multiplication

void **fq_zech_mat_mul**(*fq_zech_mat_t* C, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . This function automatically chooses between classical and KS multiplication.

void **fq_zech_mat_mul_classical**(*fq_zech_mat_t* C, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses classical matrix multiplication.

void **fq_zech_mat_mul_KS**(*fq_zech_mat_t* C, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Sets $C = AB$. Dimensions must be compatible for matrix multiplication. C is not allowed to be aliased with A or B . Uses Kronecker substitution to perform the multiplication over the integers.

void **fq_zech_mat_submul**(*fq_zech_mat_t* D, const *fq_zech_mat_t* C, const *fq_zech_mat_t* A, const *fq_zech_ctx_t* ctx)

Sets $D = C + AB$. C and D may be aliased with each other but not with A or B .

void **fq_zech_mat_mul_vec**(*fq_zech_struct* *c, const *fq_zech_mat_t* A, const *fq_zech_struct* *b, *slong* blen, const *fq_zech_ctx_t* ctx)

void **fq_zech_mat_mul_vec_ptr**(*fq_zech_struct* *const *c, const *fq_zech_mat_t* A, const *fq_zech_struct* *const *b, *slong* blen, const *fq_zech_ctx_t* ctx)

Compute a matrix-vector product of A and $(b, blen)$ and store the result in c . The vector $(b, blen)$ is either truncated or zero-extended to the number of columns of A . The number entries written to c is always equal to the number of rows of A .

void **fq_zech_mat_vec_mul**(*fq_zech_struct* *c, const *fq_zech_struct* *a, *slong* alen, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

void **fq_zech_mat_vec_mul_ptr**(*fq_zech_struct* *const *c, const *fq_zech_struct* *const *a, *slong* alen, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Compute a vector-matrix product of $(a, alen)$ and B and store the result in c . The vector $(a, alen)$ is either truncated or zero-extended to the number of rows of B . The number entries written to c is always equal to the number of columns of B .

11.21.13 LU decomposition

slong **fq_zech_mat_lu**(*slong* *P, *fq_zech_mat_t* A, int rank_check, const *fq_zech_ctx_t* ctx)

Computes a generalised LU decomposition $PLU = A$ of a given matrix A , returning the rank of A .

If A is a nonsingular square matrix, it will be overwritten with a unit diagonal lower triangular matrix L and an upper triangular matrix U (the diagonal of L will not be stored explicitly).

If A is an arbitrary matrix of rank r , U will be in row echelon form having r nonzero rows, and L will be lower triangular but truncated to r columns, having implicit ones on the r first entries of the main diagonal. All other entries will be zero.

If a nonzero value for **rank_check** is passed, the function will abandon the output matrix in an undefined state and return 0 if A is detected to be rank-deficient.

11.21.14 Reduced row echelon form

slong **fq_zech_mat_rref**(*fq_zech_mat_t* B, const *fq_zech_mat_t* A, const *fq_zech_ctx_t* ctx)

Puts B in reduced row echelon form and returns the rank of A .

The rref is computed by first obtaining an unreduced row echelon form via LU decomposition and then solving an additional triangular system.

11.21.15 Triangular solving

void **fq_zech_mat_solve_tril**(*fq_zech_mat_t* X, const *fq_zech_mat_t* L, const *fq_zech_mat_t* B, int unit, const *fq_zech_ctx_t* ctx)

Sets $X = L^{-1}B$ where L is a full rank lower triangular square matrix. If **unit** = 1, L is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

void **fq_zech_mat_solve_triu**(*fq_zech_mat_t* X, const *fq_zech_mat_t* U, const *fq_zech_mat_t* B, int unit, const *fq_zech_ctx_t* ctx)

Sets $X = U^{-1}B$ where U is a full rank upper triangular square matrix. If **unit** = 1, U is assumed to have ones on its main diagonal, and the main diagonal will not be read. X and B are allowed to be the same matrix, but no other aliasing is allowed. Automatically chooses between the classical and recursive algorithms.

11.21.16 Solving

int **fq_zech_mat_solve**(*fq_zech_mat_t* X, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$.

Returns 1 if A has full rank; otherwise returns 0 and sets the elements of X to undefined values.

The matrix A must be square.

int **fq_zech_mat_can_solve**(*fq_zech_mat_t* X, const *fq_zech_mat_t* A, const *fq_zech_mat_t* B, const *fq_zech_ctx_t* ctx)

Solves the matrix-matrix equation $AX = B$ over Fq .

Returns 1 if a solution exists; otherwise returns 0 and sets the elements of X to zero. If more than one solution exists, one of the valid solutions is given.

There are no restrictions on the shape of A and it may be singular.

11.21.17 Transforms

void **fq_zech_mat_similarity**(*fq_zech_mat_t* M, *slong* r, *fq_zech_t* d, const *fq_zech_ctx_t* ctx)

Applies a similarity transform to the $n \times n$ matrix M in-place.

If P is the $n \times n$ identity matrix the zero entries of whose row r (0-indexed) have been replaced by d , this transform is equivalent to $M = P^{-1}MP$.

Similarity transforms preserve the determinant, characteristic polynomial and minimal polynomial.

The value d is required to be reduced modulo the modulus of the entries in the matrix.

11.21.18 Characteristic polynomial

void **fq_zech_mat_charpoly**(*fq_zech_poly_t* p, const *fq_zech_mat_t* M, const *fq_zech_ctx_t* ctx)

Compute the characteristic polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.21.19 Minimal polynomial

void **fq_zech_mat_minpoly**(*fq_zech_poly_t* p, const *fq_zech_mat_t* M, const *fq_zech_ctx_t* ctx)

Compute the minimal polynomial p of the matrix M . The matrix is required to be square, otherwise an exception is raised.

11.22 fq_zech_poly.h – univariate polynomials over finite fields (Zech logarithm representation)

We represent a polynomial in $\mathbf{F}_q[X]$ as a **struct** which includes an array **coeffs** with the coefficients, as well as the length **length** and the number **alloc** of coefficients for which memory has been allocated.

As a data structure, we call this polynomial *normalised* if the top coefficient is non-zero.

Unless otherwise stated here, all functions that deal with polynomials assume that the $\mathbf{F}_q$ context of said polynomials are compatible, i.e., it assumes that the fields are generated by the same polynomial.

11.22.1 Types, macros and constants

type **fq_zech_poly_struct**

type **fq_zech_poly_t**

11.22.2 Memory management

void **fq_zech_poly_init**(*fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Initialises **poly** for use, with context **ctx**, and setting its length to zero. A corresponding call to **fq_zech_poly_clear()** must be made after finishing with the **fq_zech_poly_t** to free the memory used by the polynomial.

void **fq_zech_poly_init2**(*fq_zech_poly_t* poly, *slong* alloc, const *fq_zech_ctx_t* ctx)

Initialises **poly** with space for at least **alloc** coefficients and sets the length to zero. The allocated coefficients are all set to zero. A corresponding call to **fq_zech_poly_clear()** must be made after finishing with the **fq_zech_poly_t** to free the memory used by the polynomial.

void **fq_zech_poly_realloc**(*fq_zech_poly_t* poly, *slong* alloc, const *fq_zech_ctx_t* ctx)

Reallocates the given polynomial to have space for **alloc** coefficients. If **alloc** is zero the polynomial is cleared and then reinitialised. If the current length is greater than **alloc** the polynomial is first truncated to length **alloc**.

void **fq_zech_poly_fit_length**(*fq_zech_poly_t* poly, *slong* len, const *fq_zech_ctx_t* ctx)

If **len** is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least **len** coefficients. No data is lost when calling this function.

The function efficiently deals with the case where **fit_length** is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void **_fq_zech_poly_set_length**(*fq_zech_poly_t* poly, *slong* newlen, const *fq_zech_ctx_t* ctx)

Sets the coefficients of **poly** beyond **len** to zero and sets the length of **poly** to **len**.

void **fq_zech_poly_clear**(*fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void **_fq_zech_poly_normalise**(*fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Sets the length of **poly** so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **_fq_zech_poly_normalise2**(const *fq_zech_struct* *poly, *slong* *length, const *fq_zech_ctx_t* ctx)

Sets the length **length** of (**poly**, **length**) so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **fq_zech_poly_truncate**(*fq_zech_poly_t* poly, *slong* newlen, const *fq_zech_ctx_t* ctx)

Truncates the polynomial to length at most **n**.

void **fq_zech_poly_set_trunc**(*fq_zech_poly_t* poly1, *fq_zech_poly_t* poly2, *slong* newlen, const *fq_zech_ctx_t* ctx)

Sets **poly1** to **poly2** truncated to length **n**.

void **_fq_zech_poly_reverse**(*fq_zech_struct* *output, const *fq_zech_struct* *input, *slong* len, *slong* m, const *fq_zech_ctx_t* ctx)

Sets **output** to the reverse of **input**, which is of length **len**, but thinking of it as a polynomial of length **m**, notionally zero-padded if necessary. The length **m** must be non-negative, but there are no other restrictions. The polynomial **output** must have space for **m** coefficients.

void **fq_zech_poly_reverse**(*fq_zech_poly_t* output, const *fq_zech_poly_t* input, *slong* m, const *fq_zech_ctx_t* ctx)

Sets **output** to the reverse of **input**, thinking of it as a polynomial of length **m**, notionally zero-padded if necessary). The length **m** must be non-negative, but there are no other restrictions. The output polynomial will be set to length **m** and then normalised.

11.22.3 Polynomial parameters

slong **fq_zech_poly_degree**(const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Returns the degree of the polynomial **poly**.

slong **fq_zech_poly_length**(const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Returns the length of the polynomial **poly**.

fq_zech_struct ***fq_zech_poly_lead**(const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Returns a pointer to the leading coefficient of **poly**, or NULL if **poly** is the zero polynomial.

11.22.4 Randomisation

void **fq_zech_poly_randtest**(*fq_zech_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_zech_ctx_t* ctx)

Sets *f* to a random polynomial of length at most *len* with entries in the field described by *ctx*.

void **fq_zech_poly_randtest_not_zero**(*fq_zech_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_zech_ctx_t* ctx)

Same as **fq_zech_poly_randtest** but guarantees that the polynomial is not zero.

void **fq_zech_poly_randtest_monic**(*fq_zech_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_zech_ctx_t* ctx)

Sets *f* to a random monic polynomial of length *len* with entries in the field described by *ctx*.

void **fq_zech_poly_randtest_irreducible**(*fq_zech_poly_t* f, *flint_rand_t* state, *slong* len, const *fq_zech_ctx_t* ctx)

Sets *f* to a random monic, irreducible polynomial of length *len* with entries in the field described by *ctx*.

11.22.5 Assignment and basic manipulation

void **_fq_zech_poly_set**(*fq_zech_struct* *rop, const *fq_zech_struct* *op, *slong* len, const *fq_zech_ctx_t* ctx)

Sets (rop, len) to (op, len).

void **fq_zech_poly_set**(*fq_zech_poly_t* poly1, const *fq_zech_poly_t* poly2, const *fq_zech_ctx_t* ctx)

Sets the polynomial poly1 to the polynomial poly2.

void **fq_zech_poly_set_fq_zech**(*fq_zech_poly_t* poly, const *fq_zech_t* c, const *fq_zech_ctx_t* ctx)

Sets the polynomial poly to c.

void **fq_zech_poly_set_fmpz_mod_poly**(*fq_zech_poly_t* rop, const *fmpz_mod_poly_t* op, const *fq_zech_ctx_t* ctx)

Sets the polynomial rop to the polynomial op

void **fq_zech_poly_set_nmod_poly**(*fq_zech_poly_t* rop, const *nmod_poly_t* op, const *fq_zech_ctx_t* ctx)

Sets the polynomial rop to the polynomial op

void **fq_zech_poly_swap**(*fq_zech_poly_t* op1, *fq_zech_poly_t* op2, const *fq_zech_ctx_t* ctx)

Swaps the two polynomials op1 and op2.

void **_fq_zech_poly_zero**(*fq_zech_struct* *rop, *slong* len, const *fq_zech_ctx_t* ctx)

Sets (rop, len) to the zero polynomial.

void **fq_zech_poly_zero**(*fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Sets poly to the zero polynomial.

void **fq_zech_poly_one**(*fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Sets poly to the constant polynomial 1.

void **fq_zech_poly_gen**(*fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Sets poly to the polynomial *x*.

void **fq_zech_poly_make_monic**(*fq_zech_poly_t* rop, const *fq_zech_poly_t* op, const *fq_zech_ctx_t* ctx)

Sets rop to op, normed to have leading coefficient 1.

```
void fq_zech_poly_make_monic(fq_zech_struct *rop, const fq_zech_struct *op, slong length, const
                             fq_zech_ctx_t ctx)
```

Sets `rop` to `(op, length)`, normed to have leading coefficient 1. Assumes that `rop` has enough space for the polynomial, assumes that `op` is not zero (and thus has an invertible leading coefficient).

11.22.6 Getting and setting coefficients

```
void fq_zech_poly_get_coeff(fq_zech_t x, const fq_zech_poly_t poly, slong n, const fq_zech_ctx_t
                             ctx)
```

Sets x to the coefficient of X^n in `poly`.

```
void fq_zech_poly_set_coeff(fq_zech_poly_t poly, slong n, const fq_zech_t x, const fq_zech_ctx_t
                             ctx)
```

Sets the coefficient of X^n in `poly` to x .

```
void fq_zech_poly_set_coeff_fmpz(fq_zech_poly_t poly, slong n, const fmpz_t x, const
                                 fq_zech_ctx_t ctx)
```

Sets the coefficient of X^n in the polynomial to x , assuming $n \geq 0$.

11.22.7 Comparison

```
int fq_zech_poly_equal(const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, const
                       fq_zech_ctx_t ctx)
```

Returns nonzero if the two polynomials `poly1` and `poly2` are equal, otherwise return zero.

```
int fq_zech_poly_equal_trunc(const fq_zech_poly_t poly1, const fq_zech_poly_t poly2, slong n,
                             const fq_zech_ctx_t ctx)
```

Notionally truncate `poly1` and `poly2` to length n and return nonzero if they are equal, otherwise return zero.

```
int fq_zech_poly_is_zero(const fq_zech_poly_t poly, const fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is the zero polynomial.

```
int fq_zech_poly_is_one(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the constant polynomial 1.

```
int fq_zech_poly_is_gen(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal to the polynomial x .

```
int fq_zech_poly_is_unit(const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is a unit in the polynomial ring $\mathbf{F}_q[X]$, i.e. if it has degree 0 and is non-zero.

```
int fq_zech_poly_equal_fq_zech(const fq_zech_poly_t poly, const fq_zech_t c, const
                                fq_zech_ctx_t ctx)
```

Returns whether the polynomial `poly` is equal the (constant) $\mathbf{F}_q$ element `c`

11.22.8 Addition and subtraction

void `_fq_zech_poly_add`(*fq_zech_struct* *res, const *fq_zech_struct* *poly1, *slong* len1, const *fq_zech_struct* *poly2, *slong* len2, const *fq_zech_ctx_t* ctx)

Sets `res` to the sum of `(poly1,len1)` and `(poly2,len2)`.

void `fq_zech_poly_add`(*fq_zech_poly_t* res, const *fq_zech_poly_t* poly1, const *fq_zech_poly_t* poly2, const *fq_zech_ctx_t* ctx)

Sets `res` to the sum of `poly1` and `poly2`.

void `fq_zech_poly_add_si`(*fq_zech_poly_t* res, const *fq_zech_poly_t* poly1, *slong* c, const *fq_zech_ctx_t* ctx)

Sets `res` to the sum of `poly1` and `c`.

void `fq_zech_poly_add_series`(*fq_zech_poly_t* res, const *fq_zech_poly_t* poly1, const *fq_zech_poly_t* poly2, *slong* n, const *fq_zech_ctx_t* ctx)

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the sum.

void `_fq_zech_poly_sub`(*fq_zech_struct* *res, const *fq_zech_struct* *poly1, *slong* len1, const *fq_zech_struct* *poly2, *slong* len2, const *fq_zech_ctx_t* ctx)

Sets `res` to the difference of `(poly1,len1)` and `(poly2,len2)`.

void `fq_zech_poly_sub`(*fq_zech_poly_t* res, const *fq_zech_poly_t* poly1, const *fq_zech_poly_t* poly2, const *fq_zech_ctx_t* ctx)

Sets `res` to the difference of `poly1` and `poly2`.

void `fq_zech_poly_sub_series`(*fq_zech_poly_t* res, const *fq_zech_poly_t* poly1, const *fq_zech_poly_t* poly2, *slong* n, const *fq_zech_ctx_t* ctx)

Notionally truncate `poly1` and `poly2` to length `n` and set `res` to the difference.

void `_fq_zech_poly_neg`(*fq_zech_struct* *rop, const *fq_zech_struct* *op, *slong* len, const *fq_zech_ctx_t* ctx)

Sets `rop` to the additive inverse of `(op,len)`.

void `fq_zech_poly_neg`(*fq_zech_poly_t* res, const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Sets `res` to the additive inverse of `poly`.

11.22.9 Scalar multiplication and division

void `_fq_zech_poly_scalar_mul_fq_zech`(*fq_zech_struct* *rop, const *fq_zech_struct* *op, *slong* len, const *fq_zech_t* x, const *fq_zech_ctx_t* ctx)

Sets `(rop,len)` to the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`.

void `fq_zech_poly_scalar_mul_fq_zech`(*fq_zech_poly_t* rop, const *fq_zech_poly_t* op, const *fq_zech_t* x, const *fq_zech_ctx_t* ctx)

Sets `rop` to the product of `op` by the scalar `x`, in the context defined by `ctx`.

void `_fq_zech_poly_scalar_addmul_fq_zech`(*fq_zech_struct* *rop, const *fq_zech_struct* *op, *slong* len, const *fq_zech_t* x, const *fq_zech_ctx_t* ctx)

Adds to `(rop,len)` the product of `(op,len)` by the scalar `x`, in the context defined by `ctx`. In particular, assumes the same length for `op` and `rop`.

void `fq_zech_poly_scalar_addmul_fq_zech`(*fq_zech_poly_t* rop, const *fq_zech_poly_t* op, const *fq_zech_t* x, const *fq_zech_ctx_t* ctx)

Adds to `rop` the product of `op` by the scalar `x`, in the context defined by `ctx`.

```
void _fq_zech_poly_scalar_submul_fq_zech(fq_zech_struct *rop, const fq_zech_struct *op, slong
                                         len, const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Subtracts from (rop, len) the product of (op, len) by the scalar x , in the context defined by ctx . In particular, assumes the same length for op and rop .

```
void fq_zech_poly_scalar_submul_fq_zech(fq_zech_poly_t rop, const fq_zech_poly_t op, const
                                         fq_zech_t x, const fq_zech_ctx_t ctx)
```

Subtracts from rop the product of op by the scalar x , in the context defined by ctx .

```
void _fq_zech_poly_scalar_div_fq_zech(fq_zech_struct *rop, const fq_zech_struct *op, slong len,
                                       const fq_zech_t x, const fq_zech_ctx_t ctx)
```

Sets (rop, len) to the quotient of (op, len) by the scalar x , in the context defined by ctx . An exception is raised if x is zero.

```
void fq_zech_poly_scalar_div_fq_zech(fq_zech_poly_t rop, const fq_zech_poly_t op, const
                                       fq_zech_t x, const fq_zech_ctx_t ctx)
```

Sets rop to the quotient of op by the scalar x , in the context defined by ctx . An exception is raised if x is zero.

11.22.10 Multiplication

```
void _fq_zech_poly_mul_classical(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1,
                                 const fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 is at least len2 and neither is zero.

Permits zero padding. Does not support aliasing of rop with either $op1$ or $op2$.

```
void fq_zech_poly_mul_classical(fq_zech_poly_t rop, const fq_zech_poly_t op1, const
                                 fq_zech_poly_t op2, const fq_zech_ctx_t ctx)
```

Sets rop to the product of $op1$ and $op2$ using classical polynomial multiplication.

```
void _fq_zech_poly_mul_reorder(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const
                               fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), assuming that len1 and len2 are non-zero.

Permits zero padding. Supports aliasing.

```
void fq_zech_poly_mul_reorder(fq_zech_poly_t rop, const fq_zech_poly_t op1, const
                               fq_zech_poly_t op2, const fq_zech_ctx_t ctx)
```

Sets rop to the product of $op1$ and $op2$, reordering the two indeterminates X and Y when viewing the polynomials as elements of $\mathbf{F}_p[X, Y]$.

Suppose $\mathbf{F}_q = \mathbf{F}_p[X]/(f(X))$ and recall that elements of $\mathbf{F}_q$ are internally represented by elements of type `mpz_poly`. For small degree extensions but polynomials in $\mathbf{F}_q[Y]$ of large degree n , we change the representation to

$$\begin{aligned} g(Y) &= \sum_{i=0}^n a_i(X) Y^i \\ &= \sum_{j=0}^d \sum_{i=0}^n \text{Coeff}(a_i(X), j) Y^i. \end{aligned}$$

This allows us to use a poor algorithm (such as classical multiplication) in the X -direction and leverage the existing fast integer multiplication routines in the Y -direction where the polynomial degree n is large.

```
void _fq_zech_poly_mul_KS(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const
                        fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2).

Permits zero padding and places no assumptions on the lengths len1 and len2. Supports aliasing.

```
void fq_zech_poly_mul_KS(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t
                        op2, const fq_zech_ctx_t ctx)
```

Sets rop to the product of op1 and op2 using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_zech_poly_mul(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const
                      fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets (rop, len1 + len2 - 1) to the product of (op1, len1) and (op2, len2), choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_zech_poly_mul(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t op2,
                      const fq_zech_ctx_t ctx)
```

Sets rop to the product of op1 and op2, choosing an appropriate algorithm.

```
void _fq_zech_poly_mulow_classical(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1,
                                  const fq_zech_struct *op2, slong len2, slong n, const
                                  fq_zech_ctx_t ctx)
```

Sets (rop, n) to the first n coefficients of (op1, len1) multiplied by (op2, len2).

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Assumes neither len1 nor len2 is zero.

```
void fq_zech_poly_mulow_classical(fq_zech_poly_t rop, const fq_zech_poly_t op1, const
                                  fq_zech_poly_t op2, slong n, const fq_zech_ctx_t ctx)
```

Sets rop to the product of op1 and op2, computed using the classical or schoolbook method.

```
void _fq_zech_poly_mulow_KS(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const
                             fq_zech_struct *op2, slong len2, slong n, const fq_zech_ctx_t ctx)
```

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) and (op2, len2).

Assumes that len1 and len2 are positive, but does allow for the polynomials to be zero-padded. The polynomials may be zero, too. Assumes n is positive. Supports aliasing between rop, op1 and op2.

```
void fq_zech_poly_mulow_KS(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t
                           op2, slong n, const fq_zech_ctx_t ctx)
```

Sets rop to the product of op1 and op2.

```
void _fq_zech_poly_mulow(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const
                         fq_zech_struct *op2, slong len2, slong n, const fq_zech_ctx_t ctx)
```

Sets (rop, n) to the lowest n coefficients of the product of (op1, len1) and (op2, len2).

Assumes $0 < n \leq \text{len1} + \text{len2} - 1$. Allows for zero-padding in the inputs. Does not support aliasing between the inputs and the output.

```
void fq_zech_poly_mulow(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t
                        op2, slong n, const fq_zech_ctx_t ctx)
```

Sets rop to the lowest n coefficients of the product of op1 and op2.

```
void _fq_zech_poly_mulhigh_classical(fq_zech_struct *res, const fq_zech_struct *poly1, slong
                                     len1, const fq_zech_struct *poly2, slong len2, slong start,
                                     const fq_zech_ctx_t ctx)
```

Computes the product of (poly1, len1) and (poly2, len2) and writes the coefficients from start onwards into the high coefficients of res, the remaining coefficients being arbitrary but

reduced. Assumes that `len1 >= len2 > 0`. Aliasing of inputs and output is not permitted. Algorithm is classical multiplication.

```
void fq_zech_poly_mulhigh_classical(fq_zech_poly_t res, const fq_zech_poly_t poly1, const
                                   fq_zech_poly_t poly2, slong start, const fq_zech_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Algorithm is classical multiplication.

```
void _fq_zech_poly_mulhigh(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const
                           fq_zech_struct *poly2, slong len2, slong start, fq_zech_ctx_t ctx)
```

Computes the product of `(poly1, len1)` and `(poly2, len2)` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced. Assumes that `len1 >= len2 > 0`. Aliasing of inputs and output is not permitted.

```
void fq_zech_poly_mulhigh(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t
                           poly2, slong start, const fq_zech_ctx_t ctx)
```

Computes the product of `poly1` and `poly2` and writes the coefficients from `start` onwards into the high coefficients of `res`, the remaining coefficients being arbitrary but reduced.

```
void _fq_zech_poly_mulmod(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1, const
                           fq_zech_struct *poly2, slong len2, const fq_zech_struct *f, slong lenf,
                           const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `len1 + len2 - lenf > 0`, which is equivalent to requiring that the result will actually be reduced. Otherwise, simply use `_fq_zech_poly_mul` instead.

Aliasing of `f` and `res` is not permitted.

```
void fq_zech_poly_mulmod(fq_zech_poly_t res, const fq_zech_poly_t poly1, const fq_zech_poly_t
                           poly2, const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

```
void _fq_zech_poly_mulmod_preinv(fq_zech_struct *res, const fq_zech_struct *poly1, slong len1,
                                const fq_zech_struct *poly2, slong len2, const fq_zech_struct
                                *f, slong lenf, const fq_zech_struct *finv, slong lenfinv, const
                                fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`.

It is required that `finv` is the inverse of the reverse of `f` mod x^{lenf} .

Aliasing of `res` with any of the inputs is not permitted.

```
void fq_zech_poly_mulmod_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly1, const
                                fq_zech_poly_t poly2, const fq_zech_poly_t f, const
                                fq_zech_poly_t finv, const fq_zech_ctx_t ctx)
```

Sets `res` to the remainder of the product of `poly1` and `poly2` upon polynomial division by `f`. `finv` is the inverse of the reverse of `f`.

11.22.11 Squaring

```
void _fq_zech_poly_sqr_classical(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const
                                 fq_zech_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`, assuming that `(op, len)` is not zero and using classical polynomial multiplication.

Permits zero padding. Does not support aliasing of `rop` with either `op1` or `op2`.

```
void fq_zech_poly_sqr_classical(fq_zech_poly_t rop, const fq_zech_poly_t op, const
                               fq_zech_ctx_t ctx)
```

Sets `rop` to the square of `op` using classical polynomial multiplication.

```
void _fq_zech_poly_sqr_KS(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const
                          fq_zech_ctx_t ctx)
```

Sets `(rop, 2*len - 1)` to the square of `(op, len)`.

Permits zero padding and places no assumptions on the lengths `len1` and `len2`. Supports aliasing.

```
void fq_zech_poly_sqr_KS(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `rop` to the square `op` using Kronecker substitution, that is, by encoding each coefficient in $\mathbf{F}_q$ as an integer and reducing this problem to multiplying two polynomials over the integers.

```
void _fq_zech_poly_sqr(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const
                       fq_zech_ctx_t ctx)
```

Sets `(rop, 2 * len - 1)` to the square of `(op, len)`, choosing an appropriate algorithm.

Permits zero padding. Does not support aliasing.

```
void fq_zech_poly_sqr(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t ctx)
```

Sets `rop` to the square of `op`, choosing an appropriate algorithm.

11.22.12 Powering

```
void _fq_zech_poly_pow(fq_zech_struct *rop, const fq_zech_struct *op, slong len, ulong e, const
                      fq_zech_ctx_t ctx)
```

Sets `rop = ope`, assuming that `e`, `len > 0` and that `res` has space for `e*(len - 1) + 1` coefficients. Does not support aliasing.

```
void fq_zech_poly_pow(fq_zech_poly_t rop, const fq_zech_poly_t op, ulong e, const fq_zech_ctx_t
                      ctx)
```

Computes `rop = ope`. If `e` is zero, returns one, so that in particular `00 = 1`.

```
void _fq_zech_poly_powmod_ui_binexp(fq_zech_struct *res, const fq_zech_struct *poly, ulong e,
                                    const fq_zech_struct *f, slong lenf, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_ui_binexp(fq_zech_poly_t res, const fq_zech_poly_t poly, ulong e, const
                                    fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e >= 0`.

```
void _fq_zech_poly_powmod_ui_binexp_preinv(fq_zech_struct *res, const fq_zech_struct *poly,
                                             ulong e, const fq_zech_struct *f, slong lenf, const
                                             fq_zech_struct *finv, slong lenfinv, const
                                             fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 1`. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf - 1`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_ui_binexp_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly, ulong
                                         e, const fq_zech_poly_t f, const fq_zech_poly_t finv,
                                         const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_zech_poly_powmod_fmpz_binexp(fq_zech_struct *res, const fq_zech_struct *poly, const
                                       fmpz_t e, const fq_zech_struct *f, slong lenf, const
                                       fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $>$ 0.

We require `lenf` $>$ 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.

```
void fq_zech_poly_powmod_fmpz_binexp(fq_zech_poly_t res, const fq_zech_poly_t poly, const
                                      fmpz_t e, const fq_zech_poly_t f, const fq_zech_ctx_t
                                      ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $\geq$ 0.

```
void _fq_zech_poly_powmod_fmpz_binexp_preinv(fq_zech_struct *res, const fq_zech_struct *poly,
                                              const fmpz_t e, const fq_zech_struct *f, slong
                                              lenf, const fq_zech_struct *finv, slong lenfinv,
                                              const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $>$ 0. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf` $>$ 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.

```
void fq_zech_poly_powmod_fmpz_binexp_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly,
                                             const fmpz_t e, const fq_zech_poly_t f, const
                                             fq_zech_poly_t finv, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using binary exponentiation. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_zech_poly_powmod_fmpz_sliding_preinv(fq_zech_struct *res, const fq_zech_struct *poly,
                                              const fmpz_t e, ulong k, const fq_zech_struct *f,
                                              slong lenf, const fq_zech_struct *finv, slong
                                              lenfinv, const fq_zech_ctx_t ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e` $>$ 0. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an “optimum” size will be selected automatically base on `e`.

We require `lenf` $>$ 1. It is assumed that `poly` is already reduced modulo `f` and zero-padded as necessary to have length exactly `lenf` - 1. The output `res` must have room for `lenf` - 1 coefficients.

```
void fq_zech_poly_powmod_fmpz_sliding_preinv(fq_zech_poly_t res, const fq_zech_poly_t poly,
                                             const fmpz_t e, ulong k, const fq_zech_poly_t f,
                                             const fq_zech_poly_t finv, const fq_zech_ctx_t
                                             ctx)
```

Sets `res` to `poly` raised to the power `e` modulo `f`, using sliding-window exponentiation with window size `k`. We require `e` $\geq$ 0. We require `finv` to be the inverse of the reverse of `f`. If `k` is set to zero, then an “optimum” size will be selected automatically base on `e`.

```
void _fq_zech_poly_powmod_x_fmpz_preinv(fq_zech_struct *res, const fmpz_t e, const
                                        fq_zech_struct *f, slong lenf, const fq_zech_struct
                                        *finv, slong lenfinv, const fq_zech_ctx_t ctx)
```


Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e > 0`. We require `finv` to be the inverse of the reverse of `f`.

We require `lenf > 2`. The output `res` must have room for `lenf - 1` coefficients.

```
void fq_zech_poly_powmod_x_fmpz_preinv(fq_zech_poly_t res, const fmpz_t e, const
                                     fq_zech_poly_t f, const fq_zech_poly_t finv, const
                                     fq_zech_ctx_t ctx)
```

Sets `res` to `x` raised to the power `e` modulo `f`, using sliding window exponentiation. We require `e >= 0`. We require `finv` to be the inverse of the reverse of `f`.

```
void _fq_zech_poly_pow_trunc_binexp(fq_zech_struct *res, const fq_zech_struct *poly, ulong e,
                                   slong trunc, const fq_zech_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted. Uses the binary exponentiation method.

```
void fq_zech_poly_pow_trunc_binexp(fq_zech_poly_t res, const fq_zech_poly_t poly, ulong e, slong
                                   trunc, const fq_zech_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation. Uses the binary exponentiation method.

```
void _fq_zech_poly_pow_trunc(fq_zech_struct *res, const fq_zech_struct *poly, ulong e, slong
                             trunc, const fq_zech_ctx_t mod)
```

Sets `res` to the low `trunc` coefficients of `poly` (assumed to be zero padded if necessary to length `trunc`) to the power `e`. This is equivalent to doing a powering followed by a truncation. We require that `res` has enough space for `trunc` coefficients, that `trunc > 0` and that `e > 1`. Aliasing is not permitted.

```
void fq_zech_poly_pow_trunc(fq_zech_poly_t res, const fq_zech_poly_t poly, ulong e, slong trunc,
                             const fq_zech_ctx_t ctx)
```

Sets `res` to the low `trunc` coefficients of `poly` to the power `e`. This is equivalent to doing a powering followed by a truncation.

11.22.13 Shifting

```
void _fq_zech_poly_shift_left(fq_zech_struct *rop, const fq_zech_struct *op, slong len, slong n,
                             const fq_zech_ctx_t ctx)
```

Sets `(rop, len + n)` to `(op, len)` shifted left by `n` coefficients.

Inserts zero coefficients at the lower end. Assumes that `len` and `n` are positive, and that `rop` fits `len + n` elements. Supports aliasing between `rop` and `op`.

```
void fq_zech_poly_shift_left(fq_zech_poly_t rop, const fq_zech_poly_t op, slong n, const
                             fq_zech_ctx_t ctx)
```

Sets `rop` to `op` shifted left by `n` coeffs. Zero coefficients are inserted.

```
void _fq_zech_poly_shift_right(fq_zech_struct *rop, const fq_zech_struct *op, slong len, slong n,
                              const fq_zech_ctx_t ctx)
```

Sets `(rop, len - n)` to `(op, len)` shifted right by `n` coefficients.

Assumes that `len` and `n` are positive, that `len > n`, and that `rop` fits `len - n` elements. Supports aliasing between `rop` and `op`, although in this case the top coefficients of `op` are not set to zero.

```
void fq_zech_poly_shift_right(fq_zech_poly_t rop, const fq_zech_poly_t op, slong n, const
                              fq_zech_ctx_t ctx)
```

Sets `rop` to `op` shifted right by `n` coefficients. If `n` is equal to or greater than the current length of `op`, `rop` is set to the zero polynomial.

11.22.14 Norms

slong **fq_zech_poly_hamming_weight**(const *fq_zech_struct* *op, *slong* len, const *fq_zech_ctx_t* ctx)
Returns the number of non-zero entries in (op, len).

slong **fq_zech_poly_hamming_weight**(const *fq_zech_poly_t* op, const *fq_zech_ctx_t* ctx)
Returns the number of non-zero entries in the polynomial op.

11.22.15 Euclidean division

void **_fq_zech_poly_divrem**(*fq_zech_struct* *Q, *fq_zech_struct* *R, const *fq_zech_struct* *A, *slong* lenA, const *fq_zech_struct* *B, *slong* lenB, const *fq_zech_t* invB, const *fq_zech_ctx_t* ctx)

Computes $(Q, \text{lenA} - \text{lenB} + 1)$, (R, lenA) such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible and that *invB* is its inverse.

Assumes that $\text{len}(A), \text{len}(B) > 0$. Allows zero-padding in (A, lenA) . R and A may be aliased, but apart from this no aliasing of input and output operands is allowed.

void **fq_zech_poly_divrem**(*fq_zech_poly_t* Q, *fq_zech_poly_t* R, const *fq_zech_poly_t* A, const *fq_zech_poly_t* B, const *fq_zech_ctx_t* ctx)

Computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$.

Assumes that the leading coefficient of B is invertible. This can be taken for granted the context is for a finite field, that is, when p is prime and $f(X)$ is irreducible.

void **fq_zech_poly_divrem_f**(*fq_zech_t* f, *fq_zech_poly_t* Q, *fq_zech_poly_t* R, const *fq_zech_poly_t* A, const *fq_zech_poly_t* B, const *fq_zech_ctx_t* ctx)

Either finds a non-trivial factor f of the modulus of *ctx*, or computes Q, R such that $A = BQ + R$ and $0 \leq \text{len}(R) < \text{len}(B)$.

If the leading coefficient of B is invertible, the division with remainder operation is carried out, Q and R are computed correctly, and f is set to 1. Otherwise, f is set to a non-trivial factor of the modulus and Q and R are not touched.

Assumes that B is non-zero.

void **_fq_zech_poly_rem**(*fq_zech_struct* *R, const *fq_zech_struct* *A, *slong* lenA, const *fq_zech_struct* *B, *slong* lenB, const *fq_zech_t* invB, const *fq_zech_ctx_t* ctx)

Sets R to the remainder of the division of (A, lenA) by (B, lenB) . Assumes that the leading coefficient of (B, lenB) is invertible and that *invB* is its inverse.

void **fq_zech_poly_rem**(*fq_zech_poly_t* R, const *fq_zech_poly_t* A, const *fq_zech_poly_t* B, const *fq_zech_ctx_t* ctx)

Sets R to the remainder of the division of A by B in the context described by *ctx*.

void **_fq_zech_poly_div**(*fq_zech_struct* *Q, const *fq_zech_struct* *A, *slong* lenA, const *fq_zech_struct* *B, *slong* lenB, const *fq_zech_t* invB, const *fq_zech_ctx_t* ctx)

Notationally, computes Q, R such that $A = BQ + R$ with $0 \leq \text{len}(R) < \text{len}(B)$ but only sets $(Q, \text{lenA} - \text{lenB} + 1)$.

Allows zero-padding in A but not in B . Assumes that the leading coefficient of B is a unit.

void **fq_zech_poly_div**(*fq_zech_poly_t* Q, const *fq_zech_poly_t* A, const *fq_zech_poly_t* B, const *fq_zech_ctx_t* ctx)

Notationally finds polynomials Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q . If $\text{len}(B) = 0$ an exception is raised.

```
void _fq_zech_poly_div_newton_n_preinv(fq_zech_struct *Q, const fq_zech_struct *A, slong lenA,
                                     const fq_zech_struct *B, slong lenB, const
                                     fq_zech_struct *Binv, slong lenBinv, const
                                     fq_zech_ctx_t ctx)
```

Notionally computes polynomials Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$, but return only Q .

We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients and assume that the leading coefficient of B is a unit. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void fq_zech_poly_div_newton_n_preinv(fq_zech_poly_t Q, const fq_zech_poly_t A, const
                                     fq_zech_poly_t B, const fq_zech_poly_t Binv, const
                                     fq_zech_ctx_t ctx)
```

Notionally computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$, but returns only Q .

We assume that the leading coefficient of B is a unit and that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \times \text{the length of } B - 2$.

The algorithm used is to reverse the polynomials and divide the resulting power series, then reverse the result.

```
void _fq_zech_poly_divrem_newton_n_preinv(fq_zech_struct *Q, fq_zech_struct *R, const
                                         fq_zech_struct *A, slong lenA, const fq_zech_struct
                                         *B, slong lenB, const fq_zech_struct *Binv, slong
                                         lenBinv, const fq_zech_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R)$ less than $\text{len}B$, where A is of length $\text{len}A$ and B is of length $\text{len}B$. We require that Q have space for $\text{len}A - \text{len}B + 1$ coefficients. Furthermore, we assume that $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$. The algorithm used is to call `div_newton_preinv()` and then multiply out and compute the remainder.

```
void fq_zech_poly_divrem_newton_n_preinv(fq_zech_poly_t Q, fq_zech_poly_t R, const
                                         fq_zech_poly_t A, const fq_zech_poly_t B, const
                                         fq_zech_poly_t Binv, const fq_zech_ctx_t ctx)
```

Computes Q and R such that $A = BQ + R$ with $\text{len}(R) < \text{len}(B)$. We assume $Binv$ is the inverse of the reverse of $B \bmod x^{\text{len}(B)}$.

It is required that the length of A is less than or equal to $2 \times \text{the length of } B - 2$.

The algorithm used is to call `div_newton()` and then multiply out and compute the remainder.

```
void _fq_zech_poly_inv_series_newton(fq_zech_struct *Qinv, const fq_zech_struct *Q, slong n,
                                    const fq_zech_t cinv, const fq_zech_ctx_t ctx)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length n such that $Q * Qinv$ is 1 modulo x^n . Requires $n > 0$. This function can be viewed as inverting a power series via Newton iteration.

```
void fq_zech_poly_inv_series_newton(fq_zech_poly_t Qinv, const fq_zech_poly_t Q, slong n,
                                    const fq_zech_ctx_t ctx)
```

Given Q find $Qinv$ such that $Q * Qinv$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$. This function can be viewed as inverting a power series via Newton iteration.

```
void _fq_zech_poly_inv_series(fq_zech_struct *Qinv, const fq_zech_struct *Q, slong n, const
                              fq_zech_t cinv, const fq_zech_ctx_t ctx)
```

Given Q of length n whose constant coefficient is invertible modulo the given modulus, find a polynomial $Qinv$ of length n such that $Q * Qinv$ is 1 modulo x^n . Requires $n > 0$.

```
void fq_zech_poly_inv_series(fq_zech_poly_t Qinv, const fq_zech_poly_t Q, slong n, const
                             fq_zech_ctx_t ctx)
```

Given Q find Q_{inv} such that $Q * Q_{\text{inv}}$ is 1 modulo x^n . The constant coefficient of Q must be invertible modulo the modulus of Q . An exception is raised if this is not the case or if $n = 0$.

```
void _fq_zech_poly_div_series(fq_zech_struct *Q, const fq_zech_struct *A, slong Alen, const
                             fq_zech_struct *B, slong Blen, slong n, const fq_zech_ctx_t ctx)
```

Set (Q, n) to the quotient of the series (A, Alen) and (B, Blen) assuming $\text{Alen}, \text{Blen} \leq n$. We assume the bottom coefficient of B is invertible.

```
void fq_zech_poly_div_series(fq_zech_poly_t Q, const fq_zech_poly_t A, const fq_zech_poly_t
                             B, slong n, const fq_zech_ctx_t ctx)
```

Set Q to the quotient of the series A by B , thinking of the series as though they were of length n . We assume that the bottom coefficient of B is invertible.

11.22.16 Greatest common divisor

```
void fq_zech_poly_gcd(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t op2,
                     const fq_zech_ctx_t ctx)
```

Sets rop to the greatest common divisor of op1 and op2 , using either the Euclidean or HGCD algorithm. The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

```
slong _fq_zech_poly_gcd(fq_zech_struct *G, const fq_zech_struct *A, slong lenA, const
                       fq_zech_struct *B, slong lenB, const fq_zech_ctx_t ctx)
```

Computes the GCD of A of length lenA and B of length lenB , where $\text{lenA} \geq \text{lenB} > 0$ and sets G to it. The length of the GCD G is returned by the function. No attempt is made to make the GCD monic. It is required that G have space for lenB coefficients.

```
slong _fq_zech_poly_gcd_euclidean_f(fq_zech_t f, fq_zech_struct *G, const fq_zech_struct *A,
                                    slong lenA, const fq_zech_struct *B, slong lenB, const
                                    fq_zech_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of $(A, \text{len}(A))$ and $(B, \text{len}(B))$ and returns its length, or sets f to a non-trivial factor of the modulus of ctx and leaves the contents of the vector (G, lenB) undefined.

Assumes that $\text{len}(A) \geq \text{len}(B) > 0$ and that the vector G has space for sufficiently many coefficients.

```
void fq_zech_poly_gcd_euclidean_f(fq_zech_t f, fq_zech_poly_t G, const fq_zech_poly_t A, const
                                  fq_zech_poly_t B, const fq_zech_ctx_t ctx)
```

Either sets $f = 1$ and G to the greatest common divisor of A and B or sets f to a factor of the modulus of ctx .

```
slong _fq_zech_poly_xgcd(fq_zech_struct *G, fq_zech_struct *S, fq_zech_struct *T, const
                        fq_zech_struct *A, slong lenA, const fq_zech_struct *B, slong lenB, const
                        fq_zech_ctx_t ctx)
```

Computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fq_zech_poly_xgcd(fq_zech_poly_t G, fq_zech_poly_t S, fq_zech_poly_t T, const
fq_zech_poly_t A, const fq_zech_poly_t B, const fq_zech_ctx_t ctx)
```

Computes the GCD of A and B . The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

Polynomials S and T are computed such that $S*A + T*B = G$. The length of S will be at most `lenB` and the length of T will be at most `lenA`.

```
slong _fq_zech_poly_xgcd_euclidean_f(fq_zech_t f, fq_zech_struct *G, fq_zech_struct *S,
fq_zech_struct *T, const fq_zech_struct *A, slong lenA,
const fq_zech_struct *B, slong lenB, const fq_zech_ctx_t
ctx)
```

Either sets $f = 1$ and computes the GCD of A and B together with cofactors S and T such that $SA + TB = G$; otherwise, sets f to a non-trivial factor of the modulus of `ctx` and leaves G , S , and T undefined. Returns the length of G .

Assumes that $\text{len}(A) \geq \text{len}(B) \geq 1$ and $(\text{len}(A), \text{len}(B)) \neq (1, 1)$.

No attempt is made to make the GCD monic.

Requires that G have space for $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients. Writes $\text{len}(B) - 1$ and $\text{len}(A) - 1$ coefficients to S and T , respectively. Note that, in fact, $\text{len}(S) \leq \max(\text{len}(B) - \text{len}(G), 1)$ and $\text{len}(T) \leq \max(\text{len}(A) - \text{len}(G), 1)$.

No aliasing of input and output operands is permitted.

```
void fq_zech_poly_xgcd_euclidean_f(fq_zech_t f, fq_zech_poly_t G, fq_zech_poly_t S,
fq_zech_poly_t T, const fq_zech_poly_t A, const
fq_zech_poly_t B, const fq_zech_ctx_t ctx)
```

Either sets $f = 1$ and computes the GCD of A and B or sets f to a non-trivial factor of the modulus of `ctx`.

If the GCD is computed, polynomials S and T are computed such that $S*A + T*B = G$; otherwise, they are undefined. The length of S will be at most `lenB` and the length of T will be at most `lenA`.

The GCD of zero polynomials is defined to be zero, whereas the GCD of the zero polynomial and some other polynomial P is defined to be P . Except in the case where the GCD is zero, the GCD G is made monic.

11.22.17 Divisibility testing

```
int _fq_zech_poly_divides(fq_zech_struct *Q, const fq_zech_struct *A, slong lenA, const
fq_zech_struct *B, slong lenB, const fq_zech_t invB, const
fq_zech_ctx_t ctx)
```

Returns 1 if (B, lenB) divides (A, lenA) exactly and sets Q to the quotient, otherwise returns 0.

It is assumed that $\text{len}(A) \geq \text{len}(B) > 0$ and that Q has space for $\text{len}(A) - \text{len}(B) + 1$ coefficients.

Aliasing of Q with either of the inputs is not permitted.

This function is currently unoptimised and provided for convenience only.

```
int fq_zech_poly_divides(fq_zech_poly_t Q, const fq_zech_poly_t A, const fq_zech_poly_t B,
const fq_zech_ctx_t ctx)
```

Returns 1 if B divides A exactly and sets Q to the quotient, otherwise returns 0.

This function is currently unoptimised and provided for convenience only.

11.22.18 Derivative

```
void _fq_zech_poly_derivative(fq_zech_struct *rop, const fq_zech_struct *op, slong len, const
                             fq_zech_ctx_t ctx)
```

Sets $(rop, len - 1)$ to the derivative of (op, len) . Also handles the cases where len is 0 or 1 correctly. Supports aliasing of rop and op .

```
void fq_zech_poly_derivative(fq_zech_poly_t rop, const fq_zech_poly_t op, const fq_zech_ctx_t
                             ctx)
```

Sets rop to the derivative of op .

11.22.19 Square root

```
void _fq_zech_poly_invsqrt_series(fq_zech_struct *g, const fq_zech_struct *h, slong n,
                                 fq_zech_ctx_t mod)
```

Set the first n terms of g to the series expansion of $1/\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n . Aliasing is not permitted.

```
void fq_zech_poly_invsqrt_series(fq_zech_poly_t g, const fq_zech_poly_t h, slong n,
                                 fq_zech_ctx_t ctx)
```

Set g to the series expansion of $1/\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
void _fq_zech_poly_sqrt_series(fq_zech_struct *g, const fq_zech_struct *h, slong n,
                              fq_zech_ctx_t ctx)
```

Set the first n terms of g to the series expansion of $\sqrt{h}$. It is assumed that $n > 0$, that h has constant term 1 and that h is zero-padded as necessary to length n . Aliasing is not permitted.

```
void fq_zech_poly_sqrt_series(fq_zech_poly_t g, const fq_zech_poly_t h, slong n, fq_zech_ctx_t
                              ctx)
```

Set g to the series expansion of $\sqrt{h}$ to order $O(x^n)$. It is assumed that h has constant term 1.

```
int _fq_zech_poly_sqrt(fq_zech_struct *s, const fq_zech_struct *p, slong n, fq_zech_ctx_t mod)
```

If (p, n) is a perfect square, sets $(s, n / 2 + 1)$ to a square root of p and returns 1. Otherwise returns 0.

```
int fq_zech_poly_sqrt(fq_zech_poly_t s, const fq_zech_poly_t p, fq_zech_ctx_t mod)
```

If p is a perfect square, sets s to a square root of p and returns 1. Otherwise returns 0.

11.22.20 Evaluation

```
void _fq_zech_poly_evaluate_fq_zech(fq_zech_t rop, const fq_zech_struct *op, slong len, const
                                    fq_zech_t a, const fq_zech_ctx_t ctx)
```

Sets rop to (op, len) evaluated at a .

Supports zero padding. There are no restrictions on len , that is, len is allowed to be zero, too.

```
void fq_zech_poly_evaluate_fq_zech(fq_zech_t rop, const fq_zech_poly_t f, const fq_zech_t a,
                                   const fq_zech_ctx_t ctx)
```

Sets rop to the value of $f(a)$.

As the coefficient ring $\mathbf{F}_q$ is finite, Horner's method is sufficient.

11.22.21 Composition

```
void _fq_zech_poly_compose(fq_zech_struct *rop, const fq_zech_struct *op1, slong len1, const
fq_zech_struct *op2, slong len2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `(op1, len1)` and `(op2, len2)`.

Assumes that `rop` has space for $(len1-1)*(len2-1) + 1$ coefficients. Assumes that `op1` and `op2` are non-zero polynomials. Does not support aliasing between any of the inputs and the output.

```
void fq_zech_poly_compose(fq_zech_poly_t rop, const fq_zech_poly_t op1, const fq_zech_poly_t
op2, const fq_zech_ctx_t ctx)
```

Sets `rop` to the composition of `op1` and `op2`. To be precise about the order of composition, denoting `rop`, `op1`, and `op2` by f , g , and h , respectively, sets $f(t) = g(h(t))$.

```
void _fq_zech_poly_compose_mod_horner(fq_zech_struct *res, const fq_zech_struct *f, slong lenf,
const fq_zech_struct *g, const fq_zech_struct *h, slong
lenh, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_zech_poly_compose_mod_horner(fq_zech_poly_t res, const fq_zech_poly_t f, const
fq_zech_poly_t g, const fq_zech_poly_t h, const
fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero. The algorithm used is Horner's rule.

```
void _fq_zech_poly_compose_mod_horner_preinv(fq_zech_struct *res, const fq_zech_struct *f,
slong lenf, const fq_zech_struct *g, const
fq_zech_struct *h, slong lenh, const
fq_zech_struct *hinv, slong lenhiv, const
fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is Horner's rule.

```
void fq_zech_poly_compose_mod_horner_preinv(fq_zech_poly_t res, const fq_zech_poly_t f, const
fq_zech_poly_t g, const fq_zech_poly_t h, const
fq_zech_poly_t hinv, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is Horner's rule.

```
void _fq_zech_poly_compose_mod_brent_kung(fq_zech_struct *res, const fq_zech_struct *f, slong
lenf, const fq_zech_struct *g, const fq_zech_struct
*h, slong lenh, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_zech_poly_compose_mod_brent_kung(fq_zech_poly_t res, const fq_zech_poly_t f, const
fq_zech_poly_t g, const fq_zech_poly_t h, const
fq_zech_ctx_t ctx)
```


Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_zech_poly_compose_mod_brent_kung_preinv(fq_zech_struct *res, const fq_zech_struct *f,
                                                slong lenf, const fq_zech_struct *g, const
                                                fq_zech_struct *h, slong lenh, const
                                                fq_zech_struct *hinv, slong lenhiv, const
                                                fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_zech_poly_compose_mod_brent_kung_preinv(fq_zech_poly_t res, const fq_zech_poly_t f,
                                                const fq_zech_poly_t g, const fq_zech_poly_t
                                                h, const fq_zech_poly_t hinv, const
                                                fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The algorithm used is the Brent-Kung matrix algorithm.

```
void _fq_zech_poly_compose_mod(fq_zech_struct *res, const fq_zech_struct *f, slong lenf, const
                               fq_zech_struct *g, const fq_zech_struct *h, slong lenh, const
                               fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). The output is not allowed to be aliased with any of the inputs.

```
void fq_zech_poly_compose_mod(fq_zech_poly_t res, const fq_zech_poly_t f, const fq_zech_poly_t
                              g, const fq_zech_poly_t h, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero.

```
void _fq_zech_poly_compose_mod_preinv(fq_zech_struct *res, const fq_zech_struct *f, slong lenf,
                                       const fq_zech_struct *g, const fq_zech_struct *h, slong
                                       lenh, const fq_zech_struct *hinv, slong lenhiv, const
                                       fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that the length of g is one less than the length of h (possibly with zero padding). We also require that the length of f is less than the length of h . Furthermore, we require `hinv` to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

```
void fq_zech_poly_compose_mod_preinv(fq_zech_poly_t res, const fq_zech_poly_t f, const
                                       fq_zech_poly_t g, const fq_zech_poly_t h, const
                                       fq_zech_poly_t hinv, const fq_zech_ctx_t ctx)
```

Sets `res` to the composition $f(g)$ modulo h . We require that h is nonzero and that f has smaller degree than h . Furthermore, we require `hinv` to be the inverse of the reverse of h .

```
void _fq_zech_poly_reduce_matrix_mod_poly(fq_zech_mat_t A, const fq_zech_mat_t B, const
                                           fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Sets the i th row of A to the reduction of the i th row of B modulo f for $i = 1, \dots, \sqrt{\deg(f)}$. We require B to be at least a $\sqrt{\deg(f)} \times \deg(f)$ matrix and f to be nonzero.

```
void _fq_zech_poly_precompute_matrix(fq_zech_mat_t A, const fq_zech_struct *f, const
                                      fq_zech_struct *g, slong leng, const fq_zech_struct *ginv,
                                      slong lenginv, const fq_zech_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require `ginv` to be the inverse of the reverse of g and g to be nonzero.


```
void fq_zech_poly_precompute_matrix(fq_zech_mat_t A, const fq_zech_poly_t f, const
    fq_zech_poly_t g, const fq_zech_poly_t ginv, const
    fq_zech_ctx_t ctx)
```

Sets the i th row of A to f^i modulo g for $i = 1, \dots, \sqrt{\deg(g)}$. We require A to be a $\sqrt{\deg(g)} \times \deg(g)$ matrix. We require $ginv$ to be the inverse of the reverse of g .

```
void _fq_zech_poly_compose_mod_brent_kung_precomp_preinv(fq_zech_struct *res, const
    fq_zech_struct *f, slong lenf, const
    fq_zech_mat_t A, const
    fq_zech_struct *h, slong lenh,
    const fq_zech_struct *hinv, slong
    lenhinv, const fq_zech_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that h is nonzero. We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We also require that the length of f is less than the length of h . Furthermore, we require **hinv** to be the inverse of the reverse of h . The output is not allowed to be aliased with any of the inputs.

The algorithm used is the Brent-Kung matrix algorithm.

```
void fq_zech_poly_compose_mod_brent_kung_precomp_preinv(fq_zech_poly_t res, const
    fq_zech_poly_t f, const
    fq_zech_mat_t A, const
    fq_zech_poly_t h, const
    fq_zech_poly_t hinv, const
    fq_zech_ctx_t ctx)
```

Sets **res** to the composition $f(g)$ modulo h . We require that the i th row of A contains g^i for $i = 1, \dots, \sqrt{\deg(h)}$, i.e. A is a $\sqrt{\deg(h)} \times \deg(h)$ matrix. We require that h is nonzero and that f has smaller degree than h . Furthermore, we require **hinv** to be the inverse of the reverse of h . This version of Brent-Kung modular composition is particularly useful if one has to perform several modular composition of the form $f(g)$ modulo h for fixed g and h .

11.22.22 Output

```
int _fq_zech_poly_fprint_pretty(FILE *file, const fq_zech_struct *poly, slong len, const char *x,
    const fq_zech_ctx_t ctx)
```

Prints the pretty representation of **(poly, len)** to the stream **file**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_zech_poly_fprint_pretty(FILE *file, const fq_zech_poly_t poly, const char *x, const
    fq_zech_ctx_t ctx)
```

Prints the pretty representation of **poly** to the stream **file**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int _fq_zech_poly_print_pretty(const fq_zech_struct *poly, slong len, const char *x, const
    fq_zech_ctx_t ctx)
```

Prints the pretty representation of **(poly, len)** to **stdout**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

```
int fq_zech_poly_print_pretty(const fq_zech_poly_t poly, const char *x, const fq_zech_ctx_t ctx)
```

Prints the pretty representation of **poly** to **stdout**, using the string **x** to represent the indeterminate.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fq_zech_poly_fprint**(FILE *file, const *fq_zech_struct* *poly, *slong* len, const *fq_zech_ctx_t* ctx)

Prints the pretty representation of (poly, len) to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_zech_poly_fprint**(FILE *file, const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Prints the pretty representation of poly to the stream file.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **_fq_zech_poly_print**(const *fq_zech_struct* *poly, *slong* len, const *fq_zech_ctx_t* ctx)

Prints the pretty representation of (poly, len) to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

int **fq_zech_poly_print**(const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Prints the representation of poly to stdout.

In case of success, returns a positive value. In case of failure, returns a non-positive value.

char ***_fq_zech_poly_get_str**(const *fq_zech_struct* *poly, *slong* len, const *fq_zech_ctx_t* ctx)

Returns the plain FLINT string representation of the polynomial (poly, len).

char ***fq_zech_poly_get_str**(const *fq_zech_poly_t* poly, const *fq_zech_ctx_t* ctx)

Returns the plain FLINT string representation of the polynomial poly.

char ***_fq_zech_poly_get_str_pretty**(const *fq_zech_struct* *poly, *slong* len, const char *x, const *fq_zech_ctx_t* ctx)

Returns a pretty representation of the polynomial (poly, len) using the null-terminated string x as the variable name.

char ***fq_zech_poly_get_str_pretty**(const *fq_zech_poly_t* poly, const char *x, const *fq_zech_ctx_t* ctx)

Returns a pretty representation of the polynomial poly using the null-terminated string x as the variable name

11.22.23 Inflation and deflation

void **fq_zech_poly_inflate**(*fq_zech_poly_t* result, const *fq_zech_poly_t* input, *ulong* inflation, const *fq_zech_ctx_t* ctx)

Sets result to the inflated polynomial $p(x^n)$ where p is given by input and n is given by inflation.

void **fq_zech_poly_deflate**(*fq_zech_poly_t* result, const *fq_zech_poly_t* input, *ulong* deflation, const *fq_zech_ctx_t* ctx)

Sets result to the deflated polynomial $p(x^{1/n})$ where p is given by input and n is given by deflation. Requires $n > 0$.

ulong **fq_zech_poly_deflation**(const *fq_zech_poly_t* input, const *fq_zech_ctx_t* ctx)

Returns the largest integer by which input can be deflated. As special cases, returns 0 if input is the zero polynomial and 1 if input is a constant polynomial.

11.23 `fq_zech_poly_factor.h` – factorisation of univariate polynomials over finite fields (Zech logarithm representation)

11.23.1 Types, macros and constants

type `fq_zech_poly_factor_struct`

type `fq_zech_poly_factor_t`

11.23.2 Memory Management

void `fq_zech_poly_factor_init`(*fq_zech_poly_factor_t* fac, const *fq_zech_ctx_t* ctx)

Initialises `fac` for use. An `fq_zech_poly_factor_t` represents a polynomial in factorised form as a product of polynomials with associated exponents.

void `fq_zech_poly_factor_clear`(*fq_zech_poly_factor_t* fac, const *fq_zech_ctx_t* ctx)

Frees all memory associated with `fac`.

void `fq_zech_poly_factor_realloc`(*fq_zech_poly_factor_t* fac, *slong* alloc, const *fq_zech_ctx_t* ctx)

Reallocates the factor structure to provide space for precisely `alloc` factors.

void `fq_zech_poly_factor_fit_length`(*fq_zech_poly_factor_t* fac, *slong* len, const *fq_zech_ctx_t* ctx)

Ensures that the factor structure has space for at least `len` factors. This function takes care of the case of repeated calls by always at least doubling the number of factors the structure can hold.

11.23.3 Basic Operations

void `fq_zech_poly_factor_set`(*fq_zech_poly_factor_t* res, const *fq_zech_poly_factor_t* fac, const *fq_zech_ctx_t* ctx)

Sets `res` to the same factorisation as `fac`.

void `fq_zech_poly_factor_print_pretty`(const *fq_zech_poly_factor_t* fac, const char *var, const *fq_zech_ctx_t* ctx)

Pretty-prints the entries of `fac` to standard output.

void `fq_zech_poly_factor_print`(const *fq_zech_poly_factor_t* fac, const *fq_zech_ctx_t* ctx)

Prints the entries of `fac` to standard output.

void `fq_zech_poly_factor_insert`(*fq_zech_poly_factor_t* fac, const *fq_zech_poly_t* poly, *slong* exp, const *fq_zech_ctx_t* ctx)

Inserts the factor `poly` with multiplicity `exp` into the factorisation `fac`.

If `fac` already contains `poly`, then `exp` simply gets added to the exponent of the existing entry.

void `fq_zech_poly_factor_concat`(*fq_zech_poly_factor_t* res, const *fq_zech_poly_factor_t* fac, const *fq_zech_ctx_t* ctx)

Concatenates two factorisations.

This is equivalent to calling `fq_zech_poly_factor_insert()` repeatedly with the individual factors of `fac`.

Does not support aliasing between `res` and `fac`.

void `fq_zech_poly_factor_pow`(*fq_zech_poly_factor_t* fac, *slong* exp, const *fq_zech_ctx_t* ctx)

Raises `fac` to the power `exp`.

ulong **fq_zech_poly_remove**(*fq_zech_poly_t* f, const *fq_zech_poly_t* p, const *fq_zech_ctx_t* ctx)
Removes the highest possible power of *p* from *f* and returns the exponent.

11.23.4 Irreducibility Testing

int **fq_zech_poly_is_irreducible**(const *fq_zech_poly_t* f, const *fq_zech_ctx_t* ctx)
Returns 1 if the polynomial *f* is irreducible, otherwise returns 0.

int **fq_zech_poly_is_irreducible_ddf**(const *fq_zech_poly_t* f, const *fq_zech_ctx_t* ctx)
Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses fast distinct-degree factorisation.

int **fq_zech_poly_is_irreducible_ben_or**(const *fq_zech_poly_t* f, const *fq_zech_ctx_t* ctx)
Returns 1 if the polynomial *f* is irreducible, otherwise returns 0. Uses Ben-Or's irreducibility test.

int **_fq_zech_poly_is_squarefree**(const *fq_zech_struct* *f, *slong* len, const *fq_zech_ctx_t* ctx)
Returns 1 if (*f*, *len*) is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree. There are no restrictions on the length.

int **fq_zech_poly_is_squarefree**(const *fq_zech_poly_t* f, const *fq_zech_ctx_t* ctx)
Returns 1 if *f* is squarefree, and 0 otherwise. As a special case, the zero polynomial is not considered squarefree.

11.23.5 Factorisation

int **fq_zech_poly_factor_equal_deg_prob**(*fq_zech_poly_t* factor, *flint_rand_t* state, const *fq_zech_poly_t* pol, *slong* d, const *fq_zech_ctx_t* ctx)
Probabilistic equal degree factorisation of *pol* into irreducible factors of degree *d*. If it passes, a factor is placed in *factor* and 1 is returned, otherwise 0 is returned and the value of *factor* is undetermined.
Requires that *pol* be monic, non-constant and squarefree.

void **fq_zech_poly_factor_equal_deg**(*fq_zech_poly_factor_t* factors, const *fq_zech_poly_t* pol, *slong* d, const *fq_zech_ctx_t* ctx)
Assuming *pol* is a product of irreducible factors all of degree *d*, finds all those factors and places them in *factors*. Requires that *pol* be monic, non-constant and squarefree.

void **fq_zech_poly_factor_split_single**(*fq_zech_poly_t* linfactor, const *fq_zech_poly_t* input, const *fq_zech_ctx_t* ctx)
Assuming *input* is a product of factors all of degree 1, finds a single linear factor of *input* and places it in *linfactor*. Requires that *input* be monic and non-constant.

void **fq_zech_poly_factor_distinct_deg**(*fq_zech_poly_factor_t* res, const *fq_zech_poly_t* poly, *slong* *const *degs, const *fq_zech_ctx_t* ctx)
Factorises a monic non-constant squarefree polynomial *poly* of degree *n* into factors $f[d]$ such that for $1 \leq d \leq n$ $f[d]$ is the product of the monic irreducible factors of *poly* of degree *d*. Factors are stored in *res*, associated powers of irreducible polynomials are stored in *degs* in the same order as factors.
Requires that *degs* have enough space for irreducible polynomials' powers (maximum space required is $n * \text{sizeof}(\text{slong})$).

void **fq_zech_poly_factor_squarefree**(*fq_zech_poly_factor_t* res, const *fq_zech_poly_t* f, const *fq_zech_ctx_t* ctx)
Sets *res* to a squarefree factorization of *f*.

```
void fq_zech_poly_factor(fq_zech_poly_factor_t res, fq_zech_t lead, const fq_zech_poly_t f, const
fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors choosing the best algorithm for given modulo and degree. The output `lead` is set to the leading coefficient of f upon return. Choice of algorithm is based on heuristic measurements.

```
void fq_zech_poly_factor_cantor_zassenhaus(fq_zech_poly_factor_t res, const fq_zech_poly_t f,
const fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors using the Cantor-Zassenhaus algorithm.

```
void fq_zech_poly_factor_kaltofen_shoup(fq_zech_poly_factor_t res, const fq_zech_poly_t poly,
const fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors using the fast version of Cantor-Zassenhaus algorithm proposed by Kaltofen and Shoup (1998). More precisely this algorithm uses a “baby step/giant step” strategy for the distinct-degree factorization step.

```
void fq_zech_poly_factor_berlekamp(fq_zech_poly_factor_t factors, const fq_zech_poly_t f, const
fq_zech_ctx_t ctx)
```

Factorises a non-constant polynomial f into monic irreducible factors using the Berlekamp algorithm.

```
void fq_zech_poly_factor_with_berlekamp(fq_zech_poly_factor_t res, fq_zech_t leading_coeff,
const fq_zech_poly_t f, const fq_zech_ctx_t ctx)
```

Factorises a general polynomial f into monic irreducible factors and sets `leading_coeff` to the leading coefficient of f , or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Berlekamp on all the individual square-free factors.

```
void fq_zech_poly_factor_with_cantor_zassenhaus(fq_zech_poly_factor_t res, fq_zech_t
leading_coeff, const fq_zech_poly_t f, const
fq_zech_ctx_t ctx)
```

Factorises a general polynomial f into monic irreducible factors and sets `leading_coeff` to the leading coefficient of f , or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Cantor-Zassenhaus on all the individual square-free factors.

```
void fq_zech_poly_factor_with_kaltofen_shoup(fq_zech_poly_factor_t res, fq_zech_t
leading_coeff, const fq_zech_poly_t f, const
fq_zech_ctx_t ctx)
```

Factorises a general polynomial f into monic irreducible factors and sets `leading_coeff` to the leading coefficient of f , or 0 if f is the zero polynomial.

This function first checks for small special cases, deflates f if it is of the form $p(x^m)$ for some $m > 1$, then performs a square-free factorisation, and finally runs Kaltofen-Shoup on all the individual square-free factors.

```
void fq_zech_poly_iterated_frobenius_preinv(fq_zech_poly_t *rop, slong n, const
fq_zech_poly_t v, const fq_zech_poly_t vinv,
const fq_zech_ctx_t ctx)
```

Sets `rop[i]` to be $x^{q^i} \bmod v$ for $0 \leq i < n$.

It is required that `vinv` is the inverse of the reverse of $v \bmod x^{\text{len}v}$.

11.23.6 Root Finding

```
void fq_zech_poly_roots(fq_zech_poly_factor_t r, const fq_zech_poly_t f, int with_multiplicity,
                      const fq_zech_ctx_t ctx)
```

Fill r with factors of the form $x - r_i$ where the r_i are the distinct roots of a nonzero f in F_q . If *with_multiplicity* is zero, the exponent e_i of the factor $x - r_i$ is 1. Otherwise, it is the largest e_i such that $(x - r_i)^{e_i}$ divides f . This function throws if f is zero, but is otherwise always successful.

11.24 fq_zech_embed.h – Computing isomorphisms and embeddings of finite fields

```
void fq_zech_embed_gens(fq_zech_t gen_sub, fq_zech_t gen_sup, nmod_poly_t minpoly, const
                      fq_zech_ctx_t sub_ctx, const fq_zech_ctx_t sup_ctx)
```

Given two contexts `sub_ctx` and `sup_ctx`, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute:

- an element `gen_sub` in `sub_ctx` such that `gen_sub` generates the finite field defined by `sub_ctx`,
- its minimal polynomial `minpoly`,
- a root `gen_sup` of `minpoly` inside the field defined by `sup_ctx`.

These data uniquely define an embedding of `sub_ctx` into `sup_ctx`.

```
void _fq_zech_embed_gens_naive(fq_zech_t gen_sub, fq_zech_t gen_sup, nmod_poly_t minpoly,
                             const fq_zech_ctx_t sub_ctx, const fq_zech_ctx_t sup_ctx)
```

Given two contexts `sub_ctx` and `sup_ctx`, such that `degree(sub_ctx)` divides `degree(sup_ctx)`, compute an embedding of `sub_ctx` into `sup_ctx` defined as follows:

- `gen_sub` is the canonical generator of `sub_ctx` (i.e., the class of X),
- `minpoly` is the defining polynomial of `sub_ctx`,
- `gen_sup` is a root of `minpoly` inside the field defined by `sup_ctx`.

```
void fq_zech_embed_matrices(nmod_mat_t embed, nmod_mat_t project, const fq_zech_t gen_sub,
                          const fq_zech_ctx_t sub_ctx, const fq_zech_t gen_sup, const
                          fq_zech_ctx_t sup_ctx, const nmod_poly_t gen_minpoly)
```

Given:

- two contexts `sub_ctx` and `sup_ctx`, of respective degrees m and n , such that m divides n ;
- a generator `gen_sub` of `sub_ctx`, its minimal polynomial `gen_minpoly`, and a root `gen_sup` of `gen_minpoly` in `sup_ctx`, as returned by `fq_zech_embed_gens`;

Compute:

- the $n \times m$ matrix `embed` mapping `gen_sub` to `gen_sup`, and all their powers accordingly;
- an $m \times n$ matrix `project` such that `project` $\times$ `embed` is the $m \times m$ identity matrix.

```
void fq_zech_embed_trace_matrix(nmod_mat_t res, const nmod_mat_t basis, const fq_zech_ctx_t
                              sub_ctx, const fq_zech_ctx_t sup_ctx)
```

Given:

- two contexts `sub_ctx` and `sup_ctx`, of degrees m and n , such that m divides n ;
- an $n \times m$ matrix `basis` that maps `sub_ctx` to an isomorphic subfield in `sup_ctx`;

Compute the $m \times n$ matrix of the trace from `sup_ctx` to `sub_ctx`.

This matrix is computed as

`embed_dual_to_mono_matrix(_, sub_ctx) × basist × embed_mono_to_dual_matrix(_, sup_ctx)}`).

Note: if $m = n$, `basis` represents a Frobenius, and the result is its inverse matrix.

void `fq_zech_embed_composition_matrix`(*nmod_mat_t* matrix, const *fq_zech_t* gen, const *fq_zech_ctx_t* ctx)

Compute the *composition matrix* of `gen`.

For an element $a \in \mathbf{F}_{p^n}$, its composition matrix is the matrix whose columns are $a^0, a^1, \dots, a^{n-1}$.

void `fq_zech_embed_composition_matrix_sub`(*nmod_mat_t* matrix, const *fq_zech_t* gen, const *fq_zech_ctx_t* ctx, *slong* trunc)

Compute the *composition matrix* of `gen`, truncated to `trunc` columns.

void `fq_zech_embed_mul_matrix`(*nmod_mat_t* matrix, const *fq_zech_t* gen, const *fq_zech_ctx_t* ctx)

Compute the *multiplication matrix* of `gen`.

For an element a in $\mathbf{F}_{p^n} = \mathbf{F}_p[x]$, its multiplication matrix is the matrix whose columns are $a, ax, \dots, ax^{n-1}$.

void `fq_zech_embed_mono_to_dual_matrix`(*nmod_mat_t* res, const *fq_zech_ctx_t* ctx)

Compute the change of basis matrix from the monomial basis of `ctx` to its dual basis.

void `fq_zech_embed_dual_to_mono_matrix`(*nmod_mat_t* res, const *fq_zech_ctx_t* ctx)

Compute the change of basis matrix from the dual basis of `ctx` to its monomial basis.

void `fq_zech_modulus_pow_series_inv`(*nmod_poly_t* res, const *fq_zech_ctx_t* ctx, *slong* trunc)

Compute the power series inverse of the reverse of the modulus of `ctx` up to $O(x^{\text{trunc}})$.

void `fq_zech_modulus_derivative_inv`(*fq_zech_t* m_prime, *fq_zech_t* m_prime_inv, const *fq_zech_ctx_t* ctx)

Compute the derivative `m_prime` of the modulus of `ctx` as an element of `ctx`, and its inverse `m_prime_inv`.

P-ADIC NUMBERS

12.1 `padic.h` – p-adic numbers

12.1.1 Introduction

The `padic_t` data type represents elements of $\mathbf{Q}_p$ to precision N , stored in the form $x = p^v u$ with $u, v \in \mathbf{Z}$. Arithmetic operations can be carried out with respect to a context containing the prime number p and various pieces of pre-computed data.

Independent of the context, we consider a p -adic number $x = up^v$ to be in canonical form whenever either $p \nmid u$ or $u = v = 0$, and we say it is reduced if, in addition, for non-zero u , $u \in (0, p^{N-v})$.

We briefly describe the interface:

The functions in this module expect arguments of type `padic_t`, and each variable carries its own precision. The functions have an interface that is similar to the MPFR functions. In particular, they have the same semantics, specified as follows: Compute the requested operation exactly and then reduce the result to the precision of the output variable.

12.1.2 Data structures

A p -adic number of type `padic_t` comprises a unit u , a valuation v , and a precision N . We provide the following macros to access these fields, so that code can be developed somewhat independently from the underlying data layout.

mpz `*padic_unit(const padic_t op)`

Returns the unit part of the p -adic number as a FLINT integer, which can be used as an operand for the `mpz` functions.

slong `padic_val(const padic_t op)`

Returns the valuation part of the p -adic number.

Note that this function is implemented as a macro and that the expression `padic_val(op)` can be used as both an *lvalue* and an *rvalue*.

slong `padic_get_val(const padic_t op)`

Returns the valuation part of the p -adic number.

slong `padic_prec(const padic_t op)`

Returns the precision of the p -adic number.

Note that this function is implemented as a macro and that the expression `padic_prec(op)` can be used as both an *lvalue* and an *rvalue*.

slong `padic_get_prec(const padic_t op)`

Returns the precision of the p -adic number.

12.1.3 Context

A context object for p -adic arithmetic contains data pertinent to p -adic computations, but which we choose not to store with each element individually. Currently, this includes the prime number p , its `double` inverse in case of word-sized primes, precomputed powers of p in the range given by `min` and `max`, and the printing mode.

```
void padic_ctx_init(padic_ctx_t ctx, const fmpz_t p, slong min, slong max, enum
                    padic_print_mode mode)
```

Initialises the context `ctx` with the given data.

Assumes that p is a prime. This is not verified but the subsequent behaviour is undefined if p is a composite number.

Assumes that `min` and `max` are non-negative and that `min` is at most `max`, raising an `abort` signal otherwise.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`. Using the example $x = 7^{-1}12$ in $\mathbb{Q}_7$, these behave as follows:

In `PADIC_TERSE` mode, a p -adic number is printed in the same way as a rational number, e.g. $12/7$.

In `PADIC_SERIES` mode, a p -adic number is printed digit by digit, e.g. $5*7^{-1} + 1$.

In `PADIC_VAL_UNIT` mode, a p -adic number is printed showing the valuation and unit parts separately, e.g. $12*7^{-1}$.

```
void padic_ctx_clear(padic_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
int _padic_ctx_pow_ui(fmpz_t rop, ulong e, const padic_ctx_t ctx)
```

Sets `rop` to p^e as efficiently as possible, where `rop` is expected to be an uninitialised `fmpz_t`.

If the return value is non-zero, it is the responsibility of the caller to clear the returned integer.

12.1.4 Memory management

```
void padic_init(padic_t rop)
```

Initialises the p -adic number with the precision set to `PADIC_DEFAULT_PREC`, which is defined as 20.

```
void padic_init2(padic_t rop, slong N)
```

Initialises the p -adic number `rop` with precision N .

```
void padic_clear(padic_t rop)
```

Clears all memory used by the p -adic number `rop`.

```
void _padic_canonicalise(padic_t rop, const padic_ctx_t ctx)
```

Brings the p -adic number `rop` into canonical form.

That is to say, ensures that either $u = v = 0$ or $p \nmid u$. There is no reduction modulo a power of p .

```
void _padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Given a p -adic number `rop` in canonical form, reduces it modulo p^N .

```
void padic_reduce(padic_t rop, const padic_ctx_t ctx)
```

Ensures that the p -adic number `rop` is reduced.

12.1.5 Randomisation

void **padic_randtest**(padic_t rop, *flint_rand_t* state, const padic_ctx_t ctx)

Sets **rop** to a random p -adic number modulo p^N with valuation in the range $[-\lceil N/10 \rceil, N)$, $[N - \lceil N/10 \rceil, N)$, or $[-10, 0)$ as N is positive, negative or zero, whenever **rop** is non-zero.

void **padic_randtest_not_zero**(padic_t rop, *flint_rand_t* state, const padic_ctx_t ctx)

Sets **rop** to a random non-zero p -adic number modulo p^N , where the range of the valuation is as for the function *padic_randtest()*.

void **padic_randtest_int**(padic_t rop, *flint_rand_t* state, const padic_ctx_t ctx)

Sets **rop** to a random p -adic integer modulo p^N .

Note that whenever $N \leq 0$, **rop** is set to zero.

12.1.6 Assignments and conversions

All assignment functions set the value of **rop** from **op**, reduced to the precision of **rop**.

void **padic_set**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Sets **rop** to the p -adic number **op**.

void **padic_set_si**(padic_t rop, *slong* op, const padic_ctx_t ctx)

Sets the p -adic number **rop** to the *slong* integer **op**.

void **padic_set_ui**(padic_t rop, *ulong* op, const padic_ctx_t ctx)

Sets the p -adic number **rop** to the *ulong* integer **op**.

void **padic_set_fmpz**(padic_t rop, const *fmpz_t* op, const padic_ctx_t ctx)

Sets the p -adic number **rop** to the integer **op**.

void **padic_set_fmpq**(padic_t rop, const *fmpq_t* op, const padic_ctx_t ctx)

Sets **rop** to the rational **op**.

void **padic_set_mpz**(padic_t rop, const mpz_t op, const padic_ctx_t ctx)

Sets the p -adic number **rop** to the GMP integer **op**.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **padic_set_mpq**(padic_t rop, const mpq_t op, const padic_ctx_t ctx)

Sets **rop** to the GMP rational **op**.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **padic_get_fmpz**(*fmpz_t* rop, const padic_t op, const padic_ctx_t ctx)

Sets the integer **rop** to the exact p -adic integer **op**.

If **op** is not a p -adic integer, raises an **abort** signal.

void **padic_get_fmpq**(*fmpq_t* rop, const padic_t op, const padic_ctx_t ctx)

Sets the rational **rop** to the p -adic number **op**.

void **padic_get_mpz**(mpz_t rop, const padic_t op, const padic_ctx_t ctx)

Sets the GMP integer **rop** to the p -adic integer **op**.

If **op** is not a p -adic integer, raises an **abort** signal.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **padic_get_mpq**(mpq_t rop, const padic_t op, const padic_ctx_t ctx)

Sets the GMP rational **rop** to the value of **op**.

Note: Requires that *gmp.h* has been included before any FLINT header is included.

void **padic_swap**(padic_t op1, padic_t op2)

Swaps the two p -adic numbers **op1** and **op2**.

Note that this includes swapping the precisions. In particular, this operation is not equivalent to swapping **op1** and **op2** using **padic_set()** and an auxiliary variable whenever the precisions of the two elements are different.

void **padic_zero**(padic_t rop)

Sets the p -adic number **rop** to zero.

void **padic_one**(padic_t rop)

Sets the p -adic number **rop** to one, reduced modulo the precision of **rop**.

12.1.7 Comparison

int **padic_is_zero**(const padic_t op)

Returns whether **op** is equal to zero.

int **padic_is_one**(const padic_t op)

Returns whether **op** is equal to one, that is, whether $u = 1$ and $v = 0$.

int **padic_equal**(const padic_t op1, const padic_t op2)

Returns whether **op1** and **op2** are equal, that is, whether $u_1 = u_2$ and $v_1 = v_2$.

12.1.8 Arithmetic operations

slong *_**padic_lifts_exps**(*slong* *n, *slong* N)

Given a positive integer N define the sequence $a_0 = N, a_1 = \lceil a_0/2 \rceil, \dots, a_{n-1} = \lceil a_{n-2}/2 \rceil = 1$. Then $n = \lceil \log_2 N \rceil + 1$.

This function sets n and allocates and returns the array a .

void **_padic_lifts_pows**(*fmpz_t* *pow, const *slong* *a, *slong* n, const *fmpz_t* p)

Given an array a as computed above, this function computes the corresponding powers of p , that is, $\text{pow}[i]$ is equal to p^{a_i} .

void **padic_add**(padic_t rop, const padic_t op1, const padic_t op2, const padic_ctx_t ctx)

Sets **rop** to the sum of **op1** and **op2**.

void **padic_sub**(padic_t rop, const padic_t op1, const padic_t op2, const padic_ctx_t ctx)

Sets **rop** to the difference of **op1** and **op2**.

void **padic_neg**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Sets **rop** to the additive inverse of **op**.

void **padic_mul**(padic_t rop, const padic_t op1, const padic_t op2, const padic_ctx_t ctx)

Sets **rop** to the product of **op1** and **op2**.

void **padic_shift**(padic_t rop, const padic_t op, *slong* v, const padic_ctx_t ctx)

Sets **rop** to the product of **op** and p^v .

void **padic_div**(padic_t rop, const padic_t op1, const padic_t op2, const padic_ctx_t ctx)

Sets **rop** to the quotient of **op1** and **op2**.

void **_padic_inv_precompute**(padic_inv_t S, const *fmpz_t* p, *slong* N)

Pre-computes some data and allocates temporary space for p -adic inversion using Hensel lifting.

void **_padic_inv_clear**(padic_inv_t S)

Frees the memory used by S .

void **_padic_inv_precomp**(*fmpz_t* rop, const *fmpz_t* op, const *padic_inv_t* S)

Sets **rop** to the inverse of **op** modulo p^N , assuming that **op** is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

Uses some data S precomputed by calling the function **_padic_inv_precompute()**. Note that this object is not declared **const** and in fact it carries a field providing temporary work space. This allows repeated calls of this function to avoid repeated memory allocations, as used e.g. by the function **padic_log()**.

void **_padic_inv**(*fmpz_t* rop, const *fmpz_t* op, const *fmpz_t* p, *slong* N)

Sets **rop** to the inverse of **op** modulo p^N , assuming that **op** is a unit and $N \geq 1$.

In the current implementation, allows aliasing, but this might change in future versions.

void **padic_inv**(*padic_t* rop, const *padic_t* op, const *padic_ctx_t* ctx)

Computes the inverse of **op** modulo p^N .

Suppose that **op** is given as $x = up^v$. Raises an **abort** signal if $v < -N$. Otherwise, computes the inverse of u modulo p^{N+v} .

This function employs Hensel lifting of an inverse modulo p .

int **padic_sqrt**(*padic_t* rop, const *padic_t* op, const *padic_ctx_t* ctx)

Returns whether **op** is a p -adic square. If this is the case, sets **rop** to one of the square roots; otherwise, the value of **rop** is undefined.

We have the following theorem:

Let $u \in \mathbf{Z}^\times$. Then u is a square if and only if $u \bmod p$ is a square in $\mathbf{Z}/p\mathbf{Z}$, for $p > 2$, or if $u \bmod 8$ is a square in $\mathbf{Z}/8\mathbf{Z}$, for $p = 2$.

void **padic_pow_si**(*padic_t* rop, const *padic_t* op, *slong* e, const *padic_ctx_t* ctx)

Sets **rop** to **op** raised to the power e , which is defined as one whenever $e = 0$.

Assumes that some computations involving e and the valuation of **op** do not overflow in the **slong** range.

Note that if the input $x = p^v u$ is defined modulo p^N then $x^e = p^{ev} u^e$ is defined modulo $p^{N+(e-1)v}$, which is a precision loss in case $v < 0$.

12.1.9 Exponential

slong **_padic_exp_bound**(*slong* v, *slong* N, const *fmpz_t* p)

Returns an integer i such that for all $j \geq i$ we have $\text{ord}_p(x^j/j!) \geq N$, where $\text{ord}_p(x) = v$.

When p is a word-sized prime, returns $\left\lceil \frac{(p-1)N-1}{(p-1)v-1} \right\rceil$. Otherwise, returns $\lceil N/v \rceil$.

Assumes that $v < N$. Moreover, v has to be at least 2 or 1, depending on whether p is 2 or odd.

void **_padic_exp_rectangular**(*fmpz_t* rop, const *fmpz_t* u, *slong* v, const *fmpz_t* p, *slong* N)

void **_padic_exp_balanced**(*fmpz_t* rop, const *fmpz_t* u, *slong* v, const *fmpz_t* p, *slong* N)

void **_padic_exp**(*fmpz_t* rop, const *fmpz_t* u, *slong* v, const *fmpz_t* p, *slong* N)

Sets **rop** to the p -exponential function evaluated at $x = p^v u$, reduced modulo p^N .

Assumes that $x \neq 0$, that $\text{ord}_p(x) < N$ and that $\exp(x)$ converges, that is, that $\text{ord}_p(x)$ is at least 2 or 1 depending on whether the prime p is 2 or odd.

Supports aliasing between **rop** and u .

int **padic_exp**(padic_t y, const padic_t x, const padic_ctx_t ctx)

Returns whether the p -adic exponential function converges at the p -adic number x , and if so sets y to its value.

The p -adic exponential function is defined by the usual series

$$\exp_p(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

but this only converges only when $\text{ord}_p(x) > 1/(p-1)$. For elements $x \in \mathbf{Q}_p$, this means that $\text{ord}_p(x) \geq 1$ when $p \geq 3$ and $\text{ord}_2(x) \geq 2$ when $p = 2$.

int **padic_exp_rectangular**(padic_t y, const padic_t x, const padic_ctx_t ctx)

Returns whether the p -adic exponential function converges at the p -adic number x , and if so sets y to its value.

Uses a rectangular splitting algorithm to evaluate the series expression of $\exp(x) \bmod p^N$.

int **padic_exp_balanced**(padic_t y, const padic_t x, const padic_ctx_t ctx)

Returns whether the p -adic exponential function converges at the p -adic number x , and if so sets y to its value.

Uses a balanced approach, balancing the size of chunks of x with the valuation and hence the rate of convergence, which results in a quasi-linear algorithm in N , for fixed p .

12.1.10 Logarithm

slong **_padic_log_bound**(*slong* v, *slong* N, const *fmpz_t* p)

Returns b such that for all $i \geq b$ we have

$$iv - \text{ord}_p(i) \geq N$$

where $v \geq 1$.

Assumes that $1 \leq v < N$ or $2 \leq v < N$ when p is odd or $p = 2$, respectively, and also that $N < 2^{f-2}$ where f is FLINT_BITS.

void **_padic_log**(*fmpz_t* z, const *fmpz_t* y, *slong* v, const *fmpz_t* p, *slong* N)

void **_padic_log_rectangular**(*fmpz_t* z, const *fmpz_t* y, *slong* v, const *fmpz_t* p, *slong* N)

void **_padic_log_satoh**(*fmpz_t* z, const *fmpz_t* y, *slong* v, const *fmpz_t* p, *slong* N)

void **_padic_log_balanced**(*fmpz_t* z, const *fmpz_t* y, *slong* v, const *fmpz_t* p, *slong* N)

Computes

$$z = - \sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N},$$

reduced modulo p^N .

Note that this can be used to compute the p -adic logarithm via the equation

$$\begin{aligned} \log(x) &= \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \\ &= - \sum_{i=1}^{\infty} \frac{(1-x)^i}{i}. \end{aligned}$$

Assumes that $y = 1 - x$ is non-zero and that $v = \text{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Assumes that $v < N$, and hence in particular $N \geq 2$.

Does not support aliasing between y and z .

int **padic_log**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p -adic logarithm function converges at the p -adic number **op**, and if so sets **rop** to its value.

The p -adic logarithm function is defined by the usual series

$$\log_p(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i}$$

but this only converges when $\text{ord}_p(x-1)$ is at least 2 or 1 when $p = 2$ or $p > 2$, respectively.

int **padic_log_rectangular**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p -adic logarithm function converges at the p -adic number **op**, and if so sets **rop** to its value.

Uses a rectangular splitting algorithm to evaluate the series expression of $\log(x) \bmod p^N$.

int **padic_log_satoh**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p -adic logarithm function converges at the p -adic number **op**, and if so sets **rop** to its value.

Uses an algorithm based on a result of Satoh, Skjernaa and Taguchi that $\text{ord}_p(a^{p^k} - 1) > k$, which implies that

$$\log(a) \equiv p^{-k} \left(\log(a^{p^k}) \pmod{p^{N+k}} \right) \pmod{p^N}.$$

int **padic_log_balanced**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Returns whether the p -adic logarithm function converges at the p -adic number **op**, and if so sets **rop** to its value.

12.1.11 Special functions

void **_padic_teichmuller**(fmpz_t rop, const fmpz_t op, const fmpz_t p, slong N)

Computes the Teichmüller lift of the p -adic unit **op**, assuming that $N \geq 1$.

Supports aliasing between **rop** and **op**.

void **padic_teichmuller**(padic_t rop, const padic_t op, const padic_ctx_t ctx)

Computes the Teichmüller lift of the p -adic unit **op**.

If **op** is a p -adic integer divisible by p , sets **rop** to zero, which satisfies $t^p - t = 0$, although it is clearly not a $(p-1)$ -st root of unity.

If **op** has negative valuation, raises an **abort** signal.

ulong **padic_val_fac_ui_2**(ulong n)

Computes the 2-adic valuation of $n!$.

Note that since n fits into an **ulong**, so does $\text{ord}_2(n!)$ since $\text{ord}_2(n!) \leq (n-1)/(p-1) = n-1$.

ulong **padic_val_fac_ui**(ulong n, const fmpz_t p)

Computes the p -adic valuation of $n!$.

Note that since n fits into an **ulong**, so does $\text{ord}_p(n!)$ since $\text{ord}_p(n!) \leq (n-1)/(p-1)$.

void **padic_val_fac**(fmpz_t rop, const fmpz_t op, const fmpz_t p)

Sets **rop** to the p -adic valuation of the factorial of **op**, assuming that **op** is non-negative.

12.1.12 Input and output

char ***padic_get_str**(char *str, const padic_t op, const padic_ctx_t ctx)

Returns the string representation of the p -adic number **op** according to the printing mode set in the context.

If **str** is NULL then a new block of memory is allocated and a pointer to this is returned. Otherwise, it is assumed that the string **str** is large enough to hold the representation and it is also the return value.

int **_padic_fprint**(FILE *file, const fmpz_t u, slong v, const padic_ctx_t ctx)

int **padic_fprint**(FILE *file, const padic_t op, const padic_ctx_t ctx)

Prints the string representation of the p -adic number **op** to the stream **file**.

In the current implementation, always returns 1.

int **_padic_print**(const fmpz_t u, slong v, const padic_ctx_t ctx)

int **padic_print**(const padic_t op, const padic_ctx_t ctx)

Prints the string representation of the p -adic number **op** to the stream **stdout**.

In the current implementation, always returns 1.

void **padic_debug**(const padic_t op)

Prints debug information about **op** to the stream **stdout**, in the format "(u v N)".

12.2 padic_poly.h – polynomials over p -adic numbers

12.2.1 Module documentation

We represent a polynomial in $\mathbf{Q}_p[x]$ as a product $p^v f(x)$, where p is a prime number, $v \in \mathbf{Z}$ and $f(x) \in \mathbf{Z}[x]$. As a data structure, we call this polynomial *normalised* if the polynomial $f(x)$ is *normalised*, that is, if the top coefficient is non-zero. We say this polynomial is in *canonical form* if one of the coefficients of $f(x)$ is a p -adic unit. If $f(x)$ is the zero polynomial, we require that $v = 0$. We say this polynomial is *reduced* modulo p^N if it is in canonical form and if all coefficients lie in the range $[0, p^N)$.

12.2.2 Memory management

void **padic_poly_init**(padic_poly_t poly)

Initialises **poly** for use, setting its length to zero. The precision of the polynomial is set to **PADIC_DEFAULT_PREC**. A corresponding call to **padic_poly_clear()** must be made after finishing with the **padic_poly_t** to free the memory used by the polynomial.

void **padic_poly_init2**(padic_poly_t poly, slong alloc, slong prec)

Initialises **poly** with space for at least **alloc** coefficients and sets the length to zero. The allocated coefficients are all set to zero. The precision is set to **prec**.

void **padic_poly_realloc**(padic_poly_t poly, slong alloc, const fmpz_t p)

Reallocates the given polynomial to have space for **alloc** coefficients. If **alloc** is zero the polynomial is cleared and then reinitialised. If the current length is greater than **alloc** the polynomial is first truncated to length **alloc**.

void **padic_poly_fit_length**(padic_poly_t poly, slong len)

If **len** is greater than the number of coefficients currently allocated, then the polynomial is reallocated to have space for at least **len** coefficients. No data is lost when calling this function.

The function efficiently deals with the case where **fit_length** is called many times in small increments by at least doubling the number of allocated coefficients when length is larger than the number of coefficients currently allocated.

void **_padic_poly_set_length**(padic_poly_t poly, *slong* len)

Demotes the coefficients of **poly** beyond **len** and sets the length of **poly** to **len**.

Note that if the current length is greater than **len** the polynomial may no longer be in canonical form.

void **padic_poly_clear**(padic_poly_t poly)

Clears the given polynomial, releasing any memory used. It must be reinitialised in order to be used again.

void **_padic_poly_normalise**(padic_poly_t poly)

Sets the length of **poly** so that the top coefficient is non-zero. If all coefficients are zero, the length is set to zero. This function is mainly used internally, as all functions guarantee normalisation.

void **_padic_poly_canonicalise**(fmpz_t *poly, *slong* *v, *slong* len, const fmpz_t p)

void **padic_poly_canonicalise**(padic_poly_t poly, const fmpz_t p)

Brings the polynomial **poly** into canonical form, assuming that it is normalised already. Does *not* carry out any reduction.

void **padic_poly_reduce**(padic_poly_t poly, const padic_ctx_t ctx)

Reduces the polynomial **poly** modulo p^N , assuming that it is in canonical form already.

void **padic_poly_truncate**(padic_poly_t poly, *slong* n, const fmpz_t p)

Truncates the polynomial to length at most $\sim`n`$.

12.2.3 Polynomial parameters

slong **padic_poly_degree**(const padic_poly_t poly)

Returns the degree of the polynomial **poly**.

slong **padic_poly_length**(const padic_poly_t poly)

Returns the length of the polynomial **poly**.

slong **padic_poly_val**(const padic_poly_t poly)

Returns the valuation of the polynomial **poly**, which is defined to be the minimum valuation of all its coefficients.

The valuation of the zero polynomial is $\sim`0`$.

Note that this is implemented as a macro and can be used as either a **lvalue** or a **rvalue**.

slong **padic_poly_prec**(padic_poly_t poly)

Returns the precision of the polynomial **poly**.

Note that this is implemented as a macro and can be used as either a **lvalue** or a **rvalue**.

Note that increasing the precision might require a call to **padic_poly_reduce()**.

12.2.4 Randomisation

void **padic_poly_randtest**(padic_poly_t f, flint_rand_t state, *slong* len, const padic_ctx_t ctx)

Sets **f** to a random polynomial of length at most **len** with entries reduced modulo p^N .

void **padic_poly_randtest_not_zero**(padic_poly_t f, flint_rand_t state, *slong* len, const padic_ctx_t ctx)

Sets **f** to a non-zero random polynomial of length at most **len** with entries reduced modulo p^N .

void **padic_poly_randtest_val**(padic_poly_t *f*, *flint_rand_t* *state*, *slong* *val*, *slong* *len*, const padic_ctx_t *ctx*)

Sets f to a random polynomial of length at most *len* with at most the prescribed valuation *val* and entries reduced modulo p^N .

Specifically, we aim to set the valuation to be exactly equal to *val*, but do not check for additional cancellation when creating the coefficients.

12.2.5 Assignment and basic manipulation

void **padic_poly_set_padic**(padic_poly_t *poly*, const padic_t *x*, const padic_ctx_t *ctx*)

Sets the polynomial *poly* to the p -adic number x , reduced to the precision of the polynomial.

void **padic_poly_set**(padic_poly_t *poly1*, const padic_poly_t *poly2*, const padic_ctx_t *ctx*)

Sets the polynomial *poly1* to the polynomial *poly2*, reduced to the precision of *poly1*.

void **padic_poly_set_si**(padic_poly_t *poly*, *slong* *x*, const padic_ctx_t *ctx*)

Sets the polynomial *poly* to the signed *slong* integer x reduced to the precision of the polynomial.

void **padic_poly_set_ui**(padic_poly_t *poly*, *ulong* *x*, const padic_ctx_t *ctx*)

Sets the polynomial *poly* to the unsigned *slong* integer x reduced to the precision of the polynomial.

void **padic_poly_set_fmpz**(padic_poly_t *poly*, const *fmpz_t* *x*, const padic_ctx_t *ctx*)

Sets the polynomial *poly* to the integer x reduced to the precision of the polynomial.

void **padic_poly_set_fmpq**(padic_poly_t *poly*, const *fmpq_t* *x*, const padic_ctx_t *ctx*)

Sets the polynomial *poly* to the value of the rational x , reduced to the precision of the polynomial.

void **padic_poly_set_fmpz_poly**(padic_poly_t *rop*, const *fmpz_poly_t* *op*, const padic_ctx_t *ctx*)

Sets the polynomial *rop* to the integer polynomial *op* reduced to the precision of the polynomial.

void **padic_poly_set_fmpq_poly**(padic_poly_t *rop*, const *fmpq_poly_t* *op*, const padic_ctx_t *ctx*)

Sets the polynomial *rop* to the value of the rational polynomial *op*, reduced to the precision of the polynomial.

int **padic_poly_get_fmpz_poly**(*fmpz_poly_t* *rop*, const padic_poly_t *op*, const padic_ctx_t *ctx*)

Sets the integer polynomial *rop* to the value of the p -adic polynomial *op* and returns 1 if the polynomial is p -adically integral. Otherwise, returns 0.

void **padic_poly_get_fmpq_poly**(*fmpq_poly_t* *rop*, const padic_poly_t *op*, const padic_ctx_t *ctx*)

Sets *rop* to the rational polynomial corresponding to the p -adic polynomial *op*.

void **padic_poly_zero**(padic_poly_t *poly*)

Sets *poly* to the zero polynomial.

void **padic_poly_one**(padic_poly_t *poly*)

Sets *poly* to the constant polynomial 1, reduced to the precision of the polynomial.

void **padic_poly_swap**(padic_poly_t *poly1*, padic_poly_t *poly2*)

Swaps the two polynomials *poly1* and *poly2*, including their precisions.

This is done efficiently by swapping pointers.

12.2.6 Getting and setting coefficients

void **padic_poly_get_coeff_padic**(padic_t c, const padic_poly_t poly, *slong* n, const padic_ctx_t ctx)

Sets c to the coefficient of x^n in the polynomial, reduced modulo the precision of c .

void **padic_poly_set_coeff_padic**(padic_poly_t poly, *slong* n, const padic_t c, const padic_ctx_t ctx)

Sets the coefficient of x^n in the polynomial *poly* to c , reduced to the precision of the polynomial *poly*.

Note that this operation can take linear time in the length of the polynomial.

12.2.7 Comparison

int **padic_poly_equal**(const padic_poly_t poly1, const padic_poly_t poly2)

Returns whether the two polynomials *poly1* and *poly2* are equal.

int **padic_poly_is_zero**(const padic_poly_t poly)

Returns whether the polynomial *poly* is the zero polynomial.

int **padic_poly_is_one**(const padic_poly_t poly)

Returns whether the polynomial *poly* is equal to the constant polynomial 1 , taking the precision of the polynomial into account.

12.2.8 Addition and subtraction

void **_padic_poly_add**(*fmpz* *rop, *slong* *rval, *slong* N, const *fmpz* *op1, *slong* val1, *slong* len1, *slong* N1, const *fmpz* *op2, *slong* val2, *slong* len2, *slong* N2, const padic_ctx_t ctx)

Sets (rop, *val, FLINT_MAX(len1, len2)) to the sum of (op1, val1, len1) and (op2, val2, len2).

Assumes that the input is reduced and guarantees that this is also the case for the output.

Assumes that $\min\{v_1, v_2\} < N$.

Supports aliasing between the output and input arguments.

void **padic_poly_add**(padic_poly_t f, const padic_poly_t g, const padic_poly_t h, const padic_ctx_t ctx)

Sets f to the sum $g + h$.

void **_padic_poly_sub**(*fmpz* *rop, *slong* *rval, *slong* N, const *fmpz* *op1, *slong* val1, *slong* len1, *slong* N1, const *fmpz* *op2, *slong* val2, *slong* len2, *slong* N2, const padic_ctx_t ctx)

Sets (rop, *val, FLINT_MAX(len1, len2)) to the difference of (op1, val1, len1) and (op2, val2, len2).

Assumes that the input is reduced and guarantees that this is also the case for the output.

Assumes that $\min\{v_1, v_2\} < N$.

Support aliasing between the output and input arguments.

void **padic_poly_sub**(padic_poly_t f, const padic_poly_t g, const padic_poly_t h, const padic_ctx_t ctx)

Sets f to the difference $g - h$.

void **padic_poly_neg**(padic_poly_t f, const padic_poly_t g, const padic_ctx_t ctx)

Sets f to $-g$.

12.2.9 Scalar multiplication

```
void _padic_poly_scalar_mul_padic(mpz *rop, slong *rval, slong N, const mpz *op, slong val,
                                   slong len, const padic_t c, const padic_ctx_t ctx)
```

Sets (rop, *rval, len) to (op, val, len) multiplied by the scalar c .

The result will only be correctly reduced if the polynomial is non-zero. Otherwise, the array (rop, len) will be set to zero but the valuation *rval might be wrong.

```
void padic_poly_scalar_mul_padic(padic_poly_t rop, const padic_poly_t op, const padic_t c,
                                  const padic_ctx_t ctx)
```

Sets the polynomial rop to the product of the polynomial op and the p -adic number c , reducing the result modulo p^N .

12.2.10 Multiplication

```
void _padic_poly_mul(mpz *rop, slong *rval, slong N, const mpz *op1, slong val1, slong len1, const
                    mpz *op2, slong val2, slong len2, const padic_ctx_t ctx)
```

Sets (rop, *rval, len1 + len2 - 1) to the product of (op1, val1, len1) and (op2, val2, len2).

Assumes that the resulting valuation *rval, which is the sum of the valuations val1 and val2, is less than the precision N of the context.

Assumes that len1 $\geq$ len2 > 0 .

```
void padic_poly_mul(padic_poly_t res, const padic_poly_t poly1, const padic_poly_t poly2, const
                    padic_ctx_t ctx)
```

Sets the polynomial res to the product of the two polynomials poly1 and poly2, reduced modulo p^N .

12.2.11 Powering

```
void _padic_poly_pow(mpz *rop, slong *rval, slong N, const mpz *op, slong val, slong len, ulong e,
                    const padic_ctx_t ctx)
```

Sets the polynomial (rop, *rval, e (len - 1) + 1) to the polynomial (op, val, len) raised to the power e .

Assumes that $e > 1$ and len > 0 .

Does not support aliasing between the input and output arguments.

```
void padic_poly_pow(padic_poly_t rop, const padic_poly_t op, ulong e, const padic_ctx_t ctx)
```

Sets the polynomial rop to the polynomial op raised to the power e , reduced to the precision in rop.

In the special case $e = 0$, sets rop to the constant polynomial one reduced to the precision of rop. Also note that when $e = 1$, this operation sets rop to op and then reduces rop.

When the valuation of the input polynomial is negative, this results in a loss of p -adic precision. Suppose that the input polynomial is given to precision N and has valuation $v < 0$. The result then has valuation $ev < 0$ but is only correct to precision $N + (e - 1)v$.

12.2.12 Series inversion

void **padic_poly_inv_series**(padic_poly_t g, const padic_poly_t f, *slong* n, const padic_ctx_t ctx)

Computes the power series inverse g of f modulo X^n , where $n \geq 1$.

Given the polynomial $f \in \mathbb{Q}[X] \subset \mathbb{Q}_p[X]$, there exists a unique polynomial $f^{-1} \in \mathbb{Q}[X]$ such that $ff^{-1} = 1$ modulo X^n . This function sets g to f^{-1} reduced modulo p^N .

Assumes that the constant coefficient of f is non-zero.

Moreover, assumes that the valuation of the constant coefficient of f is minimal among the coefficients of f .

Note that the result g is zero if and only if $-\text{ord}_p(f) \geq N$.

12.2.13 Derivative

void **_padic_poly_derivative**(*fmpz* *rop, *slong* *rval, *slong* N, const *fmpz* *op, *slong* val, *slong* len, const padic_ctx_t ctx)

Sets (rop, rval) to the derivative of (op, val) reduced modulo p^N .

Supports aliasing of the input and the output parameters.

void **padic_poly_derivative**(padic_poly_t rop, const padic_poly_t op, const padic_ctx_t ctx)

Sets rop to the derivative of op, reducing the result modulo the precision of rop.

12.2.14 Shifting

void **padic_poly_shift_left**(padic_poly_t rop, const padic_poly_t op, *slong* n, const padic_ctx_t ctx)

Notationally, sets the polynomial rop to the polynomial op multiplied by x^n , where $n \geq 0$, and reduces the result.

void **padic_poly_shift_right**(padic_poly_t rop, const padic_poly_t op, *slong* n, const padic_ctx_t ctx)

Notationally, sets the polynomial rop to the polynomial op after floor division by x^n , where $n \geq 0$, ensuring the result is reduced.

12.2.15 Evaluation

void **_padic_poly_evaluate_padic**(*fmpz_t* u, *slong* *v, *slong* N, const *fmpz* *poly, *slong* val, *slong* len, const *fmpz_t* a, *slong* b, const padic_ctx_t ctx)

void **padic_poly_evaluate_padic**(padic_t y, const padic_poly_t poly, const padic_t a, const padic_ctx_t ctx)

Sets the p -adic number y to poly evaluated at a, reduced in the given context.

Suppose that the polynomial can be written as $F(X) = p^w f(X)$ with $\text{ord}_p(f) = 1$, that $\text{ord}_p(a) = b$ and that both are defined to precision $\sim N$. Then f is defined to precision $N - w$ and so $f(a)$ is defined to precision $N - w$ when a is integral and $N - w + (n - 1)b$ when $b < 0$, where $n = \deg(f)$. Thus, $y = F(a)$ is defined to precision N when a is integral and $N + (n - 1)b$ when $b < 0$.

12.2.16 Composition

```
void _padic_poly_compose(fmpz *rop, slong *rval, slong N, const fmpz *op1, slong val1, slong len1,
                        const fmpz *op2, slong val2, slong len2, const padic_ctx_t ctx)
```

Sets (rop, *rval, (len1-1)*(len2-1)+1) to the composition of the two input polynomials, reducing the result modulo p^N .

Assumes that len1 is non-zero.

Does not support aliasing.

```
void padic_poly_compose(padic_poly_t rop, const padic_poly_t op1, const padic_poly_t op2, const
                        padic_ctx_t ctx)
```

Sets rop to the composition of op1 and op2, reducing the result in the given context.

To be clear about the order of composition, let $f(X)$ and $g(X)$ denote the polynomials op1 and op2, respectively. Then rop is set to $f(g(X))$.

```
void _padic_poly_compose_pow(fmpz *rop, slong *rval, slong N, const fmpz *op, slong val, slong len,
                             slong k, const padic_ctx_t ctx)
```

Sets (rop, *rval, (len - 1)*k + 1) to the composition of (op, val, len) and the monomial x^k , where $k \geq 1$.

Assumes that len is positive.

Supports aliasing between the input and output polynomials.

```
void padic_poly_compose_pow(padic_poly_t rop, const padic_poly_t op, slong k, const padic_ctx_t
                             ctx)
```

Sets rop to the composition of op and the monomial x^k , where $k \geq 1$.

Note that no reduction takes place.

12.2.17 Input and output

```
int padic_poly_debug(const padic_poly_t poly)
```

Prints the data defining the p -adic polynomial poly in a simple format useful for debugging purposes.

In the current implementation, always returns 1.

```
int _padic_poly_fprint(FILE *file, const fmpz *poly, slong val, slong len, const padic_ctx_t ctx)
```

```
int padic_poly_fprint(FILE *file, const padic_poly_t poly, const padic_ctx_t ctx)
```

Prints a simple representation of the polynomial poly to the stream file.

A non-zero polynomial is represented by the number of coefficients, two spaces, followed by a list of the coefficients, which are printed in a way depending on the print mode,

In the PADIC_TERSE mode, the coefficients are printed as rational numbers.

The PADIC_SERIES mode is currently not supported and will raise an abort signal.

In the PADIC_VAL_UNIT mode, the coefficients are printed in the form $p^v u$.

The zero polynomial is represented by "0".

In the current implementation, always returns 1.

```
int _padic_poly_print(const fmpz *poly, slong val, slong len, const padic_ctx_t ctx)
```

```
int padic_poly_print(const padic_poly_t poly, const padic_ctx_t ctx)
```

Prints a simple representation of the polynomial poly to stdout.

In the current implementation, always returns 1.

```

int _padic_poly_fprint_pretty(FILE *file, const fmpz *poly, slong len, slong val, const char *var,
                             const padic_ctx_t ctx)
int padic_poly_fprint_pretty(FILE *file, const padic_poly_t poly, const char *var, const
                             padic_ctx_t ctx)
int _padic_poly_print_pretty(const fmpz *poly, slong len, slong val, const char *var, const
                             padic_ctx_t ctx)
int padic_poly_print_pretty(const padic_poly_t poly, const char *var, const padic_ctx_t ctx)

```

12.2.18 Testing

```

int _padic_poly_is_canonical(const fmpz *op, slong val, slong len, const padic_ctx_t ctx)
int padic_poly_is_canonical(const padic_poly_t op, const padic_ctx_t ctx)
int _padic_poly_is_reduced(const fmpz *op, slong val, slong len, slong N, const padic_ctx_t ctx)
int padic_poly_is_reduced(const padic_poly_t op, const padic_ctx_t ctx)

```

12.3 padic_mat.h – matrices over p-adic numbers

12.3.1 Module documentation

We represent a matrix over $\mathbf{Q}_p$ as a product $p^v M$, where p is a prime number, $v \in \mathbf{Z}$ and M a matrix over $\mathbf{Z}$. We say this matrix is in *canonical form* if either M is zero, in which case we choose $v = 0$, too, or if M contains at least one p -adic unit. We say this matrix is *reduced* modulo p^N if it is canonical form and if all coefficients of M lie in the range $[0, p^{N-v})$.

12.3.2 Macros

fmpz_mat_struct ***padic_mat**(const padic_mat_t A)

Returns a pointer to the unit part of the matrix, which is a matrix over $\mathbf{Z}$.

The return value can be used as an argument to the functions in the `fmpz_mat` module.

fmpz ***padic_mat_entry**(const padic_mat_t A, *slong* i, *slong* j)

Returns a pointer to unit part of the entry in position (i, j) . Note that this is not necessarily a unit.

The return value can be used as an argument to the functions in the `fmpz` module.

slong **padic_mat_val**(const padic_mat_t A)

Allow access (as L-value or R-value) to `val` field of A . This function is implemented as a macro.

slong **padic_mat_prec**(const padic_mat_t A)

Allow access (as L-value or R-value) to `prec` field of A . This function is implemented as a macro.

slong **padic_mat_get_val**(const padic_mat_t A)

Returns the valuation of the matrix.

slong **padic_mat_get_prec**(const padic_mat_t A)

Returns the p -adic precision of the matrix.

slong **padic_mat_nrows**(const padic_mat_t A)

Returns the number of rows of the matrix A .

slong **padic_mat_ncols**(const padic_mat_t A)

Returns the number of columns of the matrix A .

12.3.3 Memory management

void **padic_mat_init**(padic_mat_t A, *slong* r, *slong* c)
Initialises the matrix A as a zero matrix with the specified numbers of rows and columns and precision `PADIC_DEFAULT_PREC`.

void **padic_mat_init2**(padic_mat_t A, *slong* r, *slong* c, *slong* prec)
Initialises the matrix A as a zero matrix with the specified numbers of rows and columns and the given precision.

void **padic_mat_clear**(padic_mat_t A)
Clears the matrix A .

void **_padic_mat_canonicalise**(padic_mat_t A, const padic_ctx_t ctx)
Ensures that the matrix A is in canonical form.

void **_padic_mat_reduce**(padic_mat_t A, const padic_ctx_t ctx)
Ensures that the matrix A is reduced modulo p^N , assuming that it is in canonical form already.

void **padic_mat_reduce**(padic_mat_t A, const padic_ctx_t ctx)
Ensures that the matrix A is reduced modulo p^N , without assuming that it is necessarily in canonical form.

int **padic_mat_is_empty**(const padic_mat_t A)
Returns whether the matrix A is empty, that is, whether it has zero rows or zero columns.

int **padic_mat_is_square**(const padic_mat_t A)
Returns whether the matrix A is square.

int **padic_mat_is_canonical**(const padic_mat_t A, const padic_ctx_t p)
Returns whether the matrix A is in canonical form.

12.3.4 Basic assignment

void **padic_mat_set**(padic_mat_t B, const padic_mat_t A, const padic_ctx_t p)
Sets B to a copy of A , respecting the precision of B .

void **padic_mat_swap**(padic_mat_t A, padic_mat_t B)
Swaps the two matrices A and B . This is done efficiently by swapping pointers.

void **padic_mat_swap_entrywise**(padic_mat_t mat1, padic_mat_t mat2)
Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

void **padic_mat_zero**(padic_mat_t A)
Sets the matrix A to zero.

void **padic_mat_one**(padic_mat_t A)
Sets the matrix A to the identity matrix. If the precision is negative then the matrix will be the zero matrix.

12.3.5 Conversions

void **padic_mat_set_fmpq_mat**(padic_mat_t B, const fmpq_mat_t A, const padic_ctx_t ctx)

Sets the p -adic matrix B to the rational matrix A , reduced according to the given context.

void **padic_mat_get_fmpq_mat**(fmpq_mat_t B, const padic_mat_t A, const padic_ctx_t ctx)

Sets the rational matrix B to the p -adic matrices A ; no reduction takes place.

12.3.6 Entries

Because of the choice of the data structure, representing the matrix as $p^v M$, setting an entry of the matrix might lead to changes in all entries in the matrix M . Also, a specific entry is not readily available as a p -adic number. Thus, there are separate functions available for getting and setting entries.

void **padic_mat_get_entry_padic**(padic_t rop, const padic_mat_t op, slong i, slong j, const padic_ctx_t ctx)

Sets rop to the entry in position (i, j) in the matrix op.

void **padic_mat_set_entry_padic**(padic_mat_t rop, slong i, slong j, const padic_t op, const padic_ctx_t ctx)

Sets the entry in position (i, j) in the matrix to rop.

12.3.7 Comparison

int **padic_mat_equal**(const padic_mat_t A, const padic_mat_t B)

Returns whether the two matrices A and B are equal.

int **padic_mat_is_zero**(const padic_mat_t A)

Returns whether the matrix A is zero.

12.3.8 Input and output

int **padic_mat_fprint**(FILE *file, const padic_mat_t A, const padic_ctx_t ctx)

Prints a simple representation of the matrix A to the output stream `file`. The format is the number of rows, a space, the number of columns, two spaces, followed by a list of all the entries, one row after the other.

In the current implementation, always returns 1.

int **padic_mat_fprint_pretty**(FILE *file, const padic_mat_t A, const padic_ctx_t ctx)

Prints a *pretty* representation of the matrix A to the output stream `file`.

In the current implementation, always returns 1.

int **padic_mat_print**(const padic_mat_t A, const padic_ctx_t ctx)

int **padic_mat_print_pretty**(const padic_mat_t A, const padic_ctx_t ctx)

12.3.9 Random matrix generation

void **padic_mat_randtest**(padic_mat_t A, *flint_rand_t* state, const padic_ctx_t ctx)

Sets A to a random matrix.

The valuation will be in the range $[-\lceil N/10 \rceil, N)$, $[N - \lceil -N/10 \rceil, N)$, or $[-10, 0)$ as N is positive, negative or zero.

12.3.10 Transpose

void **padic_mat_transpose**(padic_mat_t B, const padic_mat_t A)

Sets B to A^t . Dimensions must be compatible. Aliasing is allowed for square matrices.

12.3.11 Addition and subtraction

void **_padic_mat_add**(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const padic_ctx_t ctx)

Sets C to the exact sum $A + B$, ensuring that the result is in canonical form.

void **padic_mat_add**(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const padic_ctx_t ctx)

Sets C to the sum $A + B$ modulo p^N .

void **_padic_mat_sub**(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const padic_ctx_t ctx)

Sets C to the exact difference $A - B$, ensuring that the result is in canonical form.

void **padic_mat_sub**(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const padic_ctx_t ctx)

Sets C to $A - B$, ensuring that the result is reduced.

void **_padic_mat_neg**(padic_mat_t B, const padic_mat_t A)

Sets B to $-A$ in canonical form.

void **padic_mat_neg**(padic_mat_t B, const padic_mat_t A, const padic_ctx_t ctx)

Sets B to $-A$, ensuring the result is reduced.

12.3.12 Scalar operations

void **_padic_mat_scalar_mul_padic**(padic_mat_t B, const padic_mat_t A, const padic_t c, const padic_ctx_t ctx)

Sets B to cA , ensuring that the result is in canonical form.

void **padic_mat_scalar_mul_padic**(padic_mat_t B, const padic_mat_t A, const padic_t c, const padic_ctx_t ctx)

Sets B to cA , ensuring that the result is reduced.

void **_padic_mat_scalar_mul_fmpz**(padic_mat_t B, const padic_mat_t A, const *fmpz_t* c, const padic_ctx_t ctx)

Sets B to cA , ensuring that the result is in canonical form.

void **padic_mat_scalar_mul_fmpz**(padic_mat_t B, const padic_mat_t A, const *fmpz_t* c, const padic_ctx_t ctx)

Sets B to cA , ensuring that the result is reduced.

```
void padic_mat_scalar_div_fmpz(padic_mat_t B, const padic_mat_t A, const fmpz_t c, const
                               padic_ctx_t ctx)
```

Sets B to $c^{-1}A$, assuming that $c \neq 0$. Ensures that the result B is reduced.

12.3.13 Multiplication

```
void _padic_mat_mul(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const
                    padic_ctx_t ctx)
```

Sets C to the product AB of the two matrices A and B , ensuring that C is in canonical form.

```
void padic_mat_mul(padic_mat_t C, const padic_mat_t A, const padic_mat_t B, const
                   padic_ctx_t ctx)
```

Sets C to the product AB of the two matrices A and B , ensuring that C is reduced.

12.4 qadic.h – unramified extensions over p-adic numbers

12.4.1 Data structures

We represent an element of the extension $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$ as a polynomial in $\mathbf{Q}_p[X]$ of degree less than $\deg(f)$. As such, `qadic_struct` and `qadic_t` are typedef'ed as `padic_poly_struct` and `padic_poly_t`.

12.4.2 Context

We represent an unramified extension of $\mathbf{Q}_p$ via $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$, where $f \in \mathbf{Q}_p[X]$ is a monic, irreducible polynomial which we assume to actually be in $\mathbf{Z}[X]$. The first field in the context structure is a p -adic context struct `pctx`, which contains data about the prime p , precomputed powers, the printing mode etc. The polynomial f is represented as a sparse polynomial using two arrays j and a of length `len`, where $f(X) = \sum_i a_i X^{j_i}$. We also assume that the array j is sorted in ascending order. We choose this data structure to improve reduction modulo $f(X)$ in $\mathbf{Q}_p[X]$, assuming a sparse polynomial $f(X)$ is chosen. The field `var` contains the name of a generator of the extension, which is used when printing the elements.

```
void qadic_ctx_init(qadic_ctx_t ctx, const fmpz_t p, slong d, slong min, slong max, const char
                    *var, enum padic_print_mode mode)
```

Initialises the context `ctx` with prime p , extension degree d , variable name `var` and printing mode `mode`. The defining polynomial is chosen as a Conway polynomial if possible and otherwise as a random sparse polynomial.

Stores powers of p with exponents between `min` (inclusive) and `max` exclusive. Assumes that `min` is at most `max`.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`.

This function also carries out some relevant precomputation for arithmetic in $\mathbf{Q}_p/(p^N)$ such as powers of p close to p^N .

```
void qadic_ctx_init_conway(qadic_ctx_t ctx, const fmpz_t p, slong d, slong min, slong max, const
                           char *var, enum padic_print_mode mode)
```

Initialises the context `ctx` with prime p , extension degree d , variable name `var` and printing mode `mode`. The defining polynomial is chosen as a Conway polynomial, hence has restrictions on the prime and the degree.

Stores powers of p with exponents between `min` (inclusive) and `max` exclusive. Assumes that `min` is at most `max`.

Assumes that p is a prime.

Assumes that the string `var` is a null-terminated string of length at least one.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`.

This function also carries out some relevant precomputation for arithmetic in $\mathbf{Q}_p/(p^N)$ such as powers of p close to p^N .

```
void qadic_ctx_init_modulus(qadic_ctx_t ctx, const fmpz_t p, const fmpz_mod_poly_t modulus,
                           slong min, slong max, const char *var, enum padic_print_mode
                           mode)
```

```
void qadic_ctx_init_modulus_nmod(qadic_ctx_t ctx, ulong p, const nmod_poly_t modulus, slong
                                min, slong max, const char *var, enum padic_print_mode
                                mode)
```

Initialises the context `ctx` with prime p , given defining polynomial `modulus`, variable name `var` and printing mode `mode`.

Stores powers of p with exponents between `min` (inclusive) and `max` exclusive. Assumes that `min` is at most `max`.

Assumes that p is a prime.

Assumes that `modulus` is a monic irreducible polynomial over $\mathbf{F}_p$.

Assumes that the string `var` is a null-terminated string of length at least one.

Assumes that the printing mode is one of `PADIC_TERSE`, `PADIC_SERIES`, or `PADIC_VAL_UNIT`.

This function also carries out some relevant precomputation for arithmetic in $\mathbf{Q}_p/(p^N)$ such as powers of p close to p^N .

```
void qadic_ctx_clear(qadic_ctx_t ctx)
```

Clears all memory that has been allocated as part of the context.

```
slong qadic_ctx_degree(const qadic_ctx_t ctx)
```

Returns the extension degree.

```
void qadic_ctx_print(const qadic_ctx_t ctx)
```

Prints the data from the given context.

12.4.3 Memory management

```
void qadic_init(qadic_t rop)
```

Initialises the element `rop`, setting its value to 0.

```
void qadic_init2(qadic_t rop, slong prec)
```

Initialises the element `rop` with the given output precision, setting the value to 0.

```
void qadic_clear(qadic_t rop)
```

Clears the element `rop`.

```
void _fmpz_poly_reduce(fmpz *R, slong lenR, const fmpz *a, const slong *j, slong len)
```

Reduces a polynomial (R, lenR) modulo a sparse monic polynomial $f(X) = \sum_i a_i X^{j_i}$ of degree at least 2.

Assumes that the array j of positive length `len` is sorted in ascending order.

Allows zero-padding in (R, lenR) .

void **_fmpz_mod_poly_reduce**(*fmpz* *R, *slong* lenR, const *fmpz* *a, const *slong* *j, *slong* len, const *fmpz_t* p)

Reduces a polynomial (R, lenR) modulo a sparse monic polynomial $f(X) = \sum_i a_i X^{j_i}$ of degree at least 2 in $\mathbf{Z}/(p)$, where p is typically a prime power.

Assumes that the array j of positive length len is sorted in ascending order.

Allows zero-padding in (R, lenR).

void **qadic_reduce**(qadic_t rop, const qadic_ctx_t ctx)

Reduces rop modulo $f(X)$ and p^N .

12.4.4 Properties

slong **qadic_val**(const qadic_t op)

Returns the valuation of op.

slong **qadic_prec**(const qadic_t op)

Returns the precision of op.

12.4.5 Randomisation

void **qadic_randtest**(qadic_t rop, *flint_rand_t* state, const qadic_ctx_t ctx)

Generates a random element of $\mathbf{Q}_q$.

void **qadic_randtest_not_zero**(qadic_t rop, *flint_rand_t* state, const qadic_ctx_t ctx)

Generates a random non-zero element of $\mathbf{Q}_q$.

void **qadic_randtest_val**(qadic_t rop, *flint_rand_t* state, *slong* v, const qadic_ctx_t ctx)

Generates a random element of $\mathbf{Q}_q$ with prescribed valuation val.

Note that if $v \geq N$ then the element is necessarily zero.

void **qadic_randtest_int**(qadic_t rop, *flint_rand_t* state, const qadic_ctx_t ctx)

Generates a random element of $\mathbf{Q}_q$ with non-negative valuation.

12.4.6 Assignments and conversions

void **qadic_set**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Sets rop to op.

void **qadic_zero**(qadic_t rop)

Sets rop to zero.

void **qadic_one**(qadic_t rop)

Sets rop to one, reduced in the given context.

Note that if the precision N is non-positive then rop is actually set to zero.

void **qadic_gen**(qadic_t rop, const qadic_ctx_t ctx)

Sets rop to the generator X for the extension when $N > 0$, and zero otherwise. If the extension degree is one, raises an abort signal.

void **qadic_set_ui**(qadic_t rop, *ulong* op, const qadic_ctx_t ctx)

Sets rop to the integer op, reduced in the context.

int **qadic_get_padic**(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)

If the element op lies in $\mathbf{Q}_p$, sets rop to its value and returns 1; otherwise, returns 0.

12.4.7 Comparison

int **qadic_is_zero**(const qadic_t op)

Returns whether **op** is equal to zero.

int **qadic_is_one**(const qadic_t op)

Returns whether **op** is equal to one in the given context.

int **qadic_equal**(const qadic_t op1, const qadic_t op2)

Returns whether **op1** and **op2** are equal.

12.4.8 Basic arithmetic

void **qadic_add**(qadic_t rop, const qadic_t op1, const qadic_t op2, const qadic_ctx_t ctx)

Sets **rop** to the sum of **op1** and **op2**.

Assumes that both **op1** and **op2** are reduced in the given context and ensures that **rop** is, too.

void **qadic_sub**(qadic_t rop, const qadic_t op1, const qadic_t op2, const qadic_ctx_t ctx)

Sets **rop** to the difference of **op1** and **op2**.

Assumes that both **op1** and **op2** are reduced in the given context and ensures that **rop** is, too.

void **qadic_neg**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Sets **rop** to the negative of **op**.

Assumes that **op** is reduced in the given context and ensures that **rop** is, too.

void **qadic_mul**(qadic_t rop, const qadic_t op1, const qadic_t op2, const qadic_ctx_t ctx)

Sets **rop** to the product of **op1** and **op2**, reducing the output in the given context.

void **_qadic_inv**(*fmpz* *rop, const *fmpz* *op, *slong* len, const *fmpz* *a, const *slong* *j, *slong* lena, const *fmpz_t* p, *slong* N)

Sets (**rop**, **d**) to the inverse of (**op**, **len**) modulo $f(X)$ given by (**a**, **j**, **lena**) and p^N .

Assumes that (**op**, **len**) has valuation 0, that is, that it represents a p -adic unit.

Assumes that **len** is at most d .

Does not support aliasing.

void **qadic_inv**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Sets **rop** to the inverse of **op**, reduced in the given context.

void **_qadic_pow**(*fmpz* *rop, const *fmpz* *op, *slong* len, const *fmpz_t* e, const *fmpz* *a, const *slong* *j, *slong* lena, const *fmpz_t* p)

Sets (**rop**, $2*d-1$) to (**op**, **len**) raised to the power e , reduced modulo $f(X)$ given by (**a**, **j**, **lena**) and p , which is expected to be a prime power.

Assumes that $e \geq 0$ and that **len** is positive and at most d .

Although we require that **rop** provides space for $2d - 1$ coefficients, the output will be reduced modulo $f(X)$, which is a polynomial of degree d .

Does not support aliasing.

void **qadic_pow**(qadic_t rop, const qadic_t op, const *fmpz_t* e, const qadic_ctx_t ctx)

Sets **rop** the **op** raised to the power e .

Currently assumes that $e \geq 0$.

Note that for any input **op**, **rop** is set to one in the given context whenever $e = 0$.

12.4.9 Square root

int **qadic_sqrt**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Return 1 if the input is a square (to input precision). If so, set **rop** to a square root (truncated to output precision).

struct qadic2_sqrt_precomp * **_qadic_char2_sqrt_precomp_init**(const qadic_ctx_t ctx)

int **_qadic_char2_sqrt_with_precomp**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx, const struct qadic2_sqrt_precomp *data)

void **_qadic_char2_sqrt_precomp_clear**(struct qadic2_sqrt_precomp *data)

To compute a qadic square root in characteristic 2, we need to solve the Artin-Schreier equation to get the initial square root approximation. The data needed to solve it can be precomputed once, since it depends only on the extension itself.

Note that **struct qadic2_sqrt_precomp *** is an opaque pointer; the definition of the struct is not provided.

12.4.10 Special functions

void **_qadic_exp_rectangular**(fmpz *rop, const fmpz *op, *slong* v, *slong* len, const fmpz *a, const *slong* *j, *slong* lena, const fmpz_t p, *slong* N, const fmpz_t pN)

Sets (**rop**, $2*d - 1$) to the exponential of (**op**, **v**, **len**) reduced modulo p^N , assuming that the series converges.

Assumes that (**op**, **v**, **len**) is non-zero.

Does not support aliasing.

int **qadic_exp_rectangular**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Returns whether the exponential series converges at **op** and sets **rop** to its value reduced modulo in the given context.

void **_qadic_exp_balanced**(fmpz *rop, const fmpz *x, *slong* v, *slong* len, const fmpz *a, const *slong* *j, *slong* lena, const fmpz_t p, *slong* N, const fmpz_t pN)

Sets (**rop**, **d**) to the exponential of (**op**, **v**, **len**) reduced modulo p^N , assuming that the series converges.

Assumes that **len** is in $[1, d)$ but supports zero padding, including the special case when (**op**, **len**) is zero.

Supports aliasing between **rop** and **op**.

int **qadic_exp_balanced**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Returns whether the exponential series converges at **op** and sets **rop** to its value reduced modulo in the given context.

void **_qadic_exp**(fmpz *rop, const fmpz *op, *slong* v, *slong* len, const fmpz *a, const *slong* *j, *slong* lena, const fmpz_t p, *slong* N, const fmpz_t pN)

Sets (**rop**, $2*d - 1$) to the exponential of (**op**, **v**, **len**) reduced modulo p^N , assuming that the series converges.

Assumes that (**op**, **v**, **len**) is non-zero.

Does not support aliasing.

int **qadic_exp**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Returns whether the exponential series converges at **op** and sets **rop** to its value reduced modulo in the given context.

The exponential series converges if the valuation of **op** is at least 2 or 1 when p is even or odd, respectively.

```
void _qadic_log_rectangular(fmpz *z, const fmpz *y, slong v, slong len, const fmpz *a, const slong
*j, slong lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Computes

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.$$

Note that this can be used to compute the p -adic logarithm via the equation

$$\begin{aligned} \log(x) &= \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \\ &= -\sum_{i=1}^{\infty} \frac{(1-x)^i}{i}. \end{aligned}$$

Assumes that $y = 1 - x$ is non-zero and that $v = \text{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Assumes that y is reduced modulo p^N .

Assumes that $v < N$, and in particular $N \geq 2$.

Supports aliasing between y and z .

```
int qadic_log_rectangular(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Returns whether the p -adic logarithm function converges at op , and if so sets rop to its value.

```
void _qadic_log_balanced(fmpz *z, const fmpz *y, slong len, const fmpz *a, const slong *j, slong
lena, const fmpz_t p, slong N, const fmpz_t pN)
```

Computes (z, d) as

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.$$

Assumes that $v = \text{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Supports aliasing between z and y .

```
int qadic_log_balanced(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Returns whether the p -adic logarithm function converges at op , and if so sets rop to its value.

```
void _qadic_log(fmpz *z, const fmpz *y, slong v, slong len, const fmpz *a, const slong *j, slong lena,
const fmpz_t p, slong N, const fmpz_t pN)
```

Computes (z, d) as

$$z = -\sum_{i=1}^{\infty} \frac{y^i}{i} \pmod{p^N}.$$

Note that this can be used to compute the p -adic logarithm via the equation

$$\begin{aligned} \log(x) &= \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i} \\ &= -\sum_{i=1}^{\infty} \frac{(1-x)^i}{i}. \end{aligned}$$

Assumes that $y = 1 - x$ is non-zero and that $v = \text{ord}_p(y)$ is at least 1 when p is odd and at least 2 when $p = 2$ so that the series converges.

Assumes that (y, d) is reduced modulo p^N .

Assumes that $v < N$, and hence in particular $N \geq 2$.

Supports aliasing between z and y .

int **qadic_log**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Returns whether the p -adic logarithm function converges at **op**, and if so sets **rop** to its value.

The p -adic logarithm function is defined by the usual series

$$\log_p(x) = \sum_{i=1}^{\infty} (-1)^{i-1} \frac{(x-1)^i}{i}$$

but this only converges when $\text{ord}_p(x)$ is at least 2 or 1 when $p = 2$ or $p > 2$, respectively.

void **_qadic_frobenius_a**(fmpz *rop, slong e, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)

Computes $\sigma^e(X) \bmod p^N$ where X is such that $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$.

Assumes that the precision N is at least 2 and that the extension is non-trivial, i.e. $d \geq 2$.

Assumes that $0 < e < d$.

Sets (**rop**, $2*d-1$), although the actual length of the output will be at most d .

void **_qadic_frobenius**(fmpz *rop, const fmpz *op, slong len, slong e, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)

Sets (**rop**, $2*d-1$) to Σ evaluated at (**op**, **len**).

Assumes that **len** is positive but at most d .

Assumes that $0 < e < d$.

Does not support aliasing.

void **qadic_frobenius**(qadic_t rop, const qadic_t op, slong e, const qadic_ctx_t ctx)

Evaluates the homomorphism Σ^e at **op**.

Recall that $\mathbf{Q}_q/\mathbf{Q}_p$ is Galois with Galois group $\langle \Sigma \rangle \cong \langle \sigma \rangle$, which is also isomorphic to $\mathbf{Z}/d\mathbf{Z}$, where $\sigma \in \text{Gal}(\mathbf{F}_q/\mathbf{F}_p)$ is the Frobenius element $\sigma: x \mapsto x^p$ and Σ is its lift to $\text{Gal}(\mathbf{Q}_q/\mathbf{Q}_p)$.

This functionality is implemented as `GaloisImage()` in Magma.

void **_qadic_teichmuller**(fmpz *rop, const fmpz *op, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t p, slong N)

Sets (**rop**, **d**) to the Teichmüller lift of (**op**, **len**) modulo p^N .

Does not support aliasing.

void **qadic_teichmuller**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Sets **rop** to the Teichmüller lift of **op** to the precision given in the context.

For a unit **op**, this is the unique $(q-1)$ th root of unity which is congruent to **op** modulo p .

Sets **rop** to zero if **op** is zero in the given context.

Raises an exception if the valuation of **op** is negative.

void **_qadic_trace**(fmpz_t rop, const fmpz *op, slong len, const fmpz *a, const slong *j, slong lena, const fmpz_t pN)

void **qadic_trace**(qadic_t rop, const qadic_t op, const qadic_ctx_t ctx)

Sets **rop** to the trace of **op**.

For an element $a \in \mathbf{Q}_q$, multiplication by a defines a $\mathbf{Q}_p$ -linear map on $\mathbf{Q}_q$. We define the trace of a as the trace of this map. Equivalently, if Σ generates $\text{Gal}(\mathbf{Q}_q/\mathbf{Q}_p)$ then the trace of a is equal to $\sum_{i=0}^{d-1} \Sigma^i(a)$.

```
void _qadic_norm(fmpz_t rop, const fmpz_t *op, slong len, const fmpz_t *a, const slong *j, slong lena,
               const fmpz_t p, slong N)
```

Sets `rop` to the norm of the element (op, len) in $\mathbf{Z}_q$ to precision N , where `len` is at least one.

The result will be reduced modulo p^N .

Note that whenever (op, len) is a unit, so is its norm. Thus, the output `rop` of this function will typically not have to be canonicalised or reduced by the caller.

```
void qadic_norm(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Computes the norm of `op` to the given precision.

Algorithm selection is automatic depending on the input.

```
void qadic_norm_analytic(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Whenever `op` has valuation greater than $(p - 1)^{-1}$, this routine computes its norm `rop` via

$$\text{Norm}(x) = \exp\left(\left(\text{Trace} \log(x)\right)\right).$$

In the special case that `op` lies in $\mathbf{Q}_p$, returns its norm as $\text{Norm}(x) = x^d$, where d is the extension degree.

Otherwise, raises an `abort` signal.

The complexity of this implementation is quasi-linear in d and N , and polynomial in $\log p$.

```
void qadic_norm_resultant(padic_t rop, const qadic_t op, const qadic_ctx_t ctx)
```

Sets `rop` to the norm of `op`, using the formula

$$\text{Norm}(x) = \ell(f)^{-\deg(a)} \text{Res}(f(X), a(X)),$$

where $\mathbf{Q}_q \cong \mathbf{Q}_p[X]/(f(X))$, $\ell(f)$ is the leading coefficient of $f(X)$, and $a(X) \in \mathbf{Q}_p[X]$ denotes the same polynomial as x .

The complexity of the current implementation is given by $\mathcal{O}(d^4 M(N \log p))$, where $M(n)$ denotes the complexity of multiplying to n -bit integers.

12.4.11 Output

```
int qadic_fprint_pretty(FILE *file, const qadic_t op, const qadic_ctx_t ctx)
```

Prints a pretty representation of `op` to `file`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

```
int qadic_print_pretty(const qadic_t op, const qadic_ctx_t ctx)
```

Prints a pretty representation of `op` to `stdout`.

In the current implementation, always returns 1. The return code is part of the function's signature to allow for a later implementation to return the number of characters printed or a non-positive error code.

FLOATING-POINT SUPPORT CODE

13.1 `double_extras.h` – support functions for double arithmetic

13.1.1 Random functions

double **d_randtest**(*flint_rand_t* state)

Returns a random number in the interval $[0.5, 1)$.

double **d_randtest_signed**(*flint_rand_t* state, *slong* minexp, *slong* maxexp)

Returns a random signed number with exponent between `minexp` and `maxexp` or zero.

double **d_randtest_special**(*flint_rand_t* state, *slong* minexp, *slong* maxexp)

Returns a random signed number with exponent between `minexp` and `maxexp`, zero, `D_NAN` or $\pm D\_INF$.

13.1.2 Arithmetic

double **d_polyval**(const double *poly, int len, double x)

Uses Horner's rule to evaluate the polynomial defined by the given `len` coefficients. Requires that `len` is nonzero.

double **d_mul_2exp_inrange**(double x, int i)

double **d_mul_2exp_inrange2**(double x, int i)

double **d_mul_2exp**(double x, int i)

Returns $x \cdot 2^i$.

The *inrange* version requires that 2^i is in the normal exponent range. The *inrange2* version additionally requires that both x and $x \cdot 2^i$ are in the normal exponent range, and in particular also assumes that $x \neq 0$.

13.1.3 Special functions

double **d_lambertw**(double x)

Computes the principal branch of the Lambert W function, solving the equation $x = W(x) \exp(W(x))$. If $x < -1/e$, the solution is complex, and NaN is returned.

Depending on the magnitude of x , we start from a piecewise rational approximation or a zeroth-order truncation of the asymptotic expansion at infinity, and perform 0, 1 or 2 iterations with Halley's method to obtain full accuracy.

A test of 10^7 random inputs showed a maximum relative error smaller than 0.95 times `DBL_EPSILON` (2^{-52}) for positive x . Accuracy for negative x is slightly worse, and can grow to about 10 times `DBL_EPSILON` close to $-1/e$. However, accuracy may be worse depending on compiler flags and the accuracy of the system libm functions.

int **d_is_nan**(double x)

Returns a nonzero integral value if **x** is D_NAN, and otherwise returns 0.

double **d_log2**(double x)

Returns the base 2 logarithm of **x** provided **x** is positive. If a domain or pole error occurs, the appropriate error value is returned.

13.2 d_vec.h – double precision vectors

13.2.1 Memory management

double *_**d_vec_init**(*slong* len)

Returns an initialised vector of doubles of given length. The entries are not zeroed.

void **_d_vec_clear**(double *vec)

Frees the space allocated for **vec**.

13.2.2 Randomisation

void **_d_vec_randtest**(double *f, *flint_rand_t* state, *slong* len, *slong* minexp, *slong* maxexp)

Sets the entries of a vector of the given length to random signed numbers with exponents between **minexp** and **maxexp** or zero.

13.2.3 Assignment and basic manipulation

void **_d_vec_set**(double *vec1, const double *vec2, *slong* len2)

Makes a copy of (**vec2**, **len2**) into **vec1**.

void **_d_vec_zero**(double *vec, *slong* len)

Zeros the entries of (**vec**, **len**).

13.2.4 Comparison

int **_d_vec_equal**(const double *vec1, const double *vec2, *slong* len)

Compares two vectors of the given length and returns 1 if they are equal, otherwise returns 0.

int **_d_vec_is_zero**(const double *vec, *slong* len)

Returns 1 if (**vec**, **len**) is zero, and 0 otherwise.

int **_d_vec_is_approx_zero**(const double *vec, *slong* len, double eps)

Returns 1 if the entries of (**vec**, **len**) are zero to within **eps**, and 0 otherwise.

int **_d_vec_approx_equal**(const double *vec1, const double *vec2, *slong* len, double eps)

Compares two vectors of the given length and returns 1 if their entries are within **eps** of each other, otherwise returns 0.

13.2.5 Arithmetic

void **_d_vec_add**(double *res, const double *vec1, const double *vec2, *slong* len2)

Sets (res, len2) to the sum of (vec1, len2) and (vec2, len2).

void **_d_vec_sub**(double *res, const double *vec1, const double *vec2, *slong* len2)

Sets (res, len2) to (vec1, len2) minus (vec2, len2).

void **_d_vec_mul_2exp**(double *res, const double *vec, *slong* len, int e)

Sets (res, len) to (vec, len) multiplied by 2^e .

13.2.6 Dot product and norm

double **_d_vec_dot**(const double *vec1, const double *vec2, *slong* len2)

Returns the dot product of (vec1, len2) and (vec2, len2).

double **_d_vec_norm**(const double *vec, *slong* len)

Returns the square of the Euclidean norm of (vec, len).

double **_d_vec_dot_heuristic**(const double *vec1, const double *vec2, *slong* len2, double *err)

Returns the dot product of (vec1, len2) and (vec2, len2) by adding up the positive and negative products, and doing a single subtraction of the two sums at the end. **err** is a pointer to a double in which an error bound for the operation will be stored.

double **_d_vec_dot_thrice**(const double *vec1, const double *vec2, *slong* len2, double *err)

Returns the dot product of (vec1, len2) and (vec2, len2) using error-free floating point sums and products to compute the dot product with three times (thrice) the working precision. **err** is a pointer to a double in which an error bound for the operation will be stored.

This implements the algorithm of Ogita-Rump-Oishi. See <http://www.ti3.tuhh.de/paper/rump/OgRuOi05.pdf>.

13.3 d_mat.h – double precision matrices

13.3.1 Memory management

void **d_mat_init**(d_mat_t mat, *slong* rows, *slong* cols)

Initialises a matrix with the given number of rows and columns for use.

void **d_mat_clear**(d_mat_t mat)

Clears the given matrix.

13.3.2 Basic assignment and manipulation

void **d_mat_set**(d_mat_t mat1, const d_mat_t mat2)

Sets mat1 to a copy of mat2. The dimensions of mat1 and mat2 must be the same.

void **d_mat_swap_entrywise**(d_mat_t mat1, d_mat_t mat2)

Swaps two matrices by swapping the individual entries rather than swapping the contents of the structs.

double **d_mat_entry**(d_mat_t mat, *slong* i, *slong* j)

Returns the entry of mat at row *i* and column *j*. Both *i* and *j* must not exceed the dimensions of the matrix. This function is implemented as a macro.

double **d_mat_get_entry**(const d_mat_t mat, *slong* i, *slong* j)

Returns the entry of **mat** at row *i* and column *j*. Both *i* and *j* must not exceed the dimensions of the matrix.

double ***d_mat_entry_ptr**(const d_mat_t mat, *slong* i, *slong* j)

Returns a pointer to the entry of **mat** at row *i* and column *j*. Both *i* and *j* must not exceed the dimensions of the matrix.

void **d_mat_zero**(d_mat_t mat)

Sets all entries of **mat** to 0.

13.3.3 Random matrix generation

void **d_mat_randtest**(d_mat_t mat, *flint_rand_t* state, *slong* minexp, *slong* maxexp)

Sets the entries of **mat** to random signed numbers with exponents between **minexp** and **maxexp** or zero.

13.3.4 Input and output

void **d_mat_print**(const d_mat_t mat)

Prints the given matrix to the stream **stdout**.

13.3.5 Comparison

int **d_mat_equal**(const d_mat_t mat1, const d_mat_t mat2)

Returns a non-zero value if **mat1** and **mat2** have the same dimensions and entries, and zero otherwise.

int **d_mat_approx_equal**(const d_mat_t mat1, const d_mat_t mat2, double eps)

Returns a non-zero value if **mat1** and **mat2** have the same dimensions and entries within **eps** of each other, and zero otherwise.

int **d_mat_is_square**(const d_mat_t mat)

Returns a non-zero value if the number of rows is equal to the number of columns in **mat**, and otherwise returns zero.

13.3.6 Transpose

void **d_mat_transpose**(d_mat_t B, const d_mat_t A)

Sets **B** to A^T , the transpose of **A**. Dimensions must be compatible. Aliasing is allowed for square matrices.

13.3.7 Matrix multiplication

void **d_mat_mul_classical**(d_mat_t C, const d_mat_t A, const d_mat_t B)

Sets **C** to the matrix product $C = AB$. The matrices must have compatible dimensions for matrix multiplication (an exception is raised otherwise). Aliasing is allowed.

INTERFACES

14.1 flint_ctypes - Python interface

There is a Python wrapper (`flint_ctypes`) included with FLINT available in the `src/python` directory. This wrapper is not currently officially supported and should not be used in production, but it can be useful for experimenting with FLINT.

14.1.1 Introduction

Examples:

```
>>> from flint_ctypes import *
>>> QQ.bernoulli(50)
495057205241079648212477525/66
>>> sign, primes, exponents = _.factor()
>>> sign
1
>>> primes
[5, 417202699, 47464429777438199, 2, 3, 11]
>>> exponents
[2, 1, 1, -1, -1, -1]
>>> sign * (primes ** exponents).product()
495057205241079648212477525/66
```

Types, parents and coercions

```
>>> ZZ(5)
5
>>> _.parent()
Integer ring (fmpz)
>>> QQ(5)
5
>>> _.parent()
Rational field (fmpq)
>>> ZZ(10) / ZZ(6)
Traceback (most recent call last):
...
FlintDomainError: x / y is not an element of {Integer ring (fmpz)} for {x = 10}, {y = 6}
>>> x = QQ(1) / 2; x ** x
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
FlintDomainError: x ** y is not an element of {Rational field (fmpq)} for {x = 1/2},  
↪ {y = 1/2}
```

```
>>> ZZ(10) / QQ(6)  
5/3  
>>> x = QQbar(1) / 2; x ** x  
Root a = 0.707107 of 2*a^2-1
```

Real and complex numbers

```
>>> RR.zeta(2)  
[1.644934066848226 +/- 4.57e-16]  
>>> RR.prec = 128  
>>> RR.zeta(2)  
[1.64493406684822643647241516664602518922 +/- 2.88e-39]  
>>> RR.prec = 53          # restore default
```

14.1.2 API documentation

REFERENCES

15.1 References

(In the PDF edition, this section is empty. See the bibliography listing at the end of the document.)

All referenced works: [AbbottBronsteinMulders1999], [Apostol1997], [Ari2011], [Ari2012], [Arn2010], [Arn2012], [ArnoldMonagan2011], [BBC1997], [BBC2000], [BBK2014], [BD1992], [BF2020], [BCOSSS2007], [BFSS2006], [BJ2013], [BM1980], [BZ1992], [BZ2011], [BaiWag1980], [BerTas2010], [Bin1996], [Blo2009], [Bodrato2010], [Boe2020], [Bog2012], [Bol1887], [Bor1987], [Bor2000], [Bre1978], [Bre1979], [Bre2010], [BrentKung1978], [BuhlerCrandallSompolski1992], [CFG2017], [CFG2019], [CGHJK1996], [CP2005], [Car1995], [Car2004], [Chen2003], [Cho1999], [Coh1996], [Coh2000], [Col1971], [CraPom2005], [DHBHS2004], [DYF1999], [DelegliseNicolasZimmermann2009], [DomKanTro1987], [Dup2006], [Dus1999], [EHJ2016], [EM2004], [EK2025], [Fie2007], [FieHof2014], [Fil1992], [GCL1992], [GG2003], [GS2003], [GVL1996], [Gas2018], [Gly2010], [Gos1974], [GowWag2008], [GraMol2010], [HM2017], [HS1967], [HZ2004], [HanZim2004], [Har2010], [HZ2011], [Har2012], [Har2015], [Har2018], [Hart2010], [Hen1956], [Hoe2001], [Hoe2009], [Hor1972], [Iliopoulos1989], [Igu1972], [Igu1979], [JB2018], [JM2018], [JR1999], [Joh2012], [Joh2013], [Joh2014a], [Joh2014b], [Joh2014c], [Joh2015], [Joh2015b], [Joh2016], [Joh2017], [Joh2017a], [Joh2017b], [Joh2018a], [Joh2018b], [JvdP2002], [Kahan1991], [KanBac1979], [Kar1998], [Knu1997], [Kob2010], [Kri2013], [LT2016], [Leh1970], [LukPatWil1996], [MN2019], [MP2006], [MPFR2012], [MasRob1996], [Mic2007], [Miy2010], [Mos1971], [Mul2000], [Mum1983], [Mum1984], [NIST2012], [NakTurWil1997], [NW1978], [Olv1997], [PP2010], [PS1973], [PS1991], [Paterson1973], [PernetStein2010], [Pet1999], [Pla2011], [Pla2017], [RF1994], [Rad1973], [Rademacher1937], [Ric1992], [Ric1995], [Ric1997], [Ric2007], [Ric2009], [Rosowski2023], [RosSch1962], [Rum2010], [Rys1963], [Smi2001], [SorWeb2016], [Ste2002], [Ste2010], [Stehle2010], [Stein2007], [Sut2007], [StoMul1998], [Str2014], [Str1997], [Str2012], [Tak2000], [ThullYap1990], [Tre2008], [Tru2011], [Tru2014], [Tur1953], [Villard2007], [Waksman1970], [WaktinsZeitlin1993], [Wei2000], [Whiteman1956], [Zip1985], [Zun2023], [Zun2023b], [vHP2012], [vdH1995], [vdH2006]

VERSION HISTORY

16.1 History and changes

16.1.1 FLINT version history

FLINT 3.5.0-dev

Contributors: David Sparks (DS), Fredrik Johansson (FJ), Mathieu Gouttenoire (MG)

- Smaller hash table for `n_is_prime` (FJ, DS, MG).

2025-11-25 – FLINT 3.4.0

Contributors: Albin Ahlbäck (AA), Andrii Yanovets (AY), Claus Fieker (CF), David Lowry-Duda (DLD), Dimitri Lesnoff (DL), Daniel Schultz (DS), Doug Torrance (DT), David Walker (DW), Éric Schost (ES), Fredrik Johansson (FJ), Guillem Blanco (GB), Joel Dahne (JD), Jean Kieffer (JK), kac3pro (K3), Lars Göttgens (LG), Mathieu Gouttenoire (MG), Mael Hostettler (MH), Marc Mezzarobba (MM), Pascal Molin (PM), Ricardo Buring (RB), Remi Prebet (RP), user202729 (U2), Vincent Neiger (VN).

- Features
 - Add `gr_series` as a public module (contains both the `gr_series` and `gr_series_mod` types) (FJ).
 - Add directed rounding support for `nfloat` (FJ).
 - Matrix permanents (`gr_mat_permanent`, `fmpz_mat_permanent` and `fmpq_mat_permanent`) (FJ).
 - Add the `fmpz_mod_mpoly_q` module for multivariate rational functions over a prime field (AY).
 - Gröbner basis computation for `fmpz_mod_mpoly` (AY).
 - Add `nmod_poly_minimal_irreducible` and `fq_nmod_ctx_init_minimal_weight_ui` for generating irreducible polynomials with minimal weight for any word-size prime `p` and degree `n`, and use these by default to construct a finite extension field when a Conway polynomial is not available (FJ).
 - Fast multipoint evaluation and interpolation on geometrically spaced points for `nmod_poly` (ES, VN, MH).
 - Add a debug context for generics (`gr_ctx_init_debug`) (FJ).
 - Add `_nmod_vec_invert` (VN).
 - Add `gr_ctx_ngens` and `gr_ctx_gen_name` (MM).
 - Preconditioned modular multiplication for `nmod_poly` (FJ).
 - Add uniformly random functions for various modular types (including `_nmod_vec_rand`, `nmod_mat_rand`, `fmpz_mod_mat_rand`) (DL).

- Add Kronecker substitution multiplication for `gr_poly` (FJ).
- Add `gr_poly_mullov_complex_reorder` (FJ).
- Add `_gr_mpoly_normalise` to remove zero coefficients (RB).
- Add `gr_mpoly_derivative` and `gr_mpoly_integral` (RB).
- Add `gr_derivative_gen`, with implementations for polynomial rings (RB).
- Add `gr_poly_mullov_classical` as a standalone function (FJ).
- Support `truth_t` in `flint_printf` (FJ).
- Add `delta`, `eta` parameters to `arb_mat_spd_lll_reduce` and `arb_mat_spd_is_lll_reduced` (JK).
- QR and LQ factorization of matrices (`gr_mat_qr`, `gr_mat_lq`) (FJ).
- Add `gr_mat_is_orthogonal` (FJ).
- Add `gr_mat_randtest_orthogonal` (FJ).
- Add `fmpq_poly_discriminant` (K3).
- Add `gr_mat_is_row_lll_reduced_naive` (FJ).
- Add generic Newton solver for series solutions of linear systems of ODEs with rational function coefficients (`gr_mat_gr_poly_solve_lode_newton`) (RB).
- Add `nfloat_complex_sqrt` and `nfloat_complex_rsqr` (FJ).
- Generate power series contexts in `gr_ctx_init_random` (FJ).
- Functions for converting a `gr_poly` between the standard monomial basis and a Newton basis (FJ).
- Polynomial interpolation for `gr_poly` (Newton basis and asymptotically fast) (FJ).
- Add `gr_poly_product_roots` (FJ).
- Add `fmpq_poly_interpolate_fmpq_vec` (RP).
- Add error handling to `fmpz_poly_interpolate_fmpz_vec` (FJ).
- Add `fmpz_poly_interpolate_exact` for faster exact interpolation without error handling (FJ).
- Add `gr_ctx_is_rational_vector_space`, `gr_ctx_is_real_vector_space`, `gr_ctx_is_complex_vector_space` (FJ).
- Bug fixes
 - Fix erroneous bounds in `_fmpz_mat_bound_ovals_of_cassini`, which would lead `fmpz_mat_minpoly` to compute wrong results for some matrices (CF).
 - Fix `fmpq_poly_resultant_modular_div`, `fmpz_poly_resultant_modular_div` and `nf_elem_norm_div` with non-monic polynomials (CF).
 - Fix initial value selection in `acb_poly_find_roots` preventing convergence for certain polynomials with coefficients in geometric progression (FJ).
 - Fix `gr_re` and `gr_conj` for `nfloat_complex` (FJ).
 - Fix memory corruption bug in `nmod_mpoly_divides_heap_threaded` (FJ, DS).
 - Fix and test edge cases in `fmpz_set_str` (DW).
 - Enable accidentally disabled `gr_poly` test (MM).
 - Consistently throw `FLINT_DIVZERO` when trying to divide by zero (LG).
 - Fix bug in `gr_poly_div_series_basecase`: this would previously return `GR_DOMAIN` in some cases where the quotient actually exists (FJ).

- Correct semantics for generic series: operations on `gr_series` should no longer create enclosures for elements that don't exist (FJ).
- Fix `gr_poly_shift_equivalent` in positive characteristic (MM).
- Fix a memory leak in `nmod_mpoly_gcd` and improve test coverage of this function (FJ).
- Performance
 - Use `nfloat`, `arf` and `arb` arithmetic instead of `mpf`, `mpfr` and `fmpq` in `fmpz_lll`, greatly speeding up multiprecision LLL (FJ).
 - Use SQUFOF in `fmpz_factor` to speed up factorisations of cofactors in the 65-80 bit range (FJ).
 - Reduce multiplication depth in `_gr_poly_compose_axnc` (U2).
 - Improve precision in `arb_set_str`, making it more likely to round to nearest (DW).
 - Minor speedup for `fmpz_poly_compose` (U2).
 - Speed up `n_is_prime` by replacing the BPSW test with smarter strong probable prime tests and other optimisations (FJ).
 - Speed up `n_factor_one_line` and `n_factor_SQUFOF` by improving the square root computation (FJ).
 - Speed up `n_factor` by optimising trial division, doing earlier primality testing, and using the one line factor algorithm for larger bit sizes (FJ).
 - When possible speed up `nmod_poly` multiplication via `fft_small` by a factor two for tiny moduli by packing a linear polynomial into each coefficient (FJ).
 - Add AVX2 version of `nmod_vec_dot_half` (VN).
 - Improve speed and accuracy for `arb_min` and `arb_max` with non-overlapping input (JD).
 - Speed up `_fmpz_vec_scalar_divexact_si`, `_fmpz_vec_scalar_divexact_ui` and small-divisor `_fmpz_vec_scalar_divexact_fmpz` (FJ).
 - Make multivariate polynomial multiplication via nested `gr_poly` over various ground types asymptotically fast (FJ).
 - Speed up `_nmod_poly_divrem_newton_n_preinv` for sparse moduli by handling these specially (FJ).
 - Incorporate trial division in `nmod_poly_is_irreducible` when the prime is small (FJ).
 - Speed up `nmod_poly` modular composition by using rectangular splitting instead of Horner for the third step of Brent-Kung (FJ).
 - Faster preconditioned multiplication and polynomial GCD for `mpn_mod` (FJ).
 - Improve `fmpq_poly_interpolate_fmpz_vec` by incorporating an asymptotically fast multi-modular algorithm and other improvements (RP).
 - Improve `fmpz_poly_interpolate_fmpz_vec` by incorporating an asymptotically fast multi-modular algorithm and other improvements (FJ).
 - Make various `gr_poly` functions more robust or performant when dealing with inexact representations of zero (FJ).
- Build system, general maintenance
 - Fix issues with profiling code on Clang (AA).
 - Mark many internal functions as `static` (AA).
 - Compile with `-Wmissing-prototypes` by default (AA).
 - Fix many compiler warnings (AA).
 - Fix `acb_theta_all` and `acb_theta_one` disappearing from the shared library (AA).

- Add V Makefile variable for verbose builds (DT).
- Add link-time optimization configure option (MG).
- Fix situations where the compiler runs out of registers for inline assembly on x86 (AA).
- Fix static linkage of `d_randtest` (AA).
- Remove the obsolete `mpfr_vec` and `mpfr_mat` modules (FJ).
- Refactor `fmpz_111` to reduce code size and improve clarity (FJ).
- Refactor power series code (FJ).
- Add `FLINT_NORETURN` to function pointer argument (AA).
- Test `msolve` in CI (VN).
- Corrections and improvements to the documentation (AA, DL, DLD, GB, LG, U2).
- Example programs
 - Add example program to compute pi using AGM iteration (FJ).
 - Add example program for computing coefficients of modular forms (PM).

2025-06-16 – FLINT 3.3.1

- Bug fixes
 - Set permutation vector in `gr_mat_lu_classical` even if there are zero columns (Fredrik Johansson).

2025-06-11 – FLINT 3.3.0

Main contributors: Albin Ahlbäck (AA), Ricardo Buring (RB), Lars Götting (LG), Max Horn (MH), Fredrik Johansson (FJ), Jean Kieffer (JK), Marc Mezzarobba (MM), Vincent Neiger (VN).

- Features
 - Add generic fraction fields (`gr_ctx_init_gr_fraction`) (FJ).
 - Allow rings to overload `_gr_mat_charpoly` and provide such overloads for `fmpz`, `fmpq`, `mpn_mod` and `fmpz_mod`. Furthermore, remove `fq*_mat_charpoly_danilevsky` and `fmpz_mod_mat_charpoly_berkowitz`, and wrap `fq*_mat_minpoly` and `fq*_mat_charpoly` around GR functions (FJ).
 - Remove `fq_zech_mat_reduce_rows` (FJ).
 - Add various `mulmod`, `div_preinv` and `compose_mod` methods for `gr_poly` (FJ).
 - Add `fmpz_mod_mat_pow_ui` (LG).
 - Add `gr_poly_scalar_mul` (FJ).
 - Implement `GR_METHOD_OTHER_MUL` for `gr_poly` (FJ).
 - Add `gr_mat_pow_(si|fmpz|fmpq|scalar)` (LG, FJ).
 - Add `gr_mat_func_jordan` (FJ).
 - Add `nmod_poly_mat_transpose` and `fq_default_mat_transpose` (LG).
 - Add different `powmod`-methods for `gr_poly`, based on methods from `fq_poly` (FJ).
 - Add `gr_poly_(add|sub)_scalar` (RB).
 - Support GR objects in `flint_printf` (FJ).
 - Add support for parsing builtin functions like `sqrt(x)` in `gr_set_str` (FJ).

- Recognize when generic polynomial rings are UFDs (MM).
- Add `flint_set_throw` (AA).
- Add `gr_generic_poly_factor_roots` for factoring generic polynomials over algebraically closed fields providing `gr_poly_roots` (MM).
- Add `gr_factor` for `gr_poly` (MM).
- Add `arf_nint` (FJ).
- Add `gr_poly_gcd_subresultant` (FJ).
- Add generic `gr_gcd` for fields (FJ).
- Add `gr_mpoly_inv` (FJ).
- Implement `gr_poly_gcd` over any UFD, not only fields (FJ).
- Add `gr_canonical_associate` and corresponding implementations for several types (FJ).
- Add `gr_mat_companion` and `gr_mat_companion_fraction` to construct companion matrices (RB).
- Add `gr_randtest_invertible` (RB).
- Add `gr_mat_randsimilar` (RB).
- Add `flint_merge_sort` and `flint_sort` (MM).
- Add `gr_vec_sort` for sorting generic vectors (MM).
- Add `_gr_vec_shuffle`, `gr_vec_permute`, `gr_vec_contains` (MM).
- Add `gr_in_domain` and `gr_check` (MM).
- Add `gr_poly_leading_taylor_shift` (MM).
- Add shift equivalence, dispersion, and shiftless decomposition of generic polynomials (`gr_poly_shift_equivalent`, `gr_poly_dispersion`, `gr_poly_shiftless_decomposition`) (MM).
- Add `gr_mat_mul_rosowski` (FJ).
- Add `acb_theta_one` to compute a single theta function (JK).
- Add parameter `ab` in `acb_theta_jet` (JK).
- Examples
 - Add example script to check `n_is_prime` against list of Fermat pseudoprimes (FJ).
 - Replace generated code with strings in `huge_expr.c` (FJ).
- Performance
 - Use stride instead of row pointers in all matrix types (FJ).
 - Refactor characteristic polynomial and retune it (FJ).
 - Use delayed canonicalization in `mpn_mod_mat_reduce_row` and `fmpz_mod_mat_reduce_row` (FJ).
 - Improve method choice in `gr_mat_rank` (FJ).
 - Inline operations in `_gr_fmpz_set` (FJ).
 - Use `udiv_qrnmnd` in `n_preinvert_limb` whenever CPU has fast division (AA, FJ, VN).
 - Have `gr_divexact` over a `gr_poly` ring dispatch to `gr_poly_divexact` (FJ).
 - Rewrite the `acb_theta` module for substantially improved performance (JK).
 - Overload `_gr_vec_mul` for `nmod` types (FJ).
 - Optimize `_fmpz_vec_add` and `_fmpz_vec_sub` for small coefficients by inlining (FJ).

- Bug fixes
 - Fix bug in `fmpz_mat_charpoly_modular` where wrong results were given if output polynomial was not initially set to zero by caller (FJ).
 - Fix memory leak in `gr_mat_minpoly_field` (FJ).
 - Fix aliasing for `gr_poly_div_newton_n_preinv` (FJ).
 - Fix memory leak in `gr_mat_fflu` (FJ).
 - Fix invalid write in `_gr_nmod_vec_reciprocals` when length is zero (FJ).
 - Fix various memory leaks found by the address sanitizer (RB).
 - Fix cases where `gr_poly` were not correctly normalized over a degenerate ring (FJ).
 - Fix `gr_set_str` with duplicate generator names (MM).
 - Fix `fmpz_mat_minpoly` and `fmpq_mat_minpoly` to return the correct result for the zero matrix (MH).
 - Fix bug in `arb_nint` (FJ).
 - Fix `fmpz_mat_minpoly_modular` with large coefficients (FJ).
 - Fix context bug in `_gr_gr_series_mod_gen` (RB).
 - Propagate failure in `acb_poly_find_roots` if numerical stage returns NaN (FJ).
 - Fix bug in `arb_div` giving NaN when the numerator contained zero at high precision (FJ).
 - Fix NaN output for `[+/-inf +/- inf]` in `arb_get_interval_arf` and `arb_get_interval_mpf` (FJ).
 - Correct signature of `gr_ctx_clear` to have void return type (FJ).
 - Fix function signatures in internal callbacks for `flint_parallel_binary_splitting` (Oscar Benjamin, FJ).
- Build system
 - Push pragmas into macros, allowing them to be set during configuration (AA).
 - Fix an overwriting of CFLAGS in `configure.ac` (AA).
 - Re-enable `-Wmaybe-uninitialized` (FJ).
 - Enable `-Werror=newline-eof` (AA).
 - Accommodate for Meteor Lake, Raptor Lake and Tiger Lake (VN).
 - Make sure Comet Lake enables `fft_small` (VN).
 - Error when trying to build FLINT on a non-Windows system (AA).
 - Do not use `find_library()` in `flint_ctypes` (MM).
- Tests
 - Add more test code for preinv division for `gr_poly` (FJ).
 - Add tests for `gr_mat_exp` and `gr_mat_log` (FJ).
 - Add test for `fmpz_mod_mat_transpose` (LG).
 - Test `gr_poly` over random rings (MM).
 - Have `gr_ctx_init_random` generate polynomial contexts (FJ).
- Profiling
- Maintenance
 - Replace some `fq_poly` and `fmpz_mod_poly` methods with GR wrappers (FJ).

- Include standard headers instead of using builtins (AA).
- Refactor `gr_mat_exp_jordan` and `gr_mat_log_jordan` (FJ).
- Delegate `(fmpz|fmpz_poly|fmpq)_mat_transpose` to `gr_mat` (LG).
- Add issue templates to Github (AA, Lars Kastner).
- Optimize for code size in some less performance-critical functions (FJ).
- Fix various undefined behaviors reported by GCC's address sanitizer (FJ).
- Add `FLINT_UABS` macro (FJ).
- Refactor `nullspace` and `rref` in GR, and make `fq*_mat` use these instead (FJ).
- Use `gr_poly`-methods for `powmod` in `fmpz_mod_poly` and `fq*_poly` (FJ).
- Use `gr_mat_rank_lu` in `fmpz_mod_mat_rank` and `fq*_mat_rank` (FJ).
- Remove underscore in front of `nfloat_ctx_(set|get)_real_prec` to match documentation (Joel Dahne).
- Add missing newlines at end of file (FJ).
- Fix `ENABLE_AVX2=OFF` for CMake (AA).
- Add `gr_poly_coeff_ptr` and deprecate `gr_poly_entry_ptr` (RB).
- Continuous integration
 - Use temporary Nemo branch for CI due to the change of using stride instead of row pointers in matrices (AA).
- Documentation
 - Fix `fmpz` docstrings to clarify that GCD of integers and rationals is non-negative (MH).
 - Unify some docstrings (LG).
 - Fix docstrings for various LU functions (LG).
 - Fix typos (AA, MM, user202729).
 - Touch up formula in `hypgeom.rst` (FJ).
 - Fix docstring for `fmpz_mat_snf_kannan_bachem` (Dima Pasechnik).
 - Fix docstring for `fmpz_get_nmod` (David Lowry-Duda).
 - Minor fix in documentation of `ulong_extras` (VN).
 - Document `n_factor_t` structure (Matthew House).
 - Document `gr_vec_entry_srcptr` (MM).
 - Document `truth_t` functions (MM).

2025-03-31 – FLINT 3.2.2

Main contributors: Albin Ahlbäck (AA)

- Continuous integration
 - Enable redoing upload release CI (AA).
- Build system
 - Fix installation of headers for out-of-tree builds (AA).
 - Use `__ARM_ACLE` to guard inclusion of `arm_acle.h` (Lars Göttingens).
- Maintenance

- Remove assertion with uninitialized variable (AA).

2025-03-17 – FLINT 3.2.1

- Bug fixes
 - Properly recognize AMD Zen during configuration (Albin Ahlbäck, reported by Doug Torrance).

2025-03-12 – FLINT 3.2.0

Main contributors: Albin Ahlbäck (AA), Bill Allombert (BA), Ricardo Buring (RB), Edgar Costa (EC), Tobias Diez (TD), Isuru Fernando (IF), Tommy Hofmann (TH), Max Horn (MH), Fredrik Johansson (FJ), Marc Mezzarobba (MM), Jake Moss (JM), Vincent Neiger (VN), Giacomo Pope (GP), Daniel Schultz (DS).

- Features
 - Add new module `mpn_mod` for fixed-size few-word modulo arithmetic, with additional vector, matrix and polynomial arithmetic (FJ).
 - Add new module `nfloat` for packed fixed-size floating-point numbers with additional vector and matrix arithmetic (FJ).
 - Add fixed-point arithmetic under the `nfloat` module (FJ).
 - Major work on `gr` modules, including:
 - * Toom-Cook multiplications for `gr_poly`,
 - * Add Aberth and WDK root refinement for `gr_poly`,
 - * Handle valuations and exact results in `gr_series_div`,
 - * Addition of missing modular arithmetic routines,
 - * Addition of comparison routines,
 - * Matrix norms,
 - * Improvements of Strassen matrix multiplication,
 - * Addition of Waksman matrix multiplication,
 - * Add all remaining scalar arithmetic methods for `gr_mat`,
 - * Refactoring of `gr_mpoly` contexts,
 - * Improve equality testing code for `gr_mpoly`.

All was contributed by (FJ).

- Implement computing qqbar roots of qqbar polynomials (FJ).
- Implement generic `flint_mpn_mulhigh` and `flint_mpn_sqrhigh` for all ranges based on Mulders’ algorithm (FJ).
- Implement `n_factor_evaluate` (AA).
- Wrap some more methods in `flint_ctypes` (FJ).
- Let `flint_mpn` random functions take `flint_randstate_t` as input (FJ).
- Add `fmpz_randbits_unsigned` (AA).
- Add special case for `_acb_dirichlet_zeta` whenever s is a positive integer (FJ).
- Typedef `ulong` and `slong` based on C primitives rather than relying on GMP’s `mp_limb_t` and `mp_limb_signed_t` (AA).

- Refactor FLINT’s base randomisation, and rename the types and functions to be in proper FLINT-style.
- Add some missing GR methods for modular arithmetic (FJ).
- Add `vec4n_(zero|mul|permute)` in `machine_vectors.h` (VN).
- Add `cmpabs` routines for `acb` and `acf` (FJ).
- Add `fq_default_ctx_init_randtest` (GP).
- Add some missing `fmpz_mod_mpoly_compose` functions (JM).
- Improve enclosures in exponential integrals (FJ).
- Improve transition regions and on wide intervals in `acb_hypgeom_bessel_j` (FJ).
- Allow evaluation of the deflated Riemann zeta function on small intervals containing 1 (FJ).
- Allow evaluation of `a[rc]b_poly_sinc_series` and `a[rc]b_poly_sinc_pi_series` on intervals containing 0 (FJ).
- Examples
 - Add AKS primality example program (FJ).
 - Add example on double exponential integration (Hartmut Monien).
 - Update integral examples (FJ).
- Performance
 - Implement `flint_mpn_2add_n_inplace` for x86_64 architectures supporting the ADX instruction set for adding two n limbed integers onto another n limbed integer inplace, returning the carry (AA).
 - Replace `flint_mpn_divexact_1` with `mpn_divexact_1` (AA).
 - Add `flint_mpn_mul_Xn` for $X < 16$ on Arm v8, outperforming GMP on Apple M1 (AA).
 - Add `flint_mpn_mul_2` for Arm v8 (AA).
 - Push some parameters into `flint-mparam.h`, mainly thresholds for FFT multiplication, currently only for Skylake, Zen 3 and Apple M1 (AA).
 - Add inline assembly version of `MPN_IORD_U`, where the x86 version is taken from GMP, and an Arm v8 version was added as well (AA).
 - Use `flint_mpn_mulhigh` in `mpn*_preinvn` methods (FJ).
 - Avoid 8x excessive memory allocation in `fmpz_preinvn` functions (FJ).
 - Use `flint_mpn_mulhigh` for unbalanced preinvn divisions (FJ).
 - Minimize 32-bit instructions in x86_64 assembly.
 - Add `flint_mpn_mullo_N` for $N \leq 8$ on x86_64 architectures supporting the ADX instruction set (AA).
 - Add `flint_mpn_mullo_basecase` for x86_64 architectures supporting the ADX instruction set (AA).
 - Micro-optimize `flint_mpn_mulhigh` (AA).
 - Add `flint_mpn_sqr_N` routines for $N \leq 9$ for Arm v8 (AA).
 - Add `flint_mpn_mulhigh_N` routines for $N \leq 8$ for Arm v8 (AA).
 - Add `flint_mpn_sqrhigh_N` routines for $N \leq 8$ for Arm v8 (AA).
 - Add `_flint_mpn_mulhigh_basecase` routines Arm v8 optimized for Apple M1 (AA).
 - In gr matrix rings, call `gr_mat_mul` rather than `gr_mat_mul_classical` (FJ).

- Change generic truncated power series to use `gr_poly` instead of `gr_series` as the data type (FJ).
- Optimize `nmod_poly_gcd` with 64-bit moduli (FJ).
- Do not calculate remainder in exact division of `fq_nmod_poly` unless assertion is needed.
- Implement fast `flint_mpn_mullov_n` for large n , and use this in preinversion functions (FJ).
- Simplify `FLINT_MPN_MUL_2X2` (FJ).
- Reduce inlining of `flint_mpn_mulhigh_normalised` (FJ).
- Implement faster version of `flint_mpn_addmod_2` (FJ).
- Use `flint_mpn_(add|sub)mod_n` in the `fmpz_factor` module (FJ).
- Use `fmpz_poly_divexact` instead of `fmpz_poly_divides` in one place (FJ).
- Inline `n_randint` to avoid division operations when limit is compile-time constant (AA).
- Simplify `fmpz_randprime`, and rely on `n_randprime` for bitsizes up to `FLINT_BITS` (AA).
- Skip up to one call to `fmpz_equal` in `fmpz_is_strong_probabprime` (Mathieu Gouttenoire).
- Optimize `n_revbin` for Aarch64 (AA).
- Use builtin check arithmetic in `ulong_extras.h` and `long_extras.h`, where it was useful (AA).
- Use `_fmpz_new_mpz` instead of `mpz_init` to allow the use of already allocated `mpz` (AA).
- Improve formula selection for `acb_hypgeom_beta_lower` (FJ).
- Write proper function for `fmpz_mul2_uiui` (AA).
- Replace some `n_[cf]log(X, 2)` to `FLINT_[CF]LOG2(X)` (AA).
- Replace `malloc + memset` calls by `calloc` in `block_lanczos` (AA).
- Improve tuning for `_mpz_tdiv_qr_preinvn` (FJ).
- Use preinversions in more cases in `fmpz_mod` (FJ).
- Rewrite `fmpz_lucas_chain` to use various $\mathbb{Z}/n\mathbb{Z}$ implementations (FJ).
- Use Bini’s algorithm to select initial values in `acb_poly_find_roots` (FJ).
- Speed up `nmod` dot products, including vectorized routines (VN).
- Use `nmod` dot products more throughout FLINT (VN).
- Implement more modular multiplication routines with precomputed inverses (VN).
- Add tuning for Intel Icelake (VN).
- Avoid computing some entries twice in Strassen multiplication for `(gr|fmpz)_mat` (FJ).
- Utilize symmetry when squaring in `gr_mat_mul_strassen` (FJ).
- Improve algorithm selection in `fmpz_mat_rref` (FJ).
- Use `acb_dot` in recurrence for Eisenstein series instead of multiple `addmuls` (FJ).
- Bug fixes
 - Fix segfault in `examples/fq_poly` (FJ, reported by Andrea Lesavourey).
 - Fix uninitialized and uncleared variables (FJ).
 - Fix wrong stack usage in x86 and Arm assembly routines (AA).
 - Fix missing break statement in example program (FJ).
 - Fix assertion in `flint_mpn_mulhigh` (AA).
 - Fix bug in `gr_series` (FJ).

- Fix primitive root prime (VN).
- Fix `gr_ctx_is_finite_characteristic` for `fmpz_mod` (FJ).
- Fix setting generator names for univariate `gr` rings (FJ).
- Fix memory usage in x86 `sqrhigh` assembly where out-of-bound elements were accessed (AA).
- Fix bugs in `fq_default` context initialisers (AA).
- `fmpz_remove` may reduce the number of limbs allocated, which may silently reduce the number of limbs contained in a big `fmpz`. Thus, ensure that after each call to `fmpz_remove`, the result has a proper number of limbs allocated (AA).
- Fix leading coefficient bug in `fmpz_mpoly_factor` (DS).
- Fix bug in `fq_default_poly_evaluate_fq_default` where the check for `fmpz_mod` representation was incorrect (GP).
- Fix division by zero in `mpoly_monomials_deflate` (JM).
- Fix bug introduced in new function `fq_default_ctx_init_randtest` (AA).
- Fix crash potential crashes in `fmpz_poly_sub_fmpz` (MH).
- Fix modulus 1 for `nmod_poly` and `fmpz_poly` modules (TH).
- Fix name clashes of variables in macros, as this becomes a problem whenever macros are layered (AA).
- Fix `NEED_CLZ_TAB` (AA).
- Patch a bug in Clang where it assumes `__builtin_clz` always returns something non-zero (AA).
- Patch certain tests who caused segfaults during compilations in Clang (AA).
- Avoid inline assembly for big `aors_n` in GCC on ARM since they lack enough registers to compile this (AA).
- Fix wrong arithmetic function used in `n_is_probabprime_lucas` (FJ, reported by Mikhail Hogrefe).
- Correct pointer casting for ARM in `machine_vectors.h` (AA).
- Build system
 - Add GMP's `config.guess` and utilize it (AA).
 - Check for more CPUs in `config.guess`, including Intel Comet Lake and Github's Apple M1 virtual runner (AA).
 - Validate more CPUs in `config.guess`, including `x86_64v3` (AA).
 - Set compiler architecture dependent flags depending on `$host` in Autotools (AA).
 - Add check for Aarch64/Arm v8 (AA).
 - Search for more GMP internal functions (AA).
 - Implement parameter files `flint-mparam.h` based on architecture (AA).
 - Create an include directory to build examples in `Makefile` (BA).
 - Fix missing header (VN).
 - Use CXX when testing NTL (George Huebner).
 - Check for GMP's `mpn_invert_limb` during build (AA).
 - Add `-Wno-maybe-uninitialized` to default C flags (AA).

- Enable backup Cflags in the configuration, such as if `-march=znver4` is prioritised over `-march=znver3`, then it will first try `-march=znver4` and fall back to `-march=znver3` if necessary.
- Deprecate the `--enable-tls` option in the configuration (AA).
- Add maintainer-level only option for fast build where optimizations are disabled (AA).
- Recognize Comet Lake U in `config.guess` (AA).
- Push non-executable stack when compiling assembly files (AA).
- Remove unused compiler options for Clang, hence silencing warnings (AA).
- Make Autotools build work out of tree (AA).
- Don't assume AVX is available if user specified their own CFLAGS (AA).
- Use `cp -pRP` instead of `cp -a` in Make as the latter is not supported by POSIX, causing installation failures on some systems (AA).
- Add C11 atomics for MSVC builds (IF).
- Recognize Apple M3 and Apple M4 (Pro) in `config.guess` (AA).
- Fix static build for MSVC (IF, TD).
- Only check assembly labels when it is actually being used (AA).
- Update `X86_64_PATTERN` and `X86_64_ADX_PATTERN` and add some missing patterns (AA).
- Recognize and map Zen 5 with correct CFLAGS and set some initial parameters for the architecture (AA).
- Check presence of `install-sh` instead of `config.guess` during bootstrapping as `config.guess` is always present (AA).
- Tests
 - Fix test for `flint_mpn_mul` where testing for big inputs was not done correctly (AA).
 - Cleanup tests related to assembly routines in `mpn_extras` (AA).
 - Add more test code for `gr_series` (FJ).
 - Fix warning in test (AA).
 - Add functions `gr_mat_test_mul`, `gr_mat_test_lu` and `gr_poly_test_mullow` for testing generics overrides (FJ).
 - Avoid `mpn_random2` in tests (FJ).
 - Check correct type initialisation in `fq_default` (AA).
 - Add stricter bound to test of `flint_mpn_mulhigh` (AA).
 - Revise tests for `ulong_extras` (AA).
 - Reduce test multiplier where it was excessive (AA).
 - Speedup the test for `flint_mpn_mul_toom22` (AA).
 - Merge tests for `nmod_mat_(add|sub)mul` (AA).
 - Simplify test for `fmpz_randprime`, and remove the risk this test failing (AA).
 - Revise test for `fmpz_init_set_readonly` (AA).
 - Add missing abort in test for `arb_mat_solve_preapprox` (AA).
 - Add more tests for `gr` module (FJ).
 - Add test for `dlog_rho_t` (FJ).
 - Add test for `fq_default_poly_evaluate` (GP).

- Correct initialisation-utilisation-clearing order in test for `fq_default_init` (GP).
- Add test for division by zero in `fmpz_poly_(inflate|deflate)` (JM).
- Uncomment some previously commented code in `fmpz_mod_mpoly_compose` test, use smaller primes for this test (JM).
- Use `TEST_FUNCTION_FAIL` macro in more multivariate polynomial tests (JM).
- Explicitly check zero polynomial compositions for most multivariate polynomial modules (JM).
- Randomize modulus in multivariate polynomial tests (JM).
- Profiling
 - Add profile program for `flint_mpn_divrem_preinvn` (FJ).
 - Add profile program for `flint_mpn_mulow` (AA).
 - Fix `p-mulmod_preinvn` (FJ).
 - Add profile program for dot products in the `nmod_vec` module (VN).
 - Add profile program for `nmod_vec_scalar_addmul*` (VN).
 - General profile program fixes and touchups on the `nmod` modules (VN).
- Maintenance
 - Add notes that CMake is only recommended for Windows users (AA).
 - Convert TODO from txt-format to Markdown (AA).
 - Generate and install CMake configuration files (Mehdi Chinoune).
 - Define GMP internal functions, if they are available, in `mpn_extras.h` (AA).
 - Change `#ifdef FLINT_HAVE_FFT_SMALL` to `#if FLINT_HAVE_FFT_SMALL` (AA).
 - Enable running specific tests in modules via `make check MOD=XXX ARGS=YYY` (AA).
 - Add Codecov key to CI (AA, FJ).
 - Force Unix-type newlines through git on Cygwin CI (AA).
 - Remove debug code from `qqbar_roots_poly_squarefree` (FJ).
 - Remove `FLINT_HAVE_AVX*` definitions (AA).
 - Update `flint-config.h.in` (AA).
 - Fix a macro (?) (FJ).
 - Simplify definition of `mp_limb_pair_t` (FJ).
 - Use `#include <flint/xxx.h>` in examples (BA).
 - Change `__mpz_struct *` to `mpz_ptr` (EC).
 - Move `mpn` macros from `flint.h` to `mpn_extras.h` (AA)
 - Enable compiling with `-Wextra -Werror` for a big part of the library (AA).
 - Disable static build by default (AA).
 - Do not remove intermediate assembly files for making debugging easier (AA).
 - Add `make debug MOD=XXX ARGS=YYY` shortcut for debugging with GDB (AA).
 - Add `gdb_history` and `vgcore.*` to `.gitignore` (AA).
 - Replace “FLINT2” by “FLINT” (Julian R  th).
 - Remove obviously outdated entries in `TODO.md` (FJ).
 - Remove unnecessary inclusions of `mpn_extras.h` (FJ).

- Remove some undocumented and/or unused functions, including most of `mpfr_*` modules and `n_cbrt_newton_iteration` (AA, FJ).
- Change `__mpz_struct *` to `mpz_ptr` (AA).
- Add keywords `FLINT_MALLOC` and `FLINT_RETURNS_NONNULL`. Use these along with `FLINT_WARN_UNUSED` whenever suitable in the memory managing functions in `flint.h` (AA).
- Remove inclusion of `gmp.h` and `limits.h` in `flint.h` and other headers (AA).
- Use FLINT types instead of GMP types in source code (AA).
- Remove `signed_mpn_sub_n` in favor of `flint_mpn_signed_sub_n` (AA).
- Cleanup copyright claims in `longlong.h` (AA).
- Remove `byte_swap` from `longlong.h` (AA).
- Remove `flint_mul_sizes` and use `z_mul_checked` instead (AA).
- Remove `r_shift` and `l_shift` from `flint.h`, and only define where needed (AA).
- Remove inclusion of `longlong.h` from `flint.h` (AA).
- Remove old deprecations in `fmpz.h` (AA).
- General header cleanups and removal of unnecessary inclusions in them (AA).
- Replace calls to `mpz_realloc2` to `mpz_realloc`, and only do these calls if it is truly necessary (AA).
- Add constant `MPZ_MIN_ALLOC` that decides the minimum number of allocated limbs that the `fmpz-collector` may receive such that `_fmpz_new_mpz` always returns an `mpz` with at least `MPZ_MIN_ALLOC` limbs allocated (AA).
- Merge several source files (AA).
- Simplify `MAG_FIXMUL` for 32-bit systems (AA).
- Delete `acb_vec_printd_index`, and only define where needed in test files (AA).
- Add flags to `gr_acb_ctx`, currently only analytic flag implemented (FJ).
- Further maintenance of `fft_small` (DS).
- Increase code coverage in tests (AA, FJ).
- Use `_mm256_srli_epi64` instead of `_mm256_srl_epi64` in `machine_vectors.h` (VN).
- Clarify that `libtool` is needed in `INSTALL.md` and `README.md` (AA).
- Do not use `I` as a variable name as it is reserved in C99 (Dima Pasechnik).
- Deprecate aliases in the `arith` module (AA).
- Move `add_s{n}a{2n}` and friends from `crt_helpers.h` to `longlong.h` (FJ).
- Rename `FLINT_TEST_CLEANUP` to `FLINT_TEST_CLEAR` (VN).
- Add multiple missing `flint_rand_clear` (VN).
- Fix typos in source code (Frédéric Chapoton).
- Add `flint-mparam.h` for Zen 4 (VN).
- Remove prototypes of non-existing functions `arb_poly_mulloz_ztrunc` (MM).
- Only include headers before `extern "C"` (AA).
- Undefine `small` macro in `qsieve.h` as it may cause clashes on Windows systems (AA).
- Fix `noreturn` attribute for C23, as it uses `[[noreturn]]` for this (Michael Orlitzky).
- Add newlines to end of files that were missing them (AA).
- Add `mpn_extras/inlines.c` (AA).

- Replace compound literal with a struct to accommodate MSVC (TD).
- Cast pointers in `machine_vectors.h` to suppress warnings about incompatible pointers (AA).
- Make GCC pragmas dispatch only for GCC (AA).
- Comment out unused static functions (AA).
- Check availability of `rbit` instruction for ARM in `n_revbin` (AA).
- Continuous integration
 - Add nightly build to Github (EC).
 - Add SHAsum (EC).
 - Remove Ubuntu CMake runner as we no longer recommend CMake for building FLINT on non-Windows systems (AA).
 - Add runner that checks against regression when compiling with `-Wextra -Werror` (AA).
 - Publish prepreleases only as drafts (Mahrud Sayrafi).
 - Multiple updates of the runners according to Github's runners (AA, EC).
 - Exclude profiler source code from Codecov (AA).
 - Assume `lcov` version 2.1 is available (AA).
 - Touchups and fixes for the release CI (EC).
 - Add Linux ARM runner (AA).
 - Fix release CI to automatically make releases when new tags are created, and push related files and generate website accordingly (AA, EC).
 - Rebuild citation block on website after each release (AA).
- Documentation
 - Fix typos (BA, Frédéric Chapoton, EC, MM, VN).
 - Clarify usage of inline assembly addition and subtraction macros such as `add_ssaaaa` (AA).
 - Add some explanations for `mpn_ctx_mpn_mul` (FJ).
 - Add human-readable text to documentation of `ordering_t` (RB).
 - Document Generic Ring setters for infinities and extended values (RB).
 - Fix documentation of `gr_cmp_other` (MM).
 - Fix reference to Granlund-Montgomery division (AA).
 - Correct names of the SFB-TRR 195 grant by the DFG (MH).
 - Correct `fq_default_default` to `fq_default` (AA).
 - Clarify documentation for `fq_default` context initialisers (AA).
 - Revise some `fmpz` docstrings (AA).
 - Document `MPZ_MIN_ALLOC` and low-level `fmpz` memory management functions (AA).
 - Correct docstring for `udiv_qrnd` (VN).
 - Rework docstring for `n_mulmod_precomp_shoup`, and clarify its constraints (VN).
 - Document `nmod_mat_set_mod` (VN).
 - Document `fmpz_mod_poly-ulong` scalar functions (AA).
 - Clarify that releases do not require Autotools (AA).
 - Fix signatures for `acb_mat_get_(real|imag)` (Marek Kaluba).

2024-03-18 – FLINT 3.1.2

Main contributors: Albin Ahlbäck (AA).

- Maintenance
 - Remove the need for `ldconfig` completely in FLINT's Autotools build system (AA).

2024-03-07 – FLINT 3.1.1

Main contributors: Albin Ahlbäck (AA).

- Bug fixes
 - Add `padic_types.h` to headers in `Makefile.in` (AA).
- Maintenance
 - Add check for `aligned_alloc` and `_aligned_malloc` for systems that may not provide any of these functions (AA).
 - Add options for setting `-march=ARCH` (AA).

2024-02-25 – FLINT 3.1.0

Main contributors: Fredrik Johansson (FJ), Albin Ahlbäck (AA), Jean Kieffer (JK).

- License
 - Changed license from “LGPL 2.1 or later” to “LGPL 3 or later”.
- Major interface changes
 - The methods in the `mpz_mod_mat` module now use a context object (FJ).
 - Changed `fz_mat_rref` and others to allow separate input and output matrices (FJ).
- Features
 - New module `acb_theta` for computing complex Riemann theta functions with characteristics in any dimension (JK).
 - `flint_printf` and related functions now supports printing common FLINT types, e.g. using the format string `%{mpz}` for `mpz_t` (AA).
 - Generic expression parsing: `gr_set_str` supports parsing expressions like `a*x^2+1/3` or `(0.1 + 0.2*i) +/- (0.001 + 0.001*i)` in any ring (FJ).
 - Added `mpz_mpoly_q_set_str_pretty` and `mpz_mpoly_q_get_str_pretty`.
 - Added `flint_mpn_mulhigh` (AA).
 - Primality testing for Gaussian integers (`mpz_i_is_prime`, `mpz_i_is_probabprime`) (Mathieu Gouttenoire).
 - Modular splitting evaluation of polynomials (`_gr_poly_evaluate_modular`) (David Berghaus).
 - Reversion of generic power series (`gr_poly_revert_series` and variants) (FJ).
 - Support inversion for `gr` vectors (FJ).
 - Split generic division into `gr_div` and `gr_div_nonunique` to make the semantics of division more precise (FJ).
 - Added `gr_ctx_is_zero_ring` (FJ).
 - Added `nmod_divides` (FJ).
 - Reciprocal Fibonacci constant (`arb_const_reciprocal_fibonacci`) (FJ).

-
- Added functions for working with symmetric positive-definite matrices (`fmpz_mat_is_spd`, `arb_mat_spd_get_fmpz_mat`, `arb_mat_spd_is_lll_reduced`, `arb_mat_spd_lll_reduce`, `arb_mat_randtest_cho`, `arb_mat_randtest_spd`) (JK).
 - Added several helper functions for `arb` and `acb` vectors and matrices (`_arb_vec_contains`, `_arb_vec_equal`, `_arb_vec_overlaps`, `_arb_vec_printd`, `_acb_vec_contains`, `_acb_vec_equal`, `_acb_vec_get_imag`, `_acb_vec_get_real`, `_acb_vec_overlaps`, `_acb_vec_printd`, `_acb_vec_set_real_imag`, `_acb_vec_sqr`, `arb_mat_vector_mul_col`, `arb_mat_vector_mul_row`, `acb_mat_vector_mul_col`, `acb_mat_vector_mul_row`, `acb_mat_get_imag`, `acb_mat_get_real`, `acb_mat_onei`) (JK).
 - Added `acb_urandom` and `arb_randtest_positive` (JK).
 - Added `acb_mul_i_pow_si` (JK).
 - Handle modulus 1 in `fmpz_CRT` functions (Fabian Gundlach).
 - Added `_nmod_poly_conway` and `_nmod_poly_conway_rand` (AA).
 - Added `fft_small_mulmod_satisfies_bounds`. The function `sd_fft_ctx_init` now verifies that the modulus satisfies the assumptions for correct modular arithmetic. (Daniel Schultz).
 - Allow setting generator names for `gr_mpoly`, `gr_series`, `fmpz_mpoly` and `fmpz_mpoly_q` generic rings (FJ).
 - Added `gr_gens_recursive` (FJ).
 - Allow overriding `flint_aligned_alloc` (AA).
 - Added `_fmpz_vec_dot_general` (FJ).
 - Allow setting degree and bit size evaluation limits for the `gr qqbar` context (FJ).
 - Added `nmod_poly_divexact` and use this instead of `nmod_poly_div` where an exact division is intended to improve performance (FJ).
 - Added `fmpz_poly_divexact` and use this instead of `fmpz_poly_div` where an exact division is intended to improve performance (FJ).
 - Added `fq_default_ctx_inner` to access the internal context object of a `fq_default_ctx` (FJ).
 - Implemented `gr_is_ring` and `gr_ctx_is_commutative_ring` properly (FJ).
 - Added `d_mul_2exp_inrange`, `d_mul_2exp_inrange2`, `d_mul_2exp` and `_d_vec_mul_2exp` (FJ).
 - Added `fmpz_mod_mat_det` (FJ).
 - Allow overriding `gr_mat_lu` (FJ).
- Bugs
 - Fixed threading problem in `gr_method_tab_init`: FLINT would occasionally crash when calling generics-based internal code when using a large number of threads (FJ, after debugging by Alexander Smirnov).
 - Fixed comparison of `gr` vectors with `fmpz` elements (FJ).
 - Fixed allocation bug in `gr_mpoly_mul_monomial` (FJ).
 - Fixed aliasing in `fmpz_i_divrem_approx` (FJ).
 - Avoid division by zero in `acb_poly_refine_roots_durand_kerner`: in rare instances, computing roots of an integer polynomial could hang (FJ).
 - Allow large arguments in `arb_atan_frac_bsplitt` (FJ).
 - Fixed printing large coefficients in `nmod_mpoly` (Alexander Smirnov, AA).
 - Fixed `fq*_poly_powmod` (AA).

- Fixed initialization of `fq_default_ctx` (Claus Fieker, Tommy Hofmann).
- Fixed memory leak in `gr_poly_write` (FJ).
- Fixed printing `nmod32` elements on 32-bit systems (FJ).
- Fixed `ldconfig` for BSD systems (AA).
- Fixed `FLINT_WANT_ASSERT` for CMake (AA).
- Fixed memory leak in `arb_nint` (FJ).
- Performance
 - FLINT is now built with `-O3 -march=native` by default (AA).
 - FLINT is no longer built with `-funroll-loops` by default except for select modules. This reduces the library size by more than 20%. (AA).
 - Assembly routines are now used as intended on ARM64 when compiling with GCC (AA).
 - New basecase code for `flint_mpn_mul`, `flint_mpn_mul_n` and `flint_mpn_sqr` (generic C versions, assembly for x86-64). This can yield up to a 2x speedup over GMP for short integer multiplications when calling `mpn` functions directly, though few applications currently benefit significantly due to wrapper overheads (some Arb benchmarks run ~5% faster with this change) (AA, FJ).
 - Added hardcoded low-level routines for high multiplications of two operands of the same size for Broadwell-type CPUs – `flint_mpn_mulhigh_*` and `flint_mpn_sqrhigh_*` (AA).
 - Added hardcoded low-level routines for high normalised multiplications of two normalised operands of the same size for Broadwell-type CPUs – `flint_mpn_mulhigh_normalised_*` (AA).
 - Added `flint_mpn_mulhigh_basecase` and `flint_mpn_sqrhigh_basecase` for Broadwell-type CPUs (AA).
 - Use `toom22` on top of custom basecase code for intermediate operands in `flint_mpn_mul` (FJ, based on GMP code).
 - Use `mulx` in `umul_ppmm` when available (AA).
 - Faster `_fmpz_vec_dot` (FJ).
 - Faster `_fmpz_mod_vec_dot` (FJ).
 - Faster `fmpz_poly` and `fmpz_mat` basecase algorithms based on dot products (FJ).
 - Optimized `fmpz_mat_mul_classical` (FJ).
 - Added `fmpz_mat_mul_waksman`, speeding up `fmpz_mat` multiplication for balanced matrices with huge entries (Éric Schost, Vincent Neiger, FJ).
 - Tweaks to improve GCC’s code generation for `fmpz_mpoly` and `nmod_mpoly` multiplication (FJ).
 - Improved `fmpz_mod_mat_set_fmpz_mat` (FJ).
 - Improved tuning for `fmpz_mat_sqr` (FJ).
 - Improved `fmpz_mat_sqr_bodrato` with small coefficients (Marco Bodrato).
 - Squaring optimizations in `fmpz_mat_mul_multi_mod` (FJ).
 - Use Bodrato’s sequence for Strassen multiplication (Marco Bodrato).
 - Strip trailing zeros in `fmpz_poly_gcd`: this gives a 50x speedup computing `gcd(x1000, x1001)` (FJ).
 - Faster CLD bound computation, speeding up `fmpz_poly_factor` for some polynomials (FJ).
 - Use new formulas from Jorge Zuniga and Jesús Guillera to compute `log(2)`, Catalan’s constant, `zeta(3)` and `gamma(1/3)` faster (FJ).

- Compressed the database of Conway polynomials to 10% of the original size (AA).
- Optimized context initializers for `fq`, `fq_zech`, `fq_nmod` and `qadic` (AA).
- Avoid calls to the slow standard library function `ldexp` (FJ).
- Improved `fmpz_is_probabprime` for word-size input (FJ).
- Make `gr_poly_resultant` use the quasilinear hgcd algorithm by default over finite rings and remove obsolete `fmpz_mod_poly` implementations (FJ).
- Optimized header files (AA).
- Changed several internal helper functions to forced inlines (AA).
- Merged some sources files to speed up compilation (AA).
- Faster computation of Swinnerton-Dyer polynomials (FJ).
- Added benchmark script (`dev/bench.py`) that should be used to check performance increases and regressions (FJ).
- Add special cases for Clang and MSVC in `longlong.h` (AA).
- Test code
 - Unified test programs per module: compiling FLINT’s test suite is now an order of magnitude faster (AA).
 - Added pretty-printing and timing output for unit tests (AA).
 - Improve use of test multiplier in some long-running unit tests (AA, FJ).
 - Improved test coverage (AA, FJ).
 - Allow `gr_ctx_init_random` to generate composite rings (FJ).
 - Fixed the test code of `fq*_poly_powmod` (AA).
- Maintenance
 - Require GMP $\geq 6.2.1$ and MPFR $\geq 4.1.0$ (AA).
 - Drop support for Itanium (AA).
 - Drop support for MPIR (AA).
 - Cleaned up `longlong.h` (AA).
 - Removed `ARB_VERSION`, `ANTIC_VERSION`, `CALCIUM_VERSION` and associated constants (AA).
 - Removed `_long_vec_print` and `_perm_print` (use `flint_printf` instead) (AA).
 - Removed unused functions in the `d_mat` and `aprc1` modules (AA).
 - Removed `fmpq_get_mpz_frac` (AA).
 - Removed `arb_fmpz_poly_cos_minpoly` (AA).
 - Removed `fmpz_mat_mul_classical_inline` (FJ).
 - Removed `fmpz_mat_rref_mod` (use `fmpz_mod_mat_rref` instead) (FJ).
 - Introduce `FLINT_SWAP` macro to replace several older macros (FJ).
 - Replaced `invert_limb` by `n_preinvert_limb_prenorm` (AA).
 - Renamed `_perm_set_one` to `_perm_one` (AA).
 - Only define some multithreaded “divides” function when the CPU is strongly ordered (AA).
 - Enable `fft_small` for MSVC builds (AA).
 - Use binary format instead of decimal format in `qsieve`, removing the need of conversions between said formats (AA).

- Use C11 atomics in the `fmpz` memory manager (AA).
- Merged some repeated code in the `mpoly` modules (FJ).
- Refactored `fq_default` to use `gr` generics internally (FJ).
- Replaced more functions by generics-based versions (FJ).
- Do not include `pthread.h` when opted out (AA).
- Test ARM NEON in CI via Github’s Apple M1 runner (AA).
- Test examples in CI (AA).
- Detect MPFR and GMP internals in `configure` (AA).
- Add `-lflint` to PKG-CONFIG (Josh Rickmar).
- Silence GCC compiler warnings (AA).
- Unified exception handling (AA).
- Corrected some function signatures in the documentation (Vincent Delecroix, Joel-Dahne, Edgar Costa).
- Rename the default Git-branch from `trunk` to `main` (FJ).
- Document some macros defined in `flint.h` (AA).
- Other code cleanup and modernisation (AA).
- Major cleanup in `configure.ac` and `acinclude.m4`.
- Use parts of GMP’s configuration to configure assembly properly.
- Change instances of `#ifdef FLINT_WANT_ASSERT` to `#if FLINT_WANT_ASSERT` (AA).

2023-11-10 – FLINT 3.0.1

- Build issues
 - Fix LIBS2 order for static linking (Tomás Oliveira e Silva).
 - Fix substitution of version number for older autotools (Albin Ahlbäck).
 - Fix use of `AC_SEARCH_LIBS` to find `cblas_dgemm` (Gonzalo Tornarí).
 - Add FlexiBLAS as a `cblas` option (Mahrud Sayrafi).
 - Don’t use deprecated `PythonInterp` in CMake build (Mahrud Sayrafi).
 - Fix setting version numbers and strings in CMake build (Mahrud Sayrafi).
 - Only link with NTL for the tests on CMake (Mahrud Sayrafi).
- Bugs
 - Fix bug in `nmod32` on 32-bit systems.
 - Fix missing modulus assignment in `nmod_poly_mat_window_init` (Vincent Neiger).
 - Fix tmp allocation size in `_fmpz_set_str_basecase`.
 - Fix rare arithmetic bug and memory leak in `n_factor_ecm_select_curve`.
- Other
 - Some corrections to the documentation.

2023-10-20 – FLINT 3.0.0

Merged libraries and reorganisation

- The following libraries have been merged into FLINT:
 - Arb 2.23 (arbitrary-precision ball arithmetic)
 - Calcium 0.4 (exact real and complex arithmetic)
 - Antic 0.2.5 (number fields, binary quadratic forms)
- Arb, Calcium and Antic will no longer be maintained as separate libraries. Users upgrading to FLINT 3.0 should ensure that they no longer link to the old Arb, Calcium or Antic library files or include any header files from those libraries which may be incompatible.
- The FLINT 3.0 API is largely backwards-compatible with FLINT 2.9, Arb 2.23, Calcium 0.4 and Antic 0.2.5, except for changes to rarely-used and internal functions documented below. However, the following changes to the handling of header files are likely to require (trivial) patches in many downstream codebases:
 - Header files belonging to Arb, Calcium and Antic now appear in the `flint/` subdirectory. For example, instead of `#include "arb.h"`, it is necessary to `#include "flint/arb.h"` unless `<INCLUDE_DIR>/flint` has been added to the include path.
 - Most header files no longer include their implicit dependencies. For example, `fmpz_poly.h` no longer includes `fmpz.h`. Code that used functions from the `fmpz` module but only included `fmpz_poly.h` may thus now need to include `fmpz.h` explicitly. Likewise, many inclusions of system libraries like `stdlib.h` have been removed.
- The following people helped with the merge: Fredrik Johansson, Isuru Fernando, Albin Ahlbäck.
- FLINT 3.0 has a new build system based on Autotools, contributed by Albin Ahlbäck. Among other improvements, parallel builds are much faster and it is possible to build individual targets. Additional build system and CI improvements have been made by Marc Mezzarobba, Max Horn, Edgar Costa, Alex Best, Andreas Enge, and others.
- It is now necessary to run `bootstrap.sh` to generate the `configure` script in order to build FLINT from the git repository.
- Some `configure` options have changed: for example, `--reentrant` is now `--enable-reentrant`.
- The root directory has been cleaned up by moving all source code into the `src` directory. This should not affect any users.
- The NTL interface has been moved to a single header file. The `--with-ntl` build flag is now only needed to build the test code for this interface.
- The C++ interface (`flintxx`) has been removed. This interface is now maintained in the separate repository <https://github.com/flintlib/flintxx> (Edgar Costa).

Generic rings

- The new `gr` module supports generic programming. It provides wrappers for most builtin FLINT types and allows constructing generic structures (polynomials, matrices, etc.) over arbitrary base rings. The following modules are available:
 - `gr_generic` (various generic algorithms)
 - `gr_mat` (matrices with generic elements)
 - `gr_mpoly` (multivariate polynomials with generic elements)
 - `gr_poly` (univariate polynomials with generic elements)
 - `gr_special` (special functions for generic elements)

- `gr_vec` (vectors with generic elements)

This feature is experimental: it is highly likely that some interfaces will change in a future FLINT release.

- There is also a Python wrapper (`flint_ctypes`) included with FLINT available in the `src/python` directory. Unlike other third-party FLINT wrappers available currently, this wrapper uses the `gr` interface to wrap (nearly) all FLINT types at once. This wrapper is not officially supported and will likely be deprecated in the future, but it can be useful for experimenting with FLINT.
- The generics system supports certain representations that do not have dedicated FLINT modules, for example 8-bit and 32-bit `nmods`.

Small-prime FFT

- The new `fft_small` module implements FFTs modulo word-size primes and multiplication based on such FFTs. This module requires AVX2 or NEON vector instructions and will not be built on targets that do not support them. The small-prime FFT speeds up the following functions for huge input, sometimes by a factor 2x to 10x:
 - `flint_mpn_mul` and variants, and indirectly any function based on FLINT's integer multiplication for large inputs. For example, `fmpz_mul` and `arb_mul` are faster, but `fmpz_gcd` is currently unaffected since it calls GMP.
 - `nmod_poly_mul` and variants, and indirectly any function based on `nmod_poly` multiplication.
 - `fmpz_poly_mul` and variants, and indirectly any function based on `fmpz_poly` multiplication.
 - Division functions for `fmpz` and `arb`, which now use Newton iteration instead of calling GMP for huge input.
 - `fmpz_mod` arithmetic.
 - Radix conversion functions like `fmpz_get_str`, `fmpz_set_str` and `arb_get_str`.
- The FFT was contributed by Daniel Schultz, with final integration work and adaptations for other FLINT functions (Newton iteration implementations, etc.) done by Fredrik Johansson.

Other changes

- Changed the order of the `alloc` and `length` fields in `arb_poly_t`, `acb_poly_t` and `ca_poly_t` to match the FLINT types.
- Added `fmpz_i` division, norm and GCD functions (`gcd_shortest` by Daniel Schultz).
- Added an `acf` type for complex floating-point numbers.
- Added error handling to `dirichlet_group_init`.
- Increased the prime factor limit in `dirichlet_group_init` from `1e12` to `1e16`.
- Added `arb_nonnegative_abs` (Erik Postma).
- Fixed `arb_pow` for `x` just barely containing 0, `y > 0` (Erik Postma).
- Improved precision handling in `arb_gamma` for huge input.
- Faster `arb_contains_arf`, `arb_overlaps`, `arb_gt`, `arb_lt`.
- Changed the argument order of `_fmpz_mod_poly_mullo` and `_fmpz_mod_poly_div_series`.
- Changed the call signature of many `_fmpz_mod_poly` methods to take a context object as input instead of the raw modulus.
- Support test coverage reports (`--enable-coverage`).
- Added `fmpz_poly_randtest_irreducible`.

- Improved tuning for various `nmod_poly` functions.
- Most Newton polynomial division and square root functions now use the Karp-Markstein algorithm.
- Replaced `count_leading_zeros` and `count_trailing_zeros` macros with `flint_clz` and `flint_ctz`.
- Fixed `nmod_poly_compose` which was not using an asymptotically fast algorithm.
- Various functions in the `nmod`, `fmpz_mod`, `fq` modules and elsewhere have been rewritten to use algorithms in the `generics` module. In many cases the corresponding type-specific algorithm implementation has been removed entirely (for example, `nmod_poly_divrem_newton` no longer exists).
- Fixed `fmpz_mod_poly_factor_squarefree`, `nmod_poly_factor_squarefree` and `fq*_poly_factor_squarefree` sometimes returning non-monic factors. Among other consequences, this could lead to functions like `fq_poly_roots` returning incorrect roots
- Fixed several bugs in the `fq_default` modules (Tommy Hofmann).
- Fixed stack overflow in `mpoly_divrem_ideal` functions.
- Handle the the zero polynomial correctly in `nmod_poly_shift_left` (Vincent Neiger).
- Fixed handling of permutations in `invert_cols` matrix methods (Vincent Neiger).
- Added `nmod_mat_permute_rows` (Vincent Neiger).
- Fixed bug in `mpoly_monomial_halves` (Daniel Schultz).
- Fixed overflow bug in `fmpz_mod_mpoly_divrem_ideal` (Daniel Schultz).
- Optimized `fmpz_addmul`, `fmpz_addmul_ui`, `fmpz_submul`, `fmpz_submul_ui` for small arguments.
- Fixed demotion bug in `fmpz_addmul_si` and `fmpz_submul_si`.
- Optimized `fmpq_cmp`, `fmpq_cmp_ui`, `fmpq_cmp_si`, `fmpq_cmp_fmpz` for small arguments.
- Optimized `fmpz_poly_resultant_modular` by using a tighter bound.
- Allow `lll` to work with rank deficient `Z` basis (Daniel Schultz).
- Added `fmpq_mat_can_solve_dixon` (William Hart).
- Inlined `n_gcd` (Albin Ahlbäck).
- Fixed fallback code for `sub_ddmmss` when given signed arguments.
- Many documentation fixes (Håvard Damm-Johnsen, Joel Dahne, Albin Ahlbäck, David Einstein, Alex Best, and others).
- Code simplifications (Vincent Neiger).
- Fixed several type signatures (Ricardo Buring).
- Fixed several memory leaks (Ricardo Buring).
- Fixed `fmpz_poly_factor_squarefree` crashing when given the zero polynomial.
- Added `arb_minmax` (Joel Dahne).
- Added `_push_term_ffmpz` functions to `mpoly` types (David Einstein).
- Added functions for printing `nmod` vectors (Vincent Neiger).
- Added `nmod_poly_is_monic` (Vincent Neiger).
- Fixed threaded `Arb` functions to use the thread pool (Albin Ahlbäck).
- Removed `nmod_poly_mpz` functions (Ricardo Buring).
- Fixed file handling in `qsieve` (Michiel de Wilde, Oscar Benjamin).
- Free memory in case of failure in `fq_zech_ctx_init` (Claus Fieker).
- Fixed corrupted output in `fmpz_or`.

- Added several `nmod_poly_mat` utility functions (Vincent Neiger).

List of additions

- FLINT 3.0 includes all functions in FLINT 2.9, Arb 2.23, Calcium 0.4 and Antic 0.2.5 except those listed under “list of removals”. On top of this, the following functions have been added. This list is incomplete; many internal functions and functions starting with an underscore have been omitted.
- `mpn_mul_default_mpn_ctx`, `_nmod_poly_mul_mid_default_mpn_ctx`, `_fmpz_poly_mul_mid_default_mpn_ctx` and many internal functions in the new `fft_small` module
- `acb_poly_nth_derivative`, `arb_div_arf_newton`, `arb_div_newton`, `arb_fmpz_divapprox`, `arb_nint`, `arb_poly_nth_derivative`, `arb_rsqr_arf`, `arb_rsqr_arf_newton`, `arb_sqrt_arf_newton`, `arb_sqrt_newton`, `arb_trunc`, `arb_minmax`
- `ca_set_fmpz_i`
- `flint_aligned_alloc`, `flint_aligned_free`
- `flint_get_num_available_threads`
- `flint_mpn_add_inplace_c`, `flint_mpn_cmp_ui_2exp`, `flint_mpn_mul_large`, `flint_mpn_nbits`
- `fmpz_get_str_bsplitted_threaded`
- `fmpz_mat_equal_col`, `fmpz_mat_equal_row`, `fmpz_neg_ui_array`
- `fmpz_poly_randtest_irreducible`
- `fmpz_poly_q_evaluate_fmpz`, `fmpz_poly_q_scalar_div_fmpz`, `fmpz_poly_q_scalar_div_fmpz`, `fmpz_poly_q_scalar_mul_fmpz`, `fmpz_poly_q_scalar_mul_fmpz`
- `fmpz_ui_pow_ui`
- `fmpz_i_set_qqbar`
- `get_default_mpn_ctx`
- `gr_abs`, `gr_acos`, `gr_acos_pi`, `gr_acosh`, `gr_acot`, `gr_acot_pi`, `gr_acoth`, `gr_acsc`, `gr_acsc_pi`, `gr_acsch`, `gr_add`, `gr_add_fmpz`, `gr_add_fmpz`, `gr_add_other`, `gr_add_si`, `gr_add_ui`, `gr_addmul`, `gr_addmul_fmpz`, `gr_addmul_fmpz`, `gr_addmul_other`, `gr_addmul_si`, `gr_addmul_ui`, `gr_agm`, `gr_agm1`, `gr_airy`, `gr_airy_ai`, `gr_airy_ai_prime`, `gr_airy_ai_prime_zero`, `gr_airy_ai_zero`, `gr_airy_bi`, `gr_airy_bi_prime`, `gr_airy_bi_prime_zero`, `gr_airy_bi_zero`, `gr_asec`, `gr_asec_pi`, `gr_asech`, `gr_asin`, `gr_asin_pi`, `gr_asinh`, `gr_atan`, `gr_atan2`, `gr_atan_pi`, `gr_atanh`, `gr_barnes_g`, `gr_bellnum_fmpz`, `gr_bellnum_ui`, `gr_bellnum_vec`, `gr_bernoulli_fmpz`, `gr_bernoulli_ui`, `gr_bernoulli_vec`, `gr_bernopoly_ui`, `gr_bessel_i`, `gr_bessel_i_scaled`, `gr_bessel_j`, `gr_bessel_j_y`, `gr_bessel_k`, `gr_bessel_k_scaled`, `gr_bessel_y`, `gr_beta`, `gr_beta_lower`, `gr_bin`, `gr_bin_ui`, `gr_bin_ui_vec`, `gr_bin_uiui`, `gr_bin_vec`, `gr_carlson_rc`, `gr_carlson_rd`, `gr_carlson_rf`, `gr_carlson_rg`, `gr_carlson_rj`, `gr_catalan`, `gr_ceil`, `gr_chebyshev_t`, `gr_chebyshev_t_fmpz`, `gr_chebyshev_u`, `gr_chebyshev_u_fmpz`, `gr_clear`, `gr_cmp`, `gr_cmp_other`, `gr_cmpabs`, `gr_cmpabs_other`, `gr_conj`, `gr_cos`, `gr_cos_integral`, `gr_cos_pi`, `gr_cosh`, `gr_cosh_integral`, `gr_cot`, `gr_cot_pi`, `gr_coth`, `gr_coulomb`, `gr_coulomb_f`, `gr_coulomb_g`, `gr_coulomb_hneg`, `gr_coulomb_hpos`, `gr_csc`, `gr_csc_pi`, `gr_csch`, `gr_csgn`, `gr_ctx_ca_get_option`, `gr_ctx_ca_set_option`, `gr_ctx_clear`, `gr_ctx_cmp_coercion`, `gr_ctx_data_as_ptr`, `gr_ctx_data_ptr`, `gr_ctx_fmpz_mod_set_primality`, `gr_ctx_fq_degree`, `gr_ctx_fq_order`, `gr_ctx_fq_prime`, `gr_ctx_get_real_prec`, `gr_ctx_get_str`, `gr_ctx_has_real_prec`, `gr_ctx_init_complex_acb`, `gr_ctx_init_complex_algebraic_ca`, `gr_ctx_init_complex_ca`, `gr_ctx_init_complex_float_acf`, `gr_ctx_init_complex_qqbar`, `gr_ctx_init_dirichlet_group`, `gr_ctx_init_fmpz`, `gr_ctx_init_fmpz`

```

gr_ctx_init_fmpz_mod, gr_ctx_init_fmpz_poly, gr_ctx_init_fmpz_i, gr_ctx_init_fq,
gr_ctx_init_fq_nmod, gr_ctx_init_fq_zech, gr_ctx_init_gr_series,
gr_ctx_init_gr_series_mod, gr_ctx_init_matrix_domain, gr_ctx_init_matrix_ring,
gr_ctx_init_matrix_space, gr_ctx_init_gr_mpoly, gr_ctx_init_nf,
gr_ctx_init_nf_fmpz_poly, gr_ctx_init_nmod, gr_ctx_init_nmod8,
gr_ctx_init_nmod32, gr_ctx_init_perm, gr_ctx_init_gr_poly, gr_ctx_init_psl2z,
gr_ctx_init_random, gr_ctx_init_real_algebraic_ca, gr_ctx_init_real_arb,
gr_ctx_init_real_ca, gr_ctx_init_real_float_arf, gr_ctx_init_real_qqbar,
gr_ctx_init_vector_gr_vec, gr_ctx_init_vector_space_gr_vec,
gr_ctx_is_algebraically_closed, gr_ctx_is_canonical, gr_ctx_is_commutative_ring,
gr_ctx_is_exact, gr_ctx_is_field, gr_ctx_is_finite, gr_ctx_is_finite_characteristic,
gr_ctx_is_integral_domain, gr_ctx_is_multiplicative_group,
gr_ctx_is_ordered_ring, gr_ctx_is_ring, gr_ctx_is_threadsafe,
gr_ctx_is_unique_factorization_domain, gr_ctx_matrix_is_fixed_size,
gr_ctx_print, gr_ctx_println, gr_ctx_set_real_prec, gr_ctx_sizeof_ctx,
gr_ctx_sizeof_elem, gr_ctx_vector_gr_vec_is_fixed_size, gr_ctx_write,
gr_dedekind_eta, gr_dedekind_eta_q, gr_digamma, gr_dilog, gr_dirichlet_beta,
gr_dirichlet_chi_fmpz, gr_dirichlet_chi_vec, gr_dirichlet_eta,
gr_dirichlet_hardy_theta, gr_dirichlet_hardy_z, gr_dirichlet_l, gr_div,
gr_div_fmpz, gr_div_fmpz, gr_div_other, gr_div_si, gr_div_ui, gr_divexact,
gr_divexact_fmpz, gr_divexact_fmpz, gr_divexact_other, gr_divexact_si,
gr_divexact_ui, gr_divides, gr_dot_other, gr_doublefac, gr_doublefac_ui,
gr_eisenstein_e, gr_eisenstein_g, gr_eisenstein_g_vec, gr_elliptic_e,
gr_elliptic_e_inc, gr_elliptic_f, gr_elliptic_invariants, gr_elliptic_k,
gr_elliptic_pi, gr_elliptic_pi_inc, gr_elliptic_roots, gr_equal, gr_erf,
gr_erfc, gr_erfcinv, gr_erfcx, gr_erfi, gr_erfinv, gr_euclidean_div,
gr_euclidean_divrem, gr_euclidean_rem, gr_euler, gr_eulernum_fmpz,
gr_eulernum_ui, gr_eulernum_vec, gr_eulerpoly_ui, gr_evaluate_fmpz_mpoly_iter,
gr_exp, gr_exp10, gr_exp2, gr_exp_integral, gr_exp_integral_ei, gr_exp_pi_i,
gr_expm1, gr_fac, gr_fac_fmpz, gr_fac_ui, gr_fac_vec, gr_factor, gr_falling,
gr_falling_ui, gr_fib_fmpz, gr_fib_ui, gr_fib_vec, gr_floor, gr_fmms,
gr_fmpz_mpoly_evaluate, gr_fmpz_mpoly_evaluate_horner, gr_fmpz_poly_evaluate,
gr_fmpz_poly_evaluate_horner, gr_fmpz_poly_evaluate_rectangular,
gr_fq_frobenius, gr_fq_is_primitive, gr_fq_multiplicative_order, gr_fq_norm,
gr_fq_pth_root, gr_fq_trace, gr_fresnel, gr_fresnel_c, gr_fresnel_s, gr_gamma,
gr_gamma_fmpz, gr_gamma_fmpz, gr_gamma_lower, gr_gamma_upper, gr_gcd,
gr_gegenbauer_c, gr_gen, gr_generic_acot, gr_generic_acoth, gr_generic_acsc,
gr_generic_acsch, gr_generic_add_fmpz, gr_generic_add_fmpz, gr_generic_add_other,
gr_generic_add_si, gr_generic_add_ui, gr_generic_addmul, gr_generic_addmul_fmpz,
gr_generic_addmul_fmpz, gr_generic_addmul_other, gr_generic_addmul_si,
gr_generic_addmul_ui, gr_generic_asec, gr_generic_asech, gr_generic_asin,
gr_generic_asinh, gr_generic_atan, gr_generic_atanh, gr_generic_bellnum_fmpz,
gr_generic_bellnum_ui, gr_generic_bellnum_vec, gr_generic_bernoulli_fmpz,
gr_generic_bernoulli_ui, gr_generic_bernoulli_vec, gr_generic_beta,
gr_generic_bin, gr_generic_bin_ui, gr_generic_bin_ui_vec, gr_generic_bin_uiui,
gr_generic_bin_vec, gr_generic_chebyshev_t2_fmpz, gr_generic_chebyshev_t_fmpz,
gr_generic_chebyshev_u2_fmpz, gr_generic_chebyshev_u_fmpz, gr_generic_cmp,
gr_generic_cmp_other, gr_generic_cmpabs, gr_generic_cmpabs_other,
gr_generic_cos, gr_generic_ctx_clear, gr_generic_ctx_predicate,
gr_generic_ctx_predicate_false, gr_generic_ctx_predicate_true,
gr_generic_div_fmpz, gr_generic_div_fmpz, gr_generic_div_other,
gr_generic_div_si, gr_generic_div_ui, gr_generic_divexact, gr_generic_doublefac,
gr_generic_doublefac_ui, gr_generic_erfcx, gr_generic_eulernum_fmpz,
gr_generic_eulernum_ui, gr_generic_eulernum_vec, gr_generic_exp,
gr_generic_exp10, gr_generic_exp2, gr_generic_expm1, gr_generic_fac,
gr_generic_fac_fmpz, gr_generic_fac_ui, gr_generic_fac_vec, gr_generic_falling,
gr_generic_falling_ui, gr_generic_fib2_fmpz, gr_generic_fib_fmpz,
gr_generic_fib_ui, gr_generic_fib_vec, gr_generic_get_fmpz_2exp_fmpz,

```

```
gr_generic_harmonic, gr_generic_harmonic_ui, gr_generic_hilbert_class_poly,
gr_generic_inv, gr_generic_is_invertible, gr_generic_is_neg_one,
gr_generic_is_one, gr_generic_is_square, gr_generic_is_zero,
gr_generic_log, gr_generic_log10, gr_generic_loglp, gr_generic_log2,
gr_generic_mul_2exp_fmpz, gr_generic_mul_2exp_si, gr_generic_mul_fmpq,
gr_generic_mul_fmpz, gr_generic_mul_other, gr_generic_mul_si, gr_generic_mul_two,
gr_generic_mul_ui, gr_generic_mul_ui_via_ZZ, gr_generic_neg_one,
gr_generic_other_add, gr_generic_other_add_vec, gr_generic_other_div,
gr_generic_other_div_vec, gr_generic_other_divexact_vec, gr_generic_other_mul,
gr_generic_other_mul_vec, gr_generic_other_pow, gr_generic_other_pow_vec,
gr_generic_other_sub, gr_generic_other_sub_vec, gr_generic_partitions_fmpz,
gr_generic_partitions_ui, gr_generic_partitions_vec, gr_generic_pow_fmpq,
gr_generic_pow_fmpz, gr_generic_pow_fmpz_binexp, gr_generic_pow_other,
gr_generic_pow_si, gr_generic_pow_ui, gr_generic_pow_ui_binexp,
gr_generic_randtest_not_zero, gr_generic_rfac, gr_generic_rfac_fmpz,
gr_generic_rfac_ui, gr_generic_rfac_vec, gr_generic_rising, gr_generic_rising_ui,
gr_generic_rsqr, gr_generic_scalar_add_vec, gr_generic_scalar_div_vec,
gr_generic_scalar_divexact_vec, gr_generic_scalar_mul_vec,
gr_generic_scalar_other_add_vec, gr_generic_scalar_other_div_vec,
gr_generic_scalar_other_divexact_vec, gr_generic_scalar_other_mul_vec,
gr_generic_scalar_other_pow_vec, gr_generic_scalar_other_sub_vec,
gr_generic_scalar_pow_vec, gr_generic_scalar_sub_vec, gr_generic_set_fmpq,
gr_generic_set_fmpz_2exp_fmpz, gr_generic_set_other, gr_generic_set_shallow,
gr_generic_sin, gr_generic_sin_cos, gr_generic_sqr, gr_generic_sqrt,
gr_generic_stirling_s1_ui_vec, gr_generic_stirling_s1_uiui,
gr_generic_stirling_s1u_ui_vec, gr_generic_stirling_s1u_uiui,
gr_generic_stirling_s2_ui_vec, gr_generic_stirling_s2_uiui, gr_generic_sub_fmpq,
gr_generic_sub_fmpz, gr_generic_sub_other, gr_generic_sub_si, gr_generic_sub_ui,
gr_generic_submul, gr_generic_submul_fmpq, gr_generic_submul_fmpz,
gr_generic_submul_other, gr_generic_submul_si, gr_generic_submul_ui,
gr_generic_tan, gr_generic_vec_add, gr_generic_vec_add_other,
gr_generic_vec_add_scalar, gr_generic_vec_add_scalar_fmpq,
gr_generic_vec_add_scalar_fmpz, gr_generic_vec_add_scalar_other,
gr_generic_vec_add_scalar_si, gr_generic_vec_add_scalar_ui,
gr_generic_vec_clear, gr_generic_vec_div, gr_generic_vec_div_other,
gr_generic_vec_div_scalar, gr_generic_vec_div_scalar_fmpq,
gr_generic_vec_div_scalar_fmpz, gr_generic_vec_div_scalar_other,
gr_generic_vec_div_scalar_si, gr_generic_vec_div_scalar_ui,
gr_generic_vec_divexact, gr_generic_vec_divexact_other, gr_generic_vec_divexact_scalar,
gr_generic_vec_divexact_scalar_fmpq, gr_generic_vec_divexact_scalar_fmpz,
gr_generic_vec_divexact_scalar_other, gr_generic_vec_divexact_scalar_si,
gr_generic_vec_divexact_scalar_ui, gr_generic_vec_dot, gr_generic_vec_dot_fmpz,
gr_generic_vec_dot_rev, gr_generic_vec_dot_si, gr_generic_vec_dot_ui,
gr_generic_vec_equal, gr_generic_vec_init, gr_generic_vec_is_zero,
gr_generic_vec_mul, gr_generic_vec_mul_other, gr_generic_vec_mul_scalar,
gr_generic_vec_mul_scalar_2exp_si, gr_generic_vec_mul_scalar_fmpq,
gr_generic_vec_mul_scalar_fmpz, gr_generic_vec_mul_scalar_other,
gr_generic_vec_mul_scalar_si, gr_generic_vec_mul_scalar_ui,
gr_generic_vec_neg, gr_generic_vec_normalise, gr_generic_vec_normalise_weak,
gr_generic_vec_pow, gr_generic_vec_pow_other, gr_generic_vec_pow_scalar,
gr_generic_vec_pow_scalar_fmpq, gr_generic_vec_pow_scalar_fmpz,
gr_generic_vec_pow_scalar_other, gr_generic_vec_pow_scalar_si,
gr_generic_vec_pow_scalar_ui, gr_generic_vec_reciprocals,
gr_generic_vec_scalar_addmul, gr_generic_vec_scalar_addmul_si,
gr_generic_vec_scalar_submul, gr_generic_vec_scalar_submul_si,
gr_generic_vec_set, gr_generic_vec_set_powers, gr_generic_vec_sub,
gr_generic_vec_sub_other, gr_generic_vec_sub_scalar, gr_generic_vec_sub_scalar_fmpq,
gr_generic_vec_sub_scalar_fmpz, gr_generic_vec_sub_scalar_other,
```

```

gr_generic_vec_sub_scalar_si, gr_generic_vec_sub_scalar_ui, gr_generic_vec_swap,
gr_generic_vec_zero, gr_generic_write_n, gr_get_d, gr_get_fmpq,
gr_get_fmpz, gr_get_fmpz_2exp_fmpz, gr_get_si, gr_get_str, gr_get_str_n,
gr_get_ui, gr_glaisher, gr_harmonic, gr_harmonic_ui, gr_heap_clear,
gr_heap_clear_vec, gr_heap_init, gr_heap_init_vec, gr_hermite_h,
gr_hilbert_class_poly, gr_hurwitz_zeta, gr_hypgeom_0f1, gr_hypgeom_1f1,
gr_hypgeom_2f1, gr_hypgeom_pfq, gr_hypgeom_u, gr_i, gr_im, gr_init, gr_inv,
gr_is_invertible, gr_is_neg_one, gr_is_one, gr_is_square, gr_is_zero,
gr_jacobi_p, gr_jacobi_theta, gr_jacobi_theta_1, gr_jacobi_theta_2,
gr_jacobi_theta_3, gr_jacobi_theta_4, gr_khinchin, gr_laguerre_l,
gr_lambertw, gr_lambertw_fmpz, gr_lcm, gr_legendre_p, gr_legendre_p_root_ui,
gr_legendre_q, gr_lerch_phi, gr_lgamma, gr_log, gr_log10, gr_loglp, gr_log2,
gr_log_barnes_g, gr_log_integral, gr_log_pi_i, gr_mat_add, gr_mat_add_scalar,
gr_mat_addmul_scalar, gr_mat_adjugate, gr_mat_adjugate_charpoly,
gr_mat_adjugate_cofactor, gr_mat_apply_row_similarity, gr_mat_charpoly,
gr_mat_charpoly_berkowitz, gr_mat_charpoly_danilevsky, gr_mat_charpoly_faddeev,
gr_mat_charpoly_faddeev_bsgs, gr_mat_charpoly_from_hessenberg,
gr_mat_charpoly_gauss, gr_mat_charpoly_householder, gr_mat_clear,
gr_mat_concat_horizontal, gr_mat_concat_vertical, gr_mat_det,
gr_mat_det_berkowitz, gr_mat_det_cofactor, gr_mat_det_fflu, gr_mat_det_generic,
gr_mat_det_generic_field, gr_mat_det_generic_integral_domain, gr_mat_det_lu,
gr_mat_diag_mul, gr_mat_diagonalization, gr_mat_diagonalization_generic,
gr_mat_diagonalization_precomp, gr_mat_div_scalar, gr_mat_eigenvalues,
gr_mat_eigenvalues_other, gr_mat_entry_ptr, gr_mat_entry_srcptr, gr_mat_equal,
gr_mat_exp, gr_mat_exp_jordan, gr_mat_fflu, gr_mat_find_nonzero_pivot,
gr_mat_find_nonzero_pivot_generic, gr_mat_find_nonzero_pivot_large_abs,
gr_mat_gr_poly_evaluate, gr_mat_hadamard, gr_mat_hessenberg,
gr_mat_hessenberg_gauss, gr_mat_hessenberg_householder, gr_mat_hilbert,
gr_mat_init, gr_mat_init_set, gr_mat_inv, gr_mat_invert_cols,
gr_mat_invert_rows, gr_mat_is_diagonal, gr_mat_is_empty, gr_mat_is_hessenberg,
gr_mat_is_lower_triangular, gr_mat_is_neg_one, gr_mat_is_one, gr_mat_is_scalar,
gr_mat_is_square, gr_mat_is_upper_triangular, gr_mat_is_zero,
gr_mat_jordan_blocks, gr_mat_jordan_form, gr_mat_jordan_transformation,
gr_mat_log, gr_mat_log_jordan, gr_mat_lu, gr_mat_lu_classical,
gr_mat_lu_recursive, gr_mat_minpoly_field, gr_mat_mul, gr_mat_mul_classical,
gr_mat_mul_diag, gr_mat_mul_generic, gr_mat_mul_scalar, gr_mat_mul_strassen,
gr_mat_neg, gr_mat_nonsingular_solve, gr_mat_nonsingular_solve_den,
gr_mat_nonsingular_solve_den_fflu, gr_mat_nonsingular_solve_fflu,
gr_mat_nonsingular_solve_fflu_precomp, gr_mat_nonsingular_solve_lu,
gr_mat_nonsingular_solve_lu_precomp, gr_mat_nonsingular_solve_tril,
gr_mat_nonsingular_solve_tril_classical, gr_mat_nonsingular_solve_tril_recursive,
gr_mat_nonsingular_solve_triu, gr_mat_nonsingular_solve_triu_classical,
gr_mat_nonsingular_solve_triu_recursive, gr_mat_nullspace, gr_mat_one,
gr_mat_ones, gr_mat_pascal, gr_mat_print, gr_mat_randops, gr_mat_randpermdiag,
gr_mat_ranrank, gr_mat_randtest, gr_mat_rank, gr_mat_rank_fflu, gr_mat_rank_lu,
gr_mat_reduce_row, gr_mat_rref, gr_mat_rref_den, gr_mat_rref_den_fflu,
gr_mat_rref_fflu, gr_mat_rref_lu, gr_mat_set, gr_mat_set_fmpq, gr_mat_set_fmpq_mat,
gr_mat_set_fmpz, gr_mat_set_fmpz_mat, gr_mat_set_jordan_blocks, gr_mat_set_scalar,
gr_mat_set_si, gr_mat_set_ui, gr_mat_solve_field, gr_mat_sqr, gr_mat_stirling,
gr_mat_sub, gr_mat_sub_scalar, gr_mat_submul_scalar, gr_mat_swap, gr_mat_swap_cols,
gr_mat_swap_entrywise, gr_mat_swap_rows, gr_mat_trace, gr_mat_trace_prod2,
gr_mat_transpose, gr_mat_transpose_resize, gr_mat_window_clear, gr_mat_window_init,
gr_mat_write, gr_mat_zero, gr_method_tab_init, gr_modular_delta, gr_modular_j,
gr_modular_lambda, gr_mpoly_add, gr_mpoly_assert_canonical, gr_mpoly_clear,
gr_mpoly_combine_like_terms, gr_mpoly_equal, gr_mpoly_fit_bits, gr_mpoly_fit_length,
gr_mpoly_fit_length_fit_bits, gr_mpoly_fit_length_reset_bits, gr_mpoly_gen,
gr_mpoly_get_coeff_scalar_fmpz, gr_mpoly_get_coeff_scalar_ui, gr_mpoly_init,
gr_mpoly_init2, gr_mpoly_init3, gr_mpoly_is_canonical, gr_mpoly_is_gen,

```

gr_mpoly_is_one, gr_mpoly_is_zero, gr_mpoly_mul, gr_mpoly_mul_fmpq,
gr_mpoly_mul_fmpz, gr_mpoly_mul_johnson, gr_mpoly_mul_monomial, gr_mpoly_mul_scalar,
gr_mpoly_mul_si, gr_mpoly_mul_ui, gr_mpoly_neg, gr_mpoly_one, gr_mpoly_print_pretty,
gr_mpoly_push_term_scalar_fmpz, gr_mpoly_push_term_scalar_ui, gr_mpoly_randtest_bits,
gr_mpoly_randtest_bound, gr_mpoly_set, gr_mpoly_set_coeff_fmpq_fmpz,
gr_mpoly_set_coeff_fmpq_ui, gr_mpoly_set_coeff_fmpz_fmpz, gr_mpoly_set_coeff_fmpz_ui,
gr_mpoly_set_coeff_scalar_fmpz, gr_mpoly_set_coeff_scalar_ui, gr_mpoly_set_coeff_si_fmpz,
gr_mpoly_set_coeff_si_ui, gr_mpoly_set_coeff_ui_fmpz, gr_mpoly_set_coeff_ui_ui,
gr_mpoly_set_fmpq, gr_mpoly_set_fmpz, gr_mpoly_set_scalar, gr_mpoly_set_si,
gr_mpoly_set_ui, gr_mpoly_sort_terms, gr_mpoly_sub, gr_mpoly_swap, gr_mpoly_write_pretty,
gr_mpoly_zero, gr_mul, gr_mul_2exp_fmpz, gr_mul_2exp_si, gr_mul_fmpq, gr_mul_fmpz,
gr_mul_other, gr_mul_si, gr_mul_two, gr_mul_ui, gr_neg, gr_neg_one, gr_nint,
gr_not_equal, gr_not_implemented, gr_not_in_domain, gr_one, gr_other_add,
gr_other_div, gr_other_divexact, gr_other_mul, gr_other_pow, gr_other_sub,
gr_partitions_fmpz, gr_partitions_ui, gr_partitions_vec, gr_pi, gr_poly_acos_series,
gr_poly_acosh_series, gr_poly_add, gr_poly_add_series, gr_poly_asin_series,
gr_poly_asinh_series, gr_poly_atan_series, gr_poly_atanh_series, gr_poly_clear,
gr_poly_compose, gr_poly_compose_divconquer, gr_poly_compose_horner, gr_poly_compose_series,
gr_poly_compose_series_brent_kung, gr_poly_compose_series_divconquer, gr_poly_compose_series_ho,
gr_poly_derivative, gr_poly_div, gr_poly_div_basecase, gr_poly_div_divconquer,
gr_poly_div_newton, gr_poly_div_series, gr_poly_div_series_basecase, gr_poly_div_series_invmul,
gr_poly_div_series_newton, gr_poly_divrem, gr_poly_divrem_basecase, gr_poly_divrem_divconquer,
gr_poly_divrem_newton, gr_poly_entry_ptr, gr_poly_equal, gr_poly_evaluate,
gr_poly_evaluate_horner, gr_poly_evaluate_other, gr_poly_evaluate_other_horner,
gr_poly_evaluate_other_rectangular, gr_poly_evaluate_rectangular, gr_poly_evaluate_vec_fast,
gr_poly_evaluate_vec_iter, gr_poly_exp_series, gr_poly_exp_series_basecase,
gr_poly_exp_series_basecase_mul, gr_poly_exp_series_newton, gr_poly_factor_squarefree,
gr_poly_fit_length, gr_poly_gcd, gr_poly_gcd_euclidean, gr_poly_gcd_hgcd, gr_poly_gen,
gr_poly_get_coeff_scalar, gr_poly_init, gr_poly_init2, gr_poly_integral, gr_poly_inv,
gr_poly_inv_series, gr_poly_inv_series_basecase, gr_poly_inv_series_newton, gr_poly_is_gen,
gr_poly_is_monic, gr_poly_is_one, gr_poly_is_zero, gr_poly_length, gr_poly_log1p_series,
gr_poly_log_series, gr_poly_make_monic, gr_poly_mul, gr_poly_mul_scalar, gr_poly_mullow,
gr_poly_neg, gr_poly_neg_one, gr_poly_nth_derivative, gr_poly_one, gr_poly_pow_fmpz,
gr_poly_pow_series_fmpq_recurrence, gr_poly_pow_series_ui, gr_poly_pow_series_ui_binexp,
gr_poly_pow_ui, gr_poly_pow_ui_binexp, gr_poly_print, gr_poly_randtest, gr_poly_rem,
gr_poly_resultant, gr_poly_resultant_euclidean, gr_poly_resultant_hgcd, gr_poly_resultant_small,
gr_poly_resultant_sylvester, gr_poly_reverse, gr_poly_roots, gr_poly_roots_other,
gr_poly_rsqrts_series, gr_poly_rsqrts_series_basecase, gr_poly_rsqrts_series_miller,
gr_poly_rsqrts_series_newton, gr_poly_set, gr_poly_set_coeff_fmpq, gr_poly_set_coeff_fmpz,
gr_poly_set_coeff_scalar, gr_poly_set_coeff_si, gr_poly_set_coeff_ui, gr_poly_set_fmpq,
gr_poly_set_fmpq_poly, gr_poly_set_fmpz, gr_poly_set_fmpz_poly, gr_poly_set_gr_poly_other,
gr_poly_set_scalar, gr_poly_set_si, gr_poly_set_ui, gr_poly_shift_left, gr_poly_shift_right,
gr_poly_sin_cos_series_basecase, gr_poly_sin_cos_series_tangent, gr_poly_sqrt_series,
gr_poly_sqrt_series_basecase, gr_poly_sqrt_series_miller, gr_poly_sqrt_series_newton,
gr_poly_squarefree_part, gr_poly_sub, gr_poly_sub_series, gr_poly_swap, gr_poly_tan_series,
gr_poly_tan_series_basecase, gr_poly_tan_series_newton, gr_poly_taylor_shift,
gr_poly_taylor_shift_convolution, gr_poly_taylor_shift_divconquer, gr_poly_taylor_shift_horner,
gr_poly_truncate, gr_poly_write, gr_poly_xgcd_euclidean, gr_poly_xgcd_hgcd, gr_poly_zero,
gr_polygamma, gr_polylog, gr_pow, gr_pow_fmpq, gr_pow_fmpz, gr_pow_other, gr_pow_si,
gr_pow_ui, gr_print, gr_println, gr_randtest, gr_randtest_not_zero, gr_randtest_small,
gr_re, gr_rfac, gr_rfac_fmpz, gr_rfac_ui, gr_rfac_vec, gr_rgamma, gr_riemann_xi,
gr_rising, gr_rising_ui, gr_rising_ui_forward, gr_rsqrts, gr_sec, gr_sec_pi, gr_sech,
gr_series_acos, gr_series_acosh, gr_series_add, gr_series_agm1, gr_series_airy,
gr_series_airy_ai, gr_series_airy_ai_prime, gr_series_airy_bi, gr_series_airy_bi_prime,
gr_series_asin, gr_series_asinh, gr_series_atan, gr_series_atanh, gr_series_beta_lower,
gr_series_clear, gr_series_cos_integral, gr_series_cosh_integral, gr_series_digamma,
gr_series_dirichlet_hardy_theta, gr_series_dirichlet_hardy_z, gr_series_dirichlet_l,
gr_series_div, gr_series_elliptic_k, gr_series_equal, gr_series_erf, gr_series_erfc,

`gr_series_erfi`, `gr_series_exp`, `gr_series_exp_integral_ei`, `gr_series_fresnel`, `gr_series_fresnel_s`, `gr_series_gamma`, `gr_series_gamma_lower`, `gr_series_gamma_upper`,
`gr_series_gen`, `gr_series_hurwitz_zeta`, `gr_series_hypgeom_pfq`, `gr_series_init`,
`gr_series_inv`, `gr_series_is_one`, `gr_series_is_zero`, `gr_series_jacobi_theta`, `gr_series_jacobi_theta_2`, `gr_series_jacobi_theta_3`, `gr_series_jacobi_theta_4`, `gr_series_lgamma`,
`gr_series_log`, `gr_series_log_integral`, `gr_series_make_exact`, `gr_series_mul`, `gr_series_neg`,
`gr_series_one`, `gr_series_polylog`, `gr_series_randtest`, `gr_series_rgamma`, `gr_series_rsqr`,
`gr_series_set`, `gr_series_set_fmpq`, `gr_series_set_fmpz`, `gr_series_set_gr_poly`, `gr_series_set_scalar`,
`gr_series_set_si`, `gr_series_set_ui`, `gr_series_sin_integral`, `gr_series_sinh_integral`,
`gr_series_sqrt`, `gr_series_sub`, `gr_series_swap`, `gr_series_tan`, `gr_series_weierstrass_p`,
`gr_series_write`, `gr_series_zero`, `gr_set`, `gr_set_d`, `gr_set_fmpq`, `gr_set_fmpz`, `gr_set_fmpz_2exp_fmpz`,
`gr_set_other`, `gr_set_shallow`, `gr_set_si`, `gr_set_str`, `gr_set_ui`, `gr_sgn`, `gr_sin`,
`gr_sin_cos`, `gr_sin_cos_pi`, `gr_sin_integral`, `gr_sin_pi`, `gr_sinc`, `gr_sinc_pi`, `gr_sinh`,
`gr_sinh_cosh`, `gr_sinh_integral`, `gr_spherical_y_si`, `gr_sqr`, `gr_sqrt`, `gr_stieltjes`,
`gr_stirling_s1_ui_vec`, `gr_stirling_s1_uiui`, `gr_stirling_s1u_ui_vec`, `gr_stirling_s1u_uiui`,
`gr_stirling_s2_ui_vec`, `gr_stirling_s2_uiui`, `gr_stream_init_file`, `gr_stream_init_str`,
`gr_stream_write`, `gr_stream_write_fmpz`, `gr_stream_write_free`, `gr_stream_write_si`,
`gr_stream_write_ui`, `gr_sub`, `gr_sub_fmpq`, `gr_sub_fmpz`, `gr_sub_other`, `gr_sub_si`,
`gr_sub_ui`, `gr_submul`, `gr_submul_fmpq`, `gr_submul_fmpz`, `gr_submul_other`, `gr_submul_si`,
`gr_submul_ui`, `gr_swap`, `gr_swap2`, `gr_tan`, `gr_tan_pi`, `gr_tanh`, `gr_test_add_aliasing`,
`gr_test_add_associative`, `gr_test_add_commutative`, `gr_test_add_type_variants`, `gr_test_addmul_submul`,
`gr_test_addmul_type_variants`, `gr_test_binary_op_aliasing`, `gr_test_binary_op_associative`,
`gr_test_binary_op_commutative`, `gr_test_binary_op_left_distributive`, `gr_test_binary_op_right_distributive`,
`gr_test_binary_op_type_variants`, `gr_test_complex_parts`, `gr_test_div_right_distributive`,
`gr_test_div_then_mul`, `gr_test_div_type_variants`, `gr_test_divexact`, `gr_test_divexact_type_variants`,
`gr_test_equal`, `gr_test_field`, `gr_test_get_fmpq`, `gr_test_get_fmpz`, `gr_test_get_fmpz_2exp_fmpz`,
`gr_test_get_si`, `gr_test_get_ui`, `gr_test_init_clear`, `gr_test_integral_domain`, `gr_test_inv_involutive`,
`gr_test_inv_multiplication`, `gr_test_iter`, `gr_test_mat_mul_classical_associative`,
`gr_test_mul_2exp_fmpz`, `gr_test_mul_2exp_si`, `gr_test_mul_aliasing`, `gr_test_mul_associative`,
`gr_test_mul_commutative`, `gr_test_mul_left_distributive`, `gr_test_mul_right_distributive`,
`gr_test_mul_then_div`, `gr_test_mul_type_variants`, `gr_test_multiplicative_group`, `gr_test_neg`,
`gr_test_one`, `gr_test_ordered_ring_cmp`, `gr_test_ordered_ring_cmpabs`, `gr_test_pow_fmpz_exponent_addition`,
`gr_test_pow_ui_aliasing`, `gr_test_pow_ui_base_multiplication`, `gr_test_pow_ui_base_scalar_multiplication`,
`gr_test_pow_ui_exponent_addition`, `gr_test_randtest_not_zero`, `gr_test_ring`, `gr_test_rsqr`,
`gr_test_set_fmpq`, `gr_test_set_fmpz`, `gr_test_set_si`, `gr_test_set_ui`, `gr_test_sqrt`,
`gr_test_sub_aliasing`, `gr_test_sub_equal_neg_add`, `gr_test_sub_type_variants`, `gr_test_submul_type_variants`,
`gr_test_swap`, `gr_test_vec_add`, `gr_test_vec_binary_op`, `gr_test_vec_div`, `gr_test_vec_divexact`,
`gr_test_vec_dot`, `gr_test_vec_mul`, `gr_test_vec_pow`, `gr_test_vec_sub`, `gr_test_zero_one`,
`gr_trunc`, `gr_vec_append`, `gr_vec_clear`, `gr_vec_entry_ptr`, `gr_vec_entry_srcptr`, `gr_vec_fit_length`,
`gr_vec_init`, `gr_vec_length`, `gr_vec_print`, `gr_vec_set`, `gr_vec_set_length`, `gr_vec_write`,
`gr_weierstrass_p`, `gr_weierstrass_p_inv`, `gr_weierstrass_p_prime`, `gr_weierstrass_sigma`,
`gr_weierstrass_zeta`, `gr_write`, `gr_write_n`, `gr_zero`, `gr_zeta`, `gr_zeta_nzeros`, `gr_zeta_ui`,
`gr_zeta_zero`, `gr_zeta_zero_vec`

- `gr_pos_inf`, `gr_neg_inf`, `gr_uinf`, `gr_undefined`, `gr_unknown`, `gr_arg`,
`gr_ctx_init_complex_extended_ca`, `gr_poly_divexact_basecase_bidirectional`,
`gr_poly_divexact_bidirectional`, `gr_poly_divexact_basecase`, `gr_poly_is_scalar`,
`gr_poly_div_series_divconquer`, `gr_poly_divexact_series_basecase`
- `nmod_mat_fprint_pretty`, `nmod_mat_print`, `nmod_mat_fprint`, `nmod_poly_is_monic`
- `nmod_poly_mat_set_trunc`, `nmod_poly_mat_truncate`, `nmod_poly_mat_shift_left`,
`nmod_poly_mat_shift_right`, `nmod_poly_mat_get_coeff_mat`, `nmod_poly_mat_set_coeff_mat`,
`nmod_poly_mat_set_nmod_mat`, `nmod_poly_mat_equal_nmod_mat`, `nmod_poly_mat_degree`
- `qqbar_set_fmpz`
- `fmpq_mpoly_push_term_fmpq_ffmpz`, `fmpq_mpoly_push_term_fmpz_ffmpz`,
`fmpq_mpoly_push_term_ui_ffmpz`, `fmpq_mpoly_push_term_si_ffmpz`,
`fmpz_mod_mpoly_push_term_fmpz_ffmpz`, `fmpz_mod_mpoly_push_term_ui_ffmpz`,
`fmpz_mod_mpoly_push_term_si_ffmpz`, `fmpz_mpoly_push_term_fmpz_ffmpz`,

```
fmpz_mpoly_push_term_ui_ffmpz, fmpz_mpoly_push_term_si_ffmpz,  
fq_nmod_mpoly_push_term_fq_nmod_ffmpz, nmod_mpoly_push_term_ui_ffmpz,  
fmpq_mpoly_push_term_fmpz_ffmpz, fmpq_mpoly_push_term_fmpq_ffmpz,  
fmpq_mpoly_push_term_ui_ffmpz, fmpq_mpoly_push_term_si_ffmpz,  
fmpq_mpoly_push_term_fmpq_ffmpz
```

List of removals

- The following functions that were present in FLINT 2.9, Arb 2.23 or Calcium 0.4 have been removed, deprecated, or replaced. Most are algorithms obsoleted by new gr implementations, functions dealing with removed types (fmpz) or GMP types (mpz, etc.), and internal functions that are no longer needed.
- `__fmpz_clear`, `__fmpz_eq`, `__fmpz_gt`, `__fmpz_gte`, `__fmpz_init`, `__fmpz_init_set`, `__fmpz_init_set_ui`, `__fmpz_lt`, `__fmpz_lte`, `__fmpz_neg`, `__fmpz_neq`, `__fmpz_set_si`, `__fmpz_set_ui`
- `__fmpz_mod_poly_div_divconquer`, `__fmpz_mod_poly_divrem_divconquer`, `__fq_nmod_poly_divrem_divconquer`, `__fq_poly_divrem_divconquer`, `__fq_zech_poly_divrem_divconquer`
- `__nmod_poly_div_divconquer`, `__nmod_poly_divrem_divconquer`, `__nmod_poly_invsqrt_series_prealloc`
- `_acb_poly_compose_axnc`, `_acb_poly_compose_divconquer`, `_acb_poly_compose_horner`, `_acb_poly_compose_series_brent_kung`, `_acb_poly_compose_series_horner`, `_acb_poly_sin_cos_series_basecase`, `_acb_poly_sin_cos_series_tangent`, `_acb_poly_taylor_shift_convolution`, `_acb_poly_taylor_shift_divconquer`, `_acb_poly_taylor_shift_horner`
- `acb_rising_ui_bs`, `acb_rising_ui_rs`, `acb_rising_ui_rec`
- `_arb_poly_compose_axnc`, `_arb_poly_compose_divconquer`, `_arb_poly_compose_horner`, `_arb_poly_compose_series_brent_kung`, `_arb_poly_compose_series_horner`, `_arb_poly_sin_cos_series_basecase`, `_arb_poly_sin_cos_series_tangent`, `_arb_poly_taylor_shift_convolution`, `_arb_poly_taylor_shift_divconquer`, `_arb_poly_taylor_shift_horner`
- `arb_rising_ui_bs`, `arb_rising_ui_rs`, `arb_rising_ui_rec`, `arb_rising2_ui_bs`, `arb_rising2_ui_rs`, `arb_rising2_ui`
- `_arith_bernoulli_number_vec_zeta`, `_arith_bernoulli_number_zeta`, `_arith_cos_minpoly`, `_arith_euler_number_zeta`, `_arith_number_of_partitions_mpfr`
- `_ca_poly_atan_series`, `_ca_poly_compose_divconquer`, `_ca_poly_compose_horner`
- `_fmpq_poly_set_array_mpq`
- `_fmpz_add_1x1`, `_fmpz_add_eps`, `_fmpz_add_mpn`, `_fmpz_mul_1x1`, `_fmpz_mul_mpn`, `_fmpz_normalise_naive`, `_fmpz_set_round`, `_fmpz_set_round_mpn`
- `_fmpz_deprecated_multi_crt_local_size`, `_fmpz_deprecated_multi_crt_run`, `_fmpz_deprecated_multi_crt_run_p`, `_fmpz_mod_poly_compose_divconquer`, `_fmpz_mod_poly_compose_divconquer_recursive`, `_fmpz_mod_poly_compose_horner`, `_fmpz_mod_poly_div_basecase`, `_fmpz_mod_poly_div_divconquer`, `_fmpz_mod_poly_div_divconquer_recursive`, `_fmpz_mod_poly_div_newton`, `_fmpz_mod_poly_divrem_divconquer`, `_fmpz_mod_poly_divrem_divconquer_recursive`, `_fmpz_mod_poly_gcd_cofactors`, `_fmpz_mod_poly_gcd_euclidean`, `_fmpz_mod_poly_gcd_hgcd`, `_fmpz_mod_poly_hgcd_recursive`, `_fmpz_mod_poly_hgcd_recursive_iter`, `_fmpz_mod_poly_hgcd_res`, `_fmpz_mod_poly_xgcd_euclidean`, `_fmpz_mod_poly_xgcd_hgcd`, `_fmpz_poly_evaluate_mpfr`
- `_fmpz_ui_pow_ui`, `_fmpz_vec_get_mpf_vec`

- `_fq_nmod_poly_compose_divconquer`, `_fq_nmod_poly_compose_horner`,
`_fq_nmod_poly_div_basecase`, `_fq_nmod_poly_divrem_basecase`,
`_fq_nmod_poly_divrem_divconquer`, `_fq_nmod_poly_divrem_divconquer_recursive`,
`_fq_nmod_poly_gcd_euclidean`, `_fq_nmod_poly_gcd_hgcd`, `_fq_nmod_poly_hgcd`,
`_fq_nmod_poly_hgcd_recursive`, `_fq_nmod_poly_hgcd_recursive_iter`,
`_fq_nmod_poly_xgcd_euclidean`
- `_fq_poly_compose_divconquer`, `_fq_poly_compose_horner`, `_fq_poly_div_basecase`,
`_fq_poly_divrem_basecase`, `_fq_poly_divrem_divconquer`, `_fq_poly_divrem_divconquer_recursive`,
`_fq_poly_gcd_euclidean`, `_fq_poly_gcd_hgcd`, `_fq_poly_hgcd`,
`_fq_poly_hgcd_recursive`, `_fq_poly_hgcd_recursive_iter`, `_fq_poly_xgcd_euclidean`
- `_fq_zech_poly_compose_divconquer`, `_fq_zech_poly_compose_horner`,
`_fq_zech_poly_div_basecase`, `_fq_zech_poly_divrem_basecase`,
`_fq_zech_poly_divrem_divconquer`, `_fq_zech_poly_divrem_divconquer_recursive`,
`_fq_zech_poly_gcd_euclidean`, `_fq_zech_poly_gcd_hgcd`, `_fq_zech_poly_hgcd`,
`_fq_zech_poly_hgcd_recursive`, `_fq_zech_poly_hgcd_recursive_iter`,
`_fq_zech_poly_xgcd_euclidean`
- `_nmod_mat_set_mod`
- `_nmod_poly_compose_divconquer`, `_nmod_poly_compose_series_brent_kung`,
`_nmod_poly_compose_series_divconquer`, `_nmod_poly_compose_series_horner`,
`_nmod_poly_div_basecase`, `_nmod_poly_div_basecase_1`, `_nmod_poly_div_basecase_2`,
`_nmod_poly_div_basecase_3`, `_nmod_poly_div_divconquer`, `_nmod_poly_div_divconquer_recursive`,
`_nmod_poly_div_newton`, `_nmod_poly_divrem_basecase_1`, `_nmod_poly_divrem_basecase_2`,
`_nmod_poly_divrem_basecase_3`, `_nmod_poly_divrem_divconquer`,
`_nmod_poly_divrem_divconquer_recursive`, `_nmod_poly_divrem_newton`,
`_nmod_poly_divrem_q0`, `_nmod_poly_divrem_q1`, `_nmod_poly_exp_series_basecase`,
`_nmod_poly_exp_series_monomial_ui`, `_nmod_poly_exp_series_newton`,
`_nmod_poly_hgcd_recursive`, `_nmod_poly_hgcd_recursive_iter`, `_nmod_poly_hgcd_res`,
`_nmod_poly_integral_offset`, `_nmod_poly_log_series_monomial_ui`,
`_nmod_poly_rem_basecase`, `_nmod_poly_rem_basecase_1`, `_nmod_poly_rem_basecase_2`,
`_nmod_poly_rem_basecase_3`
- `acb_poly_compose_divconquer`, `acb_poly_compose_horner`, `acb_poly_compose_series_brent_kung`,
`acb_poly_compose_series_horner`, `acb_poly_sin_cos_series_basecase`,
`acb_poly_sin_cos_series_tangent`, `acb_poly_taylor_shift_convolution`,
`acb_poly_taylor_shift_divconquer`, `acb_poly_taylor_shift_horner`
- `arb_flint_get_num_available_threads`
- `arb_poly_compose_divconquer`, `arb_poly_compose_horner`, `arb_poly_compose_series_brent_kung`,
`arb_poly_compose_series_horner`, `arb_poly_sin_cos_series_basecase`,
`arb_poly_sin_cos_series_tangent`, `arb_poly_taylor_shift_convolution`,
`arb_poly_taylor_shift_divconquer`, `arb_poly_taylor_shift_horner`
- `arb_test_multiplier`
- `arb_thread_pool_num_available`
- `arf_get_fmpr`, `arf_set_fmpr`
- `arith_cos_minpoly`, `arith_number_of_partitions_mpfr`
- `ca_mat_transpose_resize`, `ca_poly_atan_series`, `ca_poly_compose_divconquer`,
`ca_poly_compose_horner`, `calcium_test_multiplier`
- `cos_minpoly`, `cos_pi_pq`
- `fmpq_poly_evaluate_mpq`, `fmpq_poly_evaluate_mpz`, `fmpq_poly_get_coeff_mpq`,
`fmpq_poly_scalar_div_mpq`, `fmpq_poly_scalar_div_mpz`, `fmpq_poly_scalar_mul_mpq`,
`fmpq_poly_scalar_mul_mpz`, `fmpq_poly_set_array_mpq`, `fmpq_poly_set_coeff_mpq`,
`fmpq_poly_set_coeff_mpz`, `fmpq_poly_set_mpq`, `fmpq_poly_set_mpz`

- `fmpz_add`, `fmpz_add_fmpz`, `fmpz_add_naive`, `fmpz_add_si`, `fmpz_add_ui`, `fmpz_addmul`, `fmpz_addmul_fmpz`, `fmpz_addmul_si`, `fmpz_addmul_ui`, `fmpz_check_ulp`, `fmpz_cmp`, `fmpz_cmp_2exp_si`, `fmpz_cmpabs`, `fmpz_cmpabs_2exp_si`, `fmpz_cmpabs_ui`, `fmpz_div`, `fmpz_div_fmpz`, `fmpz_div_si`, `fmpz_div_ui`, `fmpz_exp`, `fmpz_expm1`, `fmpz_fmpz_div`, `fmpz_fmpz_div_fmpz`, `fmpz_get_d`, `fmpz_get_fmpq`, `fmpz_get_fmpz`, `fmpz_get_fmpz_2exp`, `fmpz_get_fmpz_fixed_fmpz`, `fmpz_get_fmpz_fixed_si`, `fmpz_get_mpfr`, `fmpz_get_si`, `fmpz_log`, `fmpz_log1p`, `fmpz_mul`, `fmpz_mul_fmpz`, `fmpz_mul_naive`, `fmpz_mul_si`, `fmpz_mul_ui`, `fmpz_pow_sloppy_fmpz`, `fmpz_pow_sloppy_si`, `fmpz_pow_sloppy_ui`, `fmpz_print`, `fmpz_printd`, `fmpz_randtest`, `fmpz_randtest_not_zero`, `fmpz_randtest_special`, `fmpz_root`, `fmpz_rsqr`, `fmpz_set_d`, `fmpz_set_fmpq`, `fmpz_set_fmpz_2exp`, `fmpz_set_mpfr`, `fmpz_set_round_ui_2exp_fmpz`, `fmpz_set_round_uiui_2exp_fmpz`, `fmpz_si_div`, `fmpz_sqrt`, `fmpz_sub`, `fmpz_sub_fmpz`, `fmpz_sub_si`, `fmpz_sub_ui`, `fmpz_submul`, `fmpz_submul_fmpz`, `fmpz_submul_si`, `fmpz_submul_ui`, `fmpz_ui_div`, `fmpz_ulp`
- `fmpz_deprecated_multi crt`, `fmpz_deprecated_multi crt_clear`, `fmpz_deprecated_multi crt_init`, `fmpz_deprecated_multi crt_precomp`, `fmpz_deprecated_multi crt_precomp_p`, `fmpz_deprecated_multi crt_precompute`, `fmpz_deprecated_multi crt_precompute_p`
- `fmpz_mat_col_equal`, `fmpz_mat_get_mpf_mat`, `fmpz_mat_row_equal`
- `fmpz_mod_ctx_get_modulus_mpf_read_only`
- `fmpz_mod_poly_compose_divconquer`, `fmpz_mod_poly_compose_horner`, `fmpz_mod_poly_div_basecase`, `fmpz_mod_poly_div_divconquer`, `fmpz_mod_poly_div_newton`, `fmpz_mod_poly_divrem_divconquer`, `fmpz_mod_poly_gcd_euclidean`, `fmpz_mod_poly_gcd_hgcd`, `fmpz_mod_poly_get_coeff_mpf`, `fmpz_mod_poly_set_coeff_mpf`, `fmpz_mod_poly_xgcd_euclidean`, `fmpz_mod_poly_xgcd_hgcd`
- `fmpz_poly_evaluate_mpf`, `fmpz_poly_evaluate_mpf`, `fmpz_poly_get_coeff_mpf`, `fmpz_poly_q_evaluate`, `fmpz_poly_q_scalar_div_mpf`, `fmpz_poly_q_scalar_div_mpf`, `fmpz_poly_q_scalar_mul_mpf`, `fmpz_poly_q_scalar_mul_mpf`, `fmpz_poly_scalar_divexact_mpf`, `fmpz_poly_scalar_fdiv_mpf`, `fmpz_poly_scalar_mul_mpf`, `fmpz_poly_set_coeff_mpf`, `fmpz_poly_set_mpf`
- `fq_nmod_poly_compose_divconquer`, `fq_nmod_poly_compose_horner`, `fq_nmod_poly_divrem_basecase`, `fq_nmod_poly_divrem_divconquer`, `fq_nmod_poly_gcd_euclidean`, `fq_nmod_poly_gcd_hgcd`, `fq_nmod_poly_xgcd_euclidean`, `fq_poly_compose_divconquer`, `fq_poly_compose_horner`, `fq_poly_divrem_basecase`, `fq_poly_divrem_divconquer`, `fq_poly_gcd_euclidean`, `fq_poly_gcd_hgcd`, `fq_poly_xgcd_euclidean`, `fq_zech_poly_compose_divconquer`, `fq_zech_poly_compose_horner`, `fq_zech_poly_divrem_basecase`, `fq_zech_poly_divrem_divconquer`, `fq_zech_poly_gcd_euclidean`, `fq_zech_poly_gcd_hgcd`, `fq_zech_poly_xgcd_euclidean`
- `mag_get_mpf`, `mag_set_mpf`
- `mpfr_cos_pi_pq`, `mpfr_zeta_inv_euler_product`
- `nmod_poly_compose_divconquer`, `nmod_poly_compose_series_brent_kung`, `nmod_poly_compose_series_divconquer`, `nmod_poly_compose_series_horner`, `nmod_poly_div_basecase`, `nmod_poly_div_divconquer`, `nmod_poly_div_newton`, `nmod_poly_divrem_divconquer`, `nmod_poly_divrem_newton`, `nmod_poly_exp_series_basecase`, `nmod_poly_exp_series_monomial_ui`, `nmod_poly_factor_get_nmod_poly`, `nmod_poly_log_series_monomial_ui`, `nmod_poly_rem_basecase`, `nmod_poly_set_fmpz_poly`, `sinh_cosh_divk_precomp`
- `_nmod_poly_powmod_mpf_binexp`, `nmod_poly_powmod_mpf_binexp`, `_nmod_poly_powmod_mpf_binexp_preinv`, `nmod_poly_powmod_mpf_binexp_preinv`, `_nmod_poly_powmod_mpf_binexp`, `nmod_poly_powmod_mpf_binexp`, `_nmod_poly_powmod_mpf_binexp_preinv`, `nmod_poly_powmod_mpf_binexp_preinv`

2022-06-24 – FLINT 2.9.0

- Add `fmpz_mod_poly_div` function
- Add `_flint_get_memory` function
- Add Eulerian polynomials
- Support “multivariate” polynomials with zero variables
- Improve Stirling numbers of both kinds
- Speed up numerous `fmpz` functions for small inputs
- Improve Bell numbers
- Speedups to `nmod` arithmetic
- Improve `nmod_mat` LU decomposition
- Fully separate `nmod` module from `nmod_vec`
- Speed up Hermite polynomials
- Add n -th derivative for $\mathbb{Z}[x]$ and $\mathbb{Q}[x]$
- Improve `fq_default` module (`nmod` is now used where optimal)
- Add `sqrt` functions for numerous polynomial/series modules and finite fields
- Add FFT matrix multiplication
- Improve CI
- Improve LLL for general use
- Add matrix-vector products over $\mathbb{Q}$
- Add `can_solve` function for `fmpq_mat`, handling non-square/singular matrices
- Document `fmpz_mod_vec` module
- Fix and document `qadic_sqrt` function
- Add parallel programming helpers

2022-04-25 – FLINT 2.8.5

- Fix a serious bug in LLL

2021-11-17 – FLINT 2.8.4

- Fix a serious bug in `fmpz_mod_poly_xgcd` for polynomials of large length
- Fix an assertion failure in `fmpz_mat_solve_fflu` (only relevant if asserts enabled)
- Fix some bugs on 32 bit machines
- Work around a compiler bug on Apple M1
- Fix bug in `nmod_mpoly_factor` (returned 0 for some factorisations)
- Fix some documentation build errors and some doc formatting issues

2021-11-03 – FLINT 2.8.3

- Fix a serious bug in `nmod_poly_xgcd_hgcd`, `nmod_poly_xgcd`, `fmpz_poly_xgcd_modular`, `fmpz_poly_xgcd`, `fmpz_poly_xgcd` for polynomials of length ≥ 340 .
- Fix some copyright assignments
- Fix some documentation errors

2021-10-15 – FLINT 2.8.2

- Fix an issue with `-disable-dependency-tracking` on some distributions

2021-10-01 – FLINT 2.8.1

- Numerous bug fixes
- Adjust soname on android
- Allow disabling of dependency tracking

2021-07-23 – FLINT 2.8.0

- New `fq_default` module which combines existing finite fields
- Speedups for linear algebra when using BLAS and/or threading
- New series expansions with coefficients in $\mathbb{Q}\mathbb{Q}$
- Faster CRT
- New `fmpz_mod_mpoly` module
- Polynomial factoring improvements over $\mathbb{Z}\mathbb{Z}$
- Fixed bugs in `gmpcompat` on Windows
- Add `fmpz_mat_can_solve_fflu` and `fmpz_mat_can_solve`
- Cleanup of the `nmod_poly` and `nmod_poly_factor` code
- Implement `nmod_mat_det_howell`
- Add `fmpz_mod_poly_divides`, `fmpz_divides`, `n_divides`, `nmod_poly_divides`
- Interface for multiplying matrices by vectors and arrays
- Nearest Euclidean division
- Subresultant GCD
- XGCD over $\mathbb{Z}\mathbb{Z}$ with canonical Bezout coefficients
- Add `fmpz_mpoly` resultant and discriminant
- Add deprecations list
- Add `FLINT_SGN` macro
- Speedups for series computations
- Switch to GitHub Actions for CI
- Improve Taylor shift
- Numerous bug fixes and speedups

2021-01-18 – FLINT 2.7.1

- Fix build bug due to missing test files
- Fix bug in `fmmpz_mod_poly_factor` when there are more than five factors
- Fix issue when using MPIR 3.0.0 on Win64 with command line build
- Fix bug in `fmmpz_mod_poly_div_series`
- Fix some broken asserts
- Support standard GNU installation directories in CMake build
- Fix stack overflow with ICC

2020-12-18 – FLINT 2.7.0

- Multivariate factorisation
- Square root and square testing for finite fields
- Square root and square testing for multivariates
- Zassenhaus factoring speedups (incl. degree pruning)
- Fast factorisation of cubic univariate polynomials
- Add context objects to `fmmpz_mod_poly` functions
- Use BLAS for matrix multiplication over $\mathbb{Z}/n\mathbb{Z}$ (small n)
- Linear solving for non-square/singular matrices (`can_solve`)
- Speed up factorisation over $\mathbb{Z}/n\mathbb{Z}$ (for multiprecision n)

2020-08-12 – FLINT 2.6.3

- Fix a bug in generator of finite field in characteristic 2
- Allow FLINT to work with GMP 6.1.2 and 6.2.0 interchangeably
- Fix some old license headers

2020-07-31 – FLINT 2.6.2

- Fix for choice of generator in an fq finite field of degree one
- Fix an incorrectly written test

2020-07-23 – FLINT 2.6.1

- Fix issues on Debian major architectures
- Fix numerous reported bugs (`mpoly`, `fq_poly`, `mpn_mul_1`, mod 2 code, etc.)

2020-06-05 – FLINT 2.6.0

- multivariate polynomials over most standard rings (sparse distributed)
- APR-CL primality proving
- elliptic curve integer factoring
- minpoly and charpoly
- improved quadratic sieve for integer factoring
- embeddings of finite fields
- pollard rho integer factoring
- p+1 integer factoring
- best of breed smooth integer factoring routine
- best of breed general integer factoring routine
- howell and strong echelon form
- large speedups for solve and hence inverse over $\mathbb{Z}$ and $\mathbb{Q}$
- randprime and nextprime functions
- pernetstein HNF improvements
- moller-granlund precomputed inverses
- resultant_modular_div
- fibonacci polynomials
- exception mechanism/flint_abort
- sqrt of series and polynomials
- division of series over $\mathbb{Z}$
- power sums
- improved base cases of various power series functions
- ability to switch memory allocators
- fast recurrence for Hermite polys
- shifted Legendre polynomials
- Laguerre polynomials
- Gegenbauer polys
- sphinx documentation
- van hoeij with gradual feeding implementation of polynomial factoring over $\mathbb{Z}$
- perfect power detection
- divisibility testing for polynomials
- fast block based memory manager for bundling fmpz allocations
- uniform random generation
- CMake build system
- linear algebra speedups when everything can be kept in longs
- nmod module for integers mod (small) n
- fmpz_mod_mat module for matrices over integers mod multiprecision n
- kronecker product (tensor product)

- random primitive polys (for finite fields)
- thread pool implementation
- threading of FFT for integer and polynomial multiplication over $\mathbb{Z}$
- threading of quadratic sieve for integer factoring
- improved threading of factoring of polynomials mod p
- threading for multivariate polynomial multiplication, division and GCD
- threaded multiplication of matrices mod p
- Berlekamp-Massey (nmod)
- fmpz_mod module for integers mod multiprecision n
- Pohlig-Hellman (discrete log)
- farey_neighbours
- remove openMP option
- additional integer division variants
- speed up mpn_mulmod_preinv
- fft precaching
- cyclotomic polynomial detection
- polynomial root finding over finite fields
- GMP 6.2 support
- MPIR 3.0.0 support
- many small speedups and additional convenience functions added

2015-08-13 – FLINT 2.5.2

- Fix further issues with soname versioning and ldconfig
- Fix a bug when using GMP instead of MPIR.

2015-08-12 – FLINT 2.5.1

- Fix some build bugs related to soname versioning and ldconfig
- Fix issue with Windows MSVC build

2015-08-07 – FLINT 2.5.0

- LLL (rational, Nguyen-Stehle, from Gram matrix, with_removal, Storjohann/ULLL)
- Hermite normal form (naive, xgcd, Domich-Kannan-Trotter, Kannan-Bachem, Pernet-Stein)
- Smith normal form (diagonal, Kannan-Bachem, Iliopoulos)
- Paterson-Stockmeyer algorithm
- modular resultant
- hgcd resultant
- polynomial discriminant
- multithreaded multimodular Taylor shift

- multithreaded Brent-Kung composition
- multithreaded Kaltofen-Shoup distinct degree factorisation
- multiplication based reduced row echelon form
- place inline functions in library for foreign function interfaces
- Primality tests for large integers (Pocklington, Morrison)
- Probable prime tests for large integers (Lucas, Baillie-PSW, strong-psp, Brillhart-Lehmer-Selfridge)
- CRT for large integers
- Dixon algorithm for nullspace
- Brent-Kung composition in irreducibility and distinct degree factorisation
- floating point QR decomposition
- Schwarz-Rutishauser Gram-Schmidt algorithm
- Ogita-Rump-Oishi dot product
- matrix window functions
- MSVC support (Brian Gladman)
- fast cube/nth-root (Newton, Kahan, magic number, Chebyshev)
- Bodrato matrix squaring
- matrix concatenation functions
- matrix content
- faster `n_gcd`
- faster `n_sqrtmod` and `fmpz_sqrtmod`
- additional functions for returning factor of modulus in polys over $\mathbb{Z}/n\mathbb{Z}$
- Hadamard matrix construction
- series addition/subtraction
- faster `prime_pi` bounds
- speedup creation of sparse polynomials
- speedup `n_isprime` `n_nextprime`
- speedup `n_isprime_pocklington`
- speedups to `fmpq_poly` and `fmpz_poly` arithmetic
- speedup polynomial irreducibility testing over $\mathbb{Z}/p\mathbb{Z}$
- speedup of rank computation over $\mathbb{ZZ}$
- made CPimport compile time dependency only
- teach `flint_printf/sprintf` about explicit width format specifiers
- support relative paths in configure
- library soname versioning
- ARM64 patches
- Support MSYS2
- Progress towards supporting MIPS64
- Fix a serious bug in `fmpz_poly_signature`

????-??-?? – FLINT 2.4.5

- fixed a severe bug in flint's `fmmpz_poly_gcd_heuristic`, reported by Anton Mellit.

????-??-?? – FLINT 2.4.4

- fixed a severe bug in flint's primality code (`n_is_prime()` affecting `n_factor()`)

2014-04-01 – FLINT 2.4.3

- Fix a linker issue on Mac OSX.

2014-03-11 – FLINT 2.4.2

- Fix bug in ARM assembly

2012-11-20 – FLINT 2.4

- C++ expressions template wrapper
- Fast factorisation of polynomials over $\mathbb{Z}/n\mathbb{Z}$
- improved p-adics
- polynomials/matrices over p-adics
- qadics
- Finite fields (small and large $\mathbb{F}_q$), polynomials/matrices over $\mathbb{F}_q$
- Finite fields with Zech logarithm representation
- Fast factorisation of polynomials over $\mathbb{F}_q$
- Faster Brent-Kung modular composition
- New prime sieving code
- Lambert-W function
- Precomputed inverses for polynomials and large integers
- Williams' P+1 integer factoring algorithm
- Harvey's KS2/KS4 polynomial multiplication
- Faster primality testing up to 64 bits
- Support for Cygwin64 and MinGW64
- Support for Clang
- Support for GMP
- Support for Boehm-Demers-Weiser GC
- Support for flint extension modules

2012-07-01 – FLINT 2.3

- general
 - many changes to the build system
 - added NTL interface
 - switched to custom memory allocation functions `flint_malloc` etc
 - in addition to the entries below, fixed a large number of memory leaks, problems with the test code, and bugs in corner cases of various functions
 - added `_fmpz_cleanup_mmpz_content` as an alternative to `_fmpz_cleanup`
 - support MinGW32
 - support Cygwin
 - bugfix on ia64
 - support sparc32/sparc64
 - support OSX
 - support Solaris, NetBSD, OpenBSD, etc (if bash, GNU Make present)
- `ulong_extras`
 - implemented the improved Lehman algorithm
 - added `n_jacobi_unsigned` to allow `n > WORD_MAX`
 - fixed `n_sqrtmod` for `n > WORD_MAX`
 - fixed bug causing `n_sqrtmod` to hang
 - added sublinear algorithm for computing factorials mod `p`
 - added `n_sqrtmod_primepow`, `n_sqrtmodn` and associated functions for computing square roots modulo composite integers
 - fixed bugs in `n_is_prime_pocklington`
 - fixed `UWORD_MAX` case in `powmod` and `powmod2`
 - fixed problems with the random number generators
 - fixed rare bug in `n_mod_precomp`
 - fixed rare bug in `n_is_prime_pseudosquare`
- `long_extras`
 - added `z_sizeinbase`
- `qsieve`
 - new module implementing a quadratic sieve for numbers up to two limbs
- `fft`
 - new module providing an efficient Schoenhage-Strassen FFT
- `longlong`
 - added assembly code for ia64 and ARM
 - fixed bug in fallback version of `add_sssaaaaaa`
- `fmpz`
 - added `fmpz_fib_ui`
 - added double precision natural logarithm

- added `fmpz_val2` for 2-valuation
- added `mul_2exp`, `div_2exp`, `cdiv_q_2exp`, `tdiv_q_2exp`, `fdiv_r`, `fdiv_r_2exp`, `tdiv_ui`, `mul_tdiv_q_2exp`
- added `get_d/set_d`
- added `fmpz_divisible`, `divisible_si`
- optimised `fmpz_powm` and `fmpz_powm_ui`
- added `clog`, `clog_ui`, `flog`, `flog_ui` for computing logarithms
- added `abs_lbound_ui_2exp`, `ubound_ui_2exp`
- added `fmpz_rfac_ui` and `fmpz_rfac_uiui` for rising factorials
- added functions to obtain read-only `fmpz_t`'s from `mpz_t`'s
- added `fmpz_init_set`, `init_set_ui`
- added `fmpz_gcdinv`
- added `fmpz_is_square`
- added `fmpz_tstbit`, `setbit`, `clrbit`, `complement`, `combit`, `and`, `or`, `xor`, `popcnt`
- added a sign flag for CRT instead of using separate methods
- fixed bugs in `fmpz_sqrtmod`
- fixed a bug in `fmpz_bit_unpack` that could cause corruption of the global `fmpz` array when compiled in single mode
- fixed a bug in `fmpz_sub_ui` that could cause memory corruption
- `fmpz_vec`
 - added functions for obtaining the largest absolute value coefficient
 - added functions for computing the sum and product of an integer vector
 - made `max_bits` much faster
 - added `_fmpz_vec_mod_fmpz`
 - made `randtest` produce sparse output
- `fmpz_poly`
 - added `fmpz_poly_sqr`, `fmpz_poly_sqr_low` for squaring a polynomial
 - added `fmpz_poly_lcm`
 - made multipoint interpolation faster by using the Newton basis
 - added a function for fast division by a linear polynomial
 - added power series composition (classical and Brent-Kung)
 - added power series reversion (classical, Newton, fast Lagrange)
 - added a function for obtaining the largest absolute value coefficient
 - fixed quadratic memory usage and stack overflow when performing unbalanced division or pseudo division using the `divconquer` algorithm
 - fixed a bug in `poly_zero_coeffs`
 - fixed a bug in `xgcd_modular`
 - allowing ± 1 in the constant term of power series inversion
 - fixed aliasing bug in `divrem`
 - added restartable Hensel lifting and associated utility functions

- fixed `rem`, which used to only call the basecase algorithm
- fixed `pseudo_divrem`, which used to only call the basecase algorithm
- implemented Schoenhage-Strassen multiplication (`mul_SS`, `mullow_SS`) and enabled this by default
- fixed a bug in the heuristic GCD algorithm
- added functions for Newton basis conversion
- added functions for fast Taylor shift
- added `fmpz_poly_sqrt` implementing a basecase algorithm
- added scalar `mul_2exp`, `fdiv_2exp`, `tdiv_2exp`
- made `randtest` produce sparse output
- added `fmpz_poly_equal_fmpz`
- improved performance by always using basecase multiplication when one polynomial is short
- improved algorithm selection for `fmpz_poly_gcd`
- fixed several bugs in `gcd_modular`
- improved performance of `gcd_modular`
- `fmpz_poly_factor`
 - new module for factorisation of `fmpz_polys`
 - added a naive implementation of the Zassenhaus algorithm
- `fmpz_mod_poly`
 - new module for polynomials modulo over $\mathbb{Z}/n\mathbb{Z}$ for arbitrary-precision n
 - multiplication, powering
 - classical and `divconquer` division
 - series inversion
 - Euclidean GCD and `XGCD`
 - `invmod`
 - radix conversion
 - `divconquer` composition
 - GCD and division functions that test invertibility of the leading coefficient
- `fmpz_mat`
 - added `det_divisor` for computing a random divisor of the determinant
 - faster determinant computation using divisor trick
 - faster determinant computation by using multimodular updates
 - fixed $n \times 0 \times m$ product not zeroing the result
 - various interface improvements
 - faster implementation of Cramer’s rule for multiple right hand sides
 - added `fmpz_mat_fread` and `read`
 - added multi CRT/mod functions
 - added `trace`
- `fmpz_poly_mat`

- fixed $n \times 0 \times m$ product not zeroing the result
- added inverse
- added rank computation
- added reduced row echelon form and nullspace computation
- added more utility functions
- added squaring and exponentiation
- added balanced product of a sequence of matrices
- added truncate, mulow, sqrlow, pow_trunc
- added trace
- fmpz_factor
 - new module providing interface for integer factorisation
 - fast expansion of a factored integer
- fmpq
 - cleaned up and improved performance of rational reconstruction code
 - allow separate numerator and denominator bounds for rational reconstruction
 - added continued fraction expansion
 - added functions for summation using binary splitting
 - added fmpq_swap
 - added fmpq_print, fmpq_get_str
 - added fmpq_pow_si
 - added functions to obtain read-only fmpq_t's from mpq_t's
 - added fmpq_cmp
- fmpq_mat
 - fixed $n \times 0 \times m$ product not zeroing the result
 - added fmpq_mat_transpose
 - added trace
- fmpq_poly
 - improved speed of multipoint interpolation using __fmpz_poly_div_root
 - fmpq_poly: added power series composition (classical and Brent-Kung)
 - fmpq_poly: added power series reversion (classical, Newton, fast Lagrange)
 - fixed bug wherein set_array_mpq modified the input
 - added gcd, xgcd, lcm, resultant
 - added fmpq_poly_set_fmpz
 - added fmpq_poly_get_slice, fmpq_poly_reverse
 - fixed aliasing bug in divrem
 - changed some functions to use FLINT scalar types instead of MPIR data types
 - added fmpq_poly_get_numerator
- nmod_poly
 - implemented the half gcd algorithm for subquadratic gcd and xgcd

- added multipoint evaluation and interpolation
- added asymptotically fast multipoint evaluation and interpolation
- added a function for forming the product of linear factors
- added a function for fast division by a linear polynomial
- added power series composition (classical and Brent-Kung)
- added power series reversion (classical, Newton, fast Lagrange)
- added `nmod_poly_mulmod`, `powmod` and related functions (ported from `flint1`)
- added squarefree, irreducibility tests (ported from `flint1`)
- added Berlekamp and Cantor-Zassenhaus factoring (ported from `flint1`)
- fixed quadratic memory usage and stack overflow when performing unbalanced division using the `divconquer` algorithm
- added `compose_series_divconquer`
- added resultant
- fixed aliasing bug in `divrem`
- added `rem` functions
- added `divrem_q0`, `q1` for special cases of division
- added functions for fast Taylor shift
- added `nmod_poly_sqrt`
- made `fread` read the modulus from the file
- made `randtest` produce sparse output
- fixed bug in `xgcd_euclidean` with scalar inputs
- `nmod_vec`
 - added functions and macros for computing dot products
 - made `randtest` produce sparse output
- `nmod_mat`
 - added `addmul/submul` functions
 - asymptotically fast solving of triangular systems
 - asymptotically fast LUP decomposition
 - asymptotically fast determinant and rank computation
 - asymptotically fast reduced row echelon form and nullspace
 - asymptotically fast nonsingular solving
 - asymptotically fast inverse
 - tidied some interfaces
 - fixed $n \times 0 \times m$ product not zeroing the result
 - added trace
 - made multiplication faster for tiny moduli by means of bit packing
- `nmod_poly_mat`
 - new module for matrices over $\mathbb{Z}/n\mathbb{Z}[x]$, with similar functionality as the `fmpz_poly_mat` module
 - determinant, rank, solving, reduced echelon form, nullspace

- fraction-free Gaussian elimination
- multiplication using bit packing
- multiplication using evaluation-interpolation
- determinant using evaluation-interpolation
- `padic`
 - restructured and improved much of the code
 - added `padic_log`
 - improved log and exp using rectangular splitting
 - added asymptotically fast log and exp based on binary splitting
- `perm`
 - added the `perm` module for permutation matrices
 - computing the parity of a permutation
 - inverting a permutation
- `arith`
 - added generation of cyclotomic polynomials
 - added functions for evaluating Dedekind sums
 - fast computation of the partition function
 - added a function for factoring a Hardy-Ramanujan-Rademacher type exponential sum
 - added Chebyshev polynomials `T` and `U`
 - added computation of the minimal polynomial of $\cos(2\pi/n)$
 - added asymptotically fast high-precision approximation of $\zeta(n)$
 - added asymptotically fast computation of Euler's constant
 - added new algorithms and functions for computing Bell numbers
 - fast computation of π (adapting code written by Hanhong Xue)
 - added functions for computing the number of sum of squares representations of an integer
 - renamed functions to have an `arith` prefix

2011-06-04 – FLINT 2.2

- `fmpq` (multiprecision rational numbers)
 - Basic arithmetic functions
 - Utility functions
 - Rational reconstruction
 - Functions for enumerating the rationals
- `fmpq_mat` (matrices over $\mathbb{Q}$)
 - Basic arithmetic functions
 - Utility functions
 - Fast multiplication
 - Classical and fraction-free reduced row echelon form
 - Determinants

- Fast non-singular solving
- `fmprz_poly_mat` (matrices over $\mathbb{Z}[x]$)
 - Basic arithmetic functions
 - Utility functions
 - Fraction-free row reduction and determinants
 - Fast determinants (experimental)
- `fmprz_mat`
 - Added more utility functions (scalar multiplication, etc)
 - Added Dixon's p-adic algorithm (used by fast nonsingular rational system solving)
 - Added reduced row echelon form
 - Added conversions between `fmprz_mat` and `nmod_mat`
 - Added CRT functions for `fmprz_mats`
 - Faster matrix multiplication for small to medium dimensions
- `longlong.h`
 - Added x86 assembly macros for accumulating sums of two limb operands
- `nmod_mat`
 - Sped up arithmetic for moduli close to `FLINT_BITS`
- `arith`
 - Changed interface of various functions to use new `fmprq` type
- `fmprz`
 - Added `fmprz_set_ui_mod`
 - Inlined `fmprz_neg`, `fmprz_set_si`, `fmprz_set_ui` for better performance
 - Added `fmprz_lcm`
 - Small performance improvement to `fmprz_CRT_ui`
- `fmprz_vec`
 - Added `_fmprz_vec_lcm`
- `fmprz_poly_q` (rational functions over $\mathbb{Q}$, modeled as quotients of `fmprz_polys`)
 - Basic arithmetic functions
 - Conversion and IO functions
 - Evaluation
- `padic` (p-adic numbers – experimental)
 - Basic arithmetic
 - Data conversion and IO
 - Inverse and square root using Newton iteration
 - Teichmuller lifts (not optimised)
 - p-adic exponential function (not optimised)
- `fmprz_poly`
 - Added `fmprz_poly_gcd_modular` (and `fmprz_poly_gcd` wrapper)
 - Added `fmprz_poly_xgcd_modular` (and `fmprz_poly_xgcd` wrapper)

- Added conversions between `fmpz_poly` and `nmod_poly`
- Added CRT functions
- Added multipoint evaluation and interpolation
- `nmod_poly`
 - Added `nmod_poly_xgcd_euclidean` (and `nmod_poly_xgcd` wrapper)
 - `nmod_poly_gcd` wrapper
- `mpn_extras`
 - Added `MPN_NORM` and `MPN_SWAP` macros.
 - Added `mpn_gcd_full` to remove some of the restrictions from the usual `mpn_gcd`
- build fixes
 - fixed make install to create nonexistent dirs (reported by Serge Torres)
 - `-L` use `/usr` instead of `/usr/local` by default (reported by Serge Torres)
 - guards for system headers because of flint's use of `ulong`

2011-03-09 – FLINT 2.1

- `fmpz`
 - Simplified interface for fast multimodular reduction and CRT reconstruction
 - Fixed segmentation fault in `fmpz_multi_mod_ui` when the input exceeds the product of the moduli
 - Added simple incremental CRT functions (`fmpz_CRT_ui`, `fmpz_CRT_ui_unsigned`) to complement the existing fast ones
 - Added example programs for CRT
 - Added random number generators designed for testing modular code (`fmpz_randtest_mod`, `fmpz_randtest_mod_signed`)
 - Added `fmpz_fdiv_ui` for remainder on division by an `ulong`
 - Added `fmpz_bin_uiui` for computing binomial coefficients
 - Added `fmpz_mul2_uiui` and `fmpz_divexact2_uiui` for multiplying or dividing an `fmpz` by a pair of `ulongs` (efficiently when their product fits in a single limb)
- `fmpz_mat`
 - Added utility functions for basic arithmetic and creating unit matrices
 - Added multimodular determinant computation (certified or heuristic)
 - Added support for computing right nullspaces (`fmpz_mat_kernel`). Fast only for small matrices.
 - Some internal code cleanup and various small fixes
- `nmod_mat`
 - Faster Gaussian elimination for small moduli
 - Faster determinants
 - Faster matrix inversion and nonsingular solving
- `nmod_poly`
 - Added `nmod_poly_integral` for computing integrals

- Added fast square root and inverse square root of power series
- Added fast transcendental functions of power series (log, exp, sin, cos, tan, sinh, cosh, tanh, asin, atan, asinh, atanh)
- Made `nmod_poly_inv_series_newton` more memory efficient
- `fmppq_poly`
 - Added `fmppq_poly_integral` for computing integrals
 - Added fast transcendental functions of power series (log, exp, sin, cos, tan, sinh, cosh, tanh, asin, atan, asinh, atanh)
- `arith`
 - Made computation of vectors of Bernoulli numbers faster
 - Added fast computation of single Bernoulli numbers
 - Added a separate function for computing denominators of Bernoulli numbers
 - Added fast computation of Bell numbers (vector and single)
 - Added fast computation of Euler numbers (vector and single)
 - Added fast computation of Euler polynomials
 - Added fast computation of Swinnerton-Dyer polynomials
 - Added fast computation of Legendre polynomials
 - Added fast vector computation of the partition function
 - Added fast vector computation of Landau's function
- `ulong_extras`
 - Added a function for computing factorials mod `n`
- `build system`
 - Added support for building static and shared libraries
 - All object files and `test/profile/example` binaries now build in separate build directory
- `documentation`
 - Large number of corrections

2011-01-16 – FLINT 2.0

N.B: FLINT 2 is a complete rewrite of flint from scratch It includes the following modules:

- `ulong_extras`: (word sized integers and modular arithmetic)
 - random numbers (`randint`, `randbits`, `randprime`, `randint`)
 - powering
 - reverse binary
 - `mod`, `divrem`, `mulmod` all with precomputed inverses
 - `gcd`, `invgcd`, `xgcd`
 - jacobi symbols
 - `addmod`, `submod`, `invmod`, `powmod`
 - prime sieve, `nextprime`, `prime-pi`, `nth-prime`
 - primality testing (`small`, binary search, Pocklington-Lehmer, Pseudosquare)

- probably prime tests (strong base-a, Fermat, Fibonacci, BPSW, Lucas)
- sqrt, sqrtrem, is-square, perfect-power (2,3,5)
- remove, is-squarefree
- factorisation (trial-range, trial, power (2,3,5), one-line, SQUFOF)
- Moebius mu, Euler phi
- fmpz: (memory managed multiple precision integers)
 - memory management (init, clear)
 - random numbers (randbits, randm)
 - conversion to and from long, ulong, doubles, mpz's, strings
 - read/write to file, stdin, stdout
 - sizeinbase, bits, size, sgn, swap, set, zero
 - cmp, cmp-ui, cmpabs, equal, is-zero, is-one
 - neg, abs, add, add-ui, sub, sub-ui, mul, mul-si, mul-ui, mul-2exp
 - addmul, addmul-ui, submul, submul-ui
 - cdiv-q, cdiv-q-si, cdiv-q-ui
 - fdiv-q, fdiv-q-si, fdiv-q-ui, fdiv-qr, fdiv-q-2exp
 - tdiv-q, tdiv-q-si
 - divexact, divexact-si, divexact-ui
 - mod, mod-ui
 - powering
 - sqrt, sqrt-rem
 - factorial
 - gcd, invmod
 - bit-pack, bit-unpack
 - multimodular reduction, CRT
- fmpz_vec: (vectors over fmpz's)
 - memory management (init, clear)
 - random vectors
 - max-bits, max-limbs
 - read/write to file/stdin/stdout
 - set, swap, zero, neg
 - equal, is-zero
 - sort
 - add, sub
 - scalar multiplication by fmpz, ulong, long, 2exp
 - exact division by fmpz, long, ulong
 - fdiv-q by fmpz, long, ulong, 2exp
 - tdiv-q by fmpz, long, ulong
 - addmul by fmpz, long, long by 2exp

- submul by fmpz, long, long by 2exp
- Gaussian content
- fmpz_poly: (polys over fmpz's)
 - memory management (init, realloc, fit-length, clear)
 - random polys
 - normalise, set-length, truncate
 - length, degree, max-limbs, max-bits
 - set, set-si, set-ui, set-fmpz, set-str
 - get-str, get-str-pretty
 - zero, one, zero-coeffs
 - swap, reverse
 - get/set coeffs from fmpz, long, ulong
 - get-coeff-ptr, lead
 - equal, is-zero
 - add, sub
 - scalar multiplication by fmpz, long, ulong
 - scalar addmul/submul by fmpz
 - scalar fdiv by fmpz, long, ulong
 - scalar tdiv by fmpz, long, ulong
 - scalar divexact by fmpz, long, ulong
 - bit pack, bit unpack
 - multiplication (classical, karatsuba, KS)
 - mullow (classical, karatsuba, KS)
 - mulhigh (classical, karatsuba)
 - middle product (classical)
 - powering (small, binary exponent, binomial, multinomial, addition chains)
 - truncated powering (binary exponent)
 - shift left/right
 - euclidean norm
 - gcd (subresultant)
 - resultant
 - content, primitive part
 - divrem (basecase, divide-and-conquer)
 - div (basecase, divide-and-conquer)
 - rem (basecase)
 - invert series (basecase, Newton)
 - div series
 - pseudo divrem (basecase, divide-and-conquer, Cohen)
 - rem (Cohen)

- div
- evaluate (Horner) at fmpz, mpq, a mod n
- composition (Horner, divide-and-conquer)
- signature
- read/write to file/stdin/stdout
- fmpz_poly: (polynomials over $\mathbb{Q}$ stored as poly over fmpz with fmpz denominator)
 - memory management (init, realloc, fit-length, clear)
 - random polys
 - set-length, canonicalise, normalise, truncate
 - is-canonical, length, degree
 - reference to numerator, denominator
 - set, set-si, set-ui, set-fmpz, set-mpz, set-mpq
 - set-array-mpq, set-str
 - get-str, get-str-pretty
 - zero, neg, swap
 - invert
 - set coefficient to mpq, long, ulong, fmpz, mpz
 - get coefficient as mpq
 - equal, cmp, is-one, is-zero
 - add, sub
 - scalar multiplication by long, ulong, fmpz, mpq
 - scalar division by fmpz, long, ulong, mpq
 - multiplication, mullo
 - powering
 - shift left/right
 - divrem, div, rem
 - invert series (Newton iteration)
 - divide series
 - derivative
 - evaluate at fmpz, mpq
 - composition, scale by constant
 - content, primitive part
 - make-monic, is-monic
 - is-squarefree
 - read/write to file/stdin/stdout
- nmod_vec: (vectors over $\mathbb{Z}/n\mathbb{Z}$ for n fitting in a machine word)
 - memory management (init/clear)
 - macros for efficient reduction of 1, 2 and 3 limb integers mod n
 - macro for addmul mod n

- add/sub/neg individual coefficients mod n
- random vectors
- set, zero, swap
- reduce, max-bits
- equal
- add, sub, neg
- scalar multiplication by a value reduced mod n
- scalar addmul by a value reduced mod n
- `nmod_poly`: (polynomials over $\mathbb{Z}/n\mathbb{Z}$ for n fitting in a machine word)
 - memory management (init, realloc, fit-length, clear)
 - random polys
 - normalise, truncate
 - length, degree, modulus, max-bits
 - set, swap, zero, reverse
 - get/set coefficients as ulongs, strings
 - read/write to file, stdin, stdout
 - equal, is-zero
 - shift left/right
 - add, sub, neg
 - scalar multiplication by a value reduced mod n
 - make-monic
 - bit pack, bit unpack
 - multiplication (classical, KS)
 - mullow (classical, KS)
 - mulhigh (classical)
 - powering (binary exponent)
 - pow-trunc (binary exponent)
 - divrem (basecase, divide-and-conquer, Newton iteration)
 - div (basecase, divide-and-conquer, Newton iteration)
 - invert series (basecase, Newton iteration)
 - divide series (Newton iteration)
 - derivative
 - evaluation at a value taken mod n
 - composition (Horner, divide-and-conquer)
 - gcd (euclidean)
- `fmpz_mat`: (matrices over `fmpz`'s)
 - memory management (init, clear)
 - random matrices (bits, integer relations, simultaneous diophantine equations NTRU-like, ajtai, permutation of rows and cols of a diagonal matrix, random of given rank, random of given determinant, random elementary operations)

- set, init-set, swap, entry pointer
 - write to file or stdout
 - equal
 - transpose
 - multiplication (classical, multimodular)
 - inverse
 - determinant
 - row reduce (Gaussian and Gauss-Jordan fraction-free elimination)
 - rank
 - solve $Ax = b$, solve $AX = B$
 - fraction free LU decomposition
- `nmod_mat`: (matrices over $\mathbb{Z}/n\mathbb{Z}$ for n fitting in a machine word)
 - memory management (init, clear)
 - random matrices (uniform, full, permutations of diagonal matrix, random of given rank, random elementary operations)
 - set, equal
 - print to stdout
 - add
 - transpose
 - multiplication (classical, Strassen, $A*B^T$)
 - row reduction (Gaussian and Gauss-Jordan)
 - determinant
 - rank
 - solve ($Ax = b$, $AX = B$, solve with precomputed LU)
 - invert
- `arith`: (arithmetic functions)
 - Bernoulli numbers
 - Bernoulli polynomials
 - primorials (product of primes up to n)
 - harmonic numbers
 - Stirling numbers
 - Euler phi function
 - Moebius mu function
 - Sigma (sum of powers of divisors)
 - Ramanujan tau function
- `examples`: (example programs)
 - compute coefficients of q-series of Delta function
- `mpfr_vec`: (vectors over mpfr reals)
 - memory management (init, clear)

- add
- set, zero
- scalar multiplication by mpfr, 2exp
- scalar product
- mpfr_mat: (matrices over mpfr reals)
 - memory management (init, clear)

2010-12-24 – FLINT 1.6.0

- Bugs:
 - Fixed a memory leak in mpz_poly_to_string_pretty
 - Fixed a bug inherited from an old version of fpLLL
 - Makefile to respect CC and CXX
 - Fixed bug in F_mpz_set_si
 - Fixed bug in F_mpz_equal
 - Most for loops to C90 standard (for easier MSVC porting)
 - Better Cygwin support
 - Fixed a bug in zmod_poly_resultant
 - Fixed bug in F_mpz_mul_KS/2
 - Fixed bug in tinyQS
 - Worked around some known bugs in older GMP/MPIR's
- Major new functionality
 - F_mpz_poly_factor_zassenhaus
 - F_mpz_poly_factor (incl. fmpz_poly_factor wrapper) using new vH-N approach (see the paper of van Hoeij and Novocin and the paper of van Hoeij, Novocin and Hart)
 - Implementation of new CLD bounds function for polynomial factors (see the paper of van Hoeij, Novocin and Hart)
 - Restartable Hensel lifting
 - Heuristic LLL implementations using doubles and mpfr
 - LLL implementations optimised for knapsack lattices
 - New (probably subquadratic) LLL implementation (ULLL)
 - zmod_poly_factor_cantor_zassenhaus
 - New F_mpz_mod_poly module for polynomials over $\mathbb{Z}/p\mathbb{Z}$ for multiprec. p
- Some of the other new functions added
 - F_mpz
 - F_mpz_gcd
 - F_mpz_smod
 - F_mpz_mod_preinv
 - F_mpz_fdiv_qr
 - F_mpz_get/set_mpfr/2exp

- `F_mpz_sscanf`
- `F_mpz_set_d`
- `F_mpz_poly`:
- read `F_mpz_poly` to_string/from_string/fprint/print/fread/pretty
- `F_mpz_poly_to/from_zmod_poly`
- `F_mpz_poly_scalar_div_exact`
- `F_mpz_poly_smod`
- `F_mpz_poly_derivative`, `F_mpz_poly_content`, `F_mpz_poly_eval_horner_d/2exp`
- `F_mpz_poly_scalar_abs`
- `F_mpz_poly_set_d_2exp`
- `F_mpz_poly_div/divrem`
- `F_mpz_poly_gcd`
- `F_mpz_poly_is_squarefree`
- `F_mpz_poly_factor_squarefree`
- `F_mpz_poly_mul_trunc_left`
- `F_mpz_poly_pseudo_div`
- `F_mpz_poly_set_coeff`
- `F_mpz_poly_pow_ui`
- Inflation/deflation trick for factorisation
- `zmod_poly`:
 - Inflation/deflation trick for factorisation
 - `mpz_mat`:
 - `mpz_mat_from_string/to_string/fprint/fread/pretty`
 - `mpq_mat`:
 - `mpq_mat_init/clear`
 - Gramm-schmidt Orthogonalisation
 - `F_mpz_mat`:
 - `F_mpz_mat_print/fprint/fread/pretty`
 - `F_mpz_mat_mul_classical`
 - `F_mpz_mat_max_bits/2`
 - `F_mpz_mat_scalar_mul/div_2exp`
 - `F_mpz_mat_col_equal`
 - `F_mpz_mat_smod`
 - `F_mpz_vec_scalar_product/norm`
 - `F_mpz_vec_add/submul_ui/si/F_mpz/2exp`
 - `zmod_mat`:
 - classical multiplication
 - strassen multiplication
 - scalar multiplication

- `zmod_mat_equal`
- `zmod_mat_add/sub`
- `zmod_mat_addmul_classical`
- `d_mat`:
- `d_vec_norm`, `d_vec_scalar_product`
- `mpfr_mat`:
- `mpfr_vec_scalar_product/norm`

2009-09-22 – FLINT 1.5.0

- Added multimodular reduction and CRT to `F_mpz` module
- Fixed some bugs in `F_mpz` module and numerous bugs in test code
- Added `zmod_poly_compose`
- Added `zmod_poly_evaluate`
- Added functions for reduced evaluation and composition to `fmpz_poly` module (contributed by Burcin Erocal)
- Fixed bugs in the primality tests in `long_extras`
- Removed all polynomial multimodular multiplication functions
- Added new thetaproduct code used in the 1 trillion triangles computation
- Fixed a severe bug in the `fmpz_poly_pseudo_div` function reported by Sebastian Pancratz
- Added `fmpz_comb_temp_init/clear` functions
- Fixed a normalisation buglet in `fmpz_poly_pack_bytes`
- Added `F_mpz_pow_ui` function (contributed by Andy Novocin)
- Fixed a severe bug in the FFT reported by William Stein and Mariah Lennox (fix contributed by David Harvey)
- Removed some memory leaks from `F_mpz` test code
- Fixed bug in `zmod_poly_evaluate` test code

2009-07-06 – FLINT 1.4.0

- Sped up `zmod_poly` division in case where operands are the same length
- Sped up `zmod_poly` division in case where operands have lengths differing by 1
- Fixed a bug in `zmod_poly_gcd` for polynomials of zero length
- Sped up `zmod_poly_gcd` considerably (both euclidean and half gcd)
- Sped up `zmod_poly_gcd_invert` and `zmod_poly_xgcd` considerably
- Made `zmod_poly_gcd_invert` and `zmod_poly_xgcd` asymptotically fast
- Made `zmod_poly_resultant` asymptotically fast
- Added optimised `zmod_poly_rem` function
- Fixed a divide by zero bug in `zmod_poly_factor_berlekamp`
- Added test code for `z_factor_tinyQS` and `z_factor_HOLF`
- Fixed many bugs in the `z_factor` code, `tinyQS` and `mpQS`

- Corrected numerous typos in the documentation and added missing descriptions
- Added `F_mpz_cmp` function
- Added documentation to the manual for the new `F_mpz` module

2009-06-09 – FLINT 1.3.0

- Added new code for checking 2nd, 3rd and 5th roots
- Fixed a bug in `z_factor`
- Connected quadratic sieve for factoring large ulongs
- Added one line factor algorithm
- constructed best of breed factor algorithm
- Fixed termination conditions for `z_intcuberoot` and `z_intfifthroot` which were broken
- Added some code for special cases which cause infinite loops in `cuberoot` and `fifthroot`
- Went back to `ceil(pow(n, 0.33333333))` and `ceil(pow(n, 0.2))` for initial guesses in `cube` and `fifthroot` functions as these were about 50% faster than `sqrt(n)` and `sqrt(sqrt(n))` respectively.
- Added test code for `z_intfifthroot`
- Added test code for `z_factor_235power`
- Fixed some uninitialised data found by `valgrind` in `intcuberoot` and `intfifthroot`
- Fixed multiply defined `PRIME_COUNT` in `long_extras-test`
- Got rid of `gotos` in some functions in `long_extras`
- Knocked optimisation level back to `-O2` because it miscompiles on `sage.math`
- Changed tables to use `uint64_t`'s instead of `ulongs` which are not 64 bits on a 32 bit machine
- Only checked `MAX_HOLF` on 64 bit machine
- Changed `MAX_SQUFOF` to `WORD(-1)`
- Check constant `0x3FFFFFFFFFUL` only on a 64 bit machine
- Fixed a bug in `z_oddprime_lt_4096` on 32 bit machines
- Fixed some TLS issues with Cygwin
- Fixed some typos in `makefile`
- Fixed a wrong path in `fmpz.c`

2009-04-18 – FLINT 1.2.5

- Upgraded to `zn_poly-0.9` to avoid a serious error in squaring of large polynomials over $\mathbb{Z}/n\mathbb{Z}$

2009-04-04 – FLINT 1.2.4

- Defined `THREAD` to be blank on Apple CC and `__thread` for thread local storage on other gcc's (where it's defined)
- `#undef` `ulong` in `profiler.h` where `time.h` and other system time headers are included (both reported by M. Abshoff)

2009-03-31 – FLINT 1.2.3

- Fixed bugs in all `fmpz_poly` evaluation functions, identified by Burcin Erocal.

2009-03-20 – FLINT 1.2.2

- Fixed a memory leak in `zmod_poly_factor`
- Fixed `zmod_poly-profile` build

2009-03-14 – FLINT 1.2.1

- Removed some FLINT 2.0 code which was interfering with the build of the NTL-interface
- Removed an `omp.h` from `fmpz_poly.c`.

2009-03-10 – FLINT 1.2.0

- made memory manager, `fmpz` and `fmpz_poly` threadsafe
- Code for running tests in parallel (not activated)
- Sped up `fmpz_poly_scalar_div_ui/si` when scalar is 1/-1
- Parallelise `_fmpz_poly_mul_modular`
- `fmpz_mul_modular_packed` to pack coefficients to the byte before running `_fmpz_poly_mul_modular`
- `fmpz_poly_pseudo_rem_cohen` (not documented)
- special case for leading coeff 1/-1 in `fmpz_poly_pseudo_divrem_basecase`
- removed a memory allocation bug which caused a massive slowdown in `fmpz_poly_pseudo_divrem_basecase`
- `fmpz_poly_pseudo_rem_basecase` (not documented)
- `fmpz_poly_pseudo_rem` (not asymptotically fast)
- `fmpz_poly_signature` (not asymptotically fast)
- basic `fmpz_poly_is_squarefree` function
- included `zn_poly` in trunk and made FLINT build `zn_poly` as part of its build process
- switched to using `zn_poly` for polynomial multiplication, newton inversion, scalar multiplication in `zmod_poly`
- Integer cube root of word sized integers
- Fibonacci pseudoprime test
- BSPW probable prime test
- $n - 1$ primality test
- Complete implementation of `z_issquarefree`

- Significantly improved the thetaproduct example program.
- Fixed bug in `fmpz_poly_byte_pack` which is triggered when trying to pack into fields a multiple of 8 bytes (could cause a segfault)
- Fixed a bug in `fmpz_poly_pseudo_divrem` (relied on an uninitialised poly to have length 0)
- Fixed bug in `fmpz_multi_CRT_ui` (could segfault)
- Fixed bug in `fmpz_multi_mod_ui` (could segfault)
- Fixed memory leak in `zmod_poly_factor_squarefree`
- Fixed memory leak in `zmod_poly_from_string`

2009-03-01 – FLINT 1.1.3

- Inserted some missing return values in `zmod_poly` test code.

2009-03-01 – FLINT 1.1.2

- Fixed some memory allocation slowdowns and bugs in `fmpz_poly` division and pseudo division functions (reported by William Stein).

2009-02-11 – FLINT 1.1.1

- Fixed bugs in `fmpz_poly_scalar_mul_fmpz`, `fmpz_poly_gcd_heuristic` and `fmpz_poly_gcd_subresultant` and fixed bugs in `test_fmpz_poly_scalar_div_fmpz`, `test_fmpz_poly_scalar_div_fmpz` and `test_fmpz_poly_scalar_div_mpz`.

2008-12-21 – FLINT 1.1.0

Some of the following features were previewed in FLINT 1.0.11.

- integer gcd (this just wraps the GMP gcd code)
- polynomial content
- primitive part
- convert to and from FLINT and NTL integers and polynomials
- get a coefficient of a polynomial efficiently as a read only `mpz_t`
- print polynomials in a prettified format with a specified variable
- Sped up integer multiplication
- Convert to and from `zmod_polys` from `fmpz_polys`
- Chinese remainder for `fmpz_polys`
- Leading coeff macro
- Euclidean norm of polynomials
- Exact division testing of polynomials
- Polynomial GCD (subresultant, heuristic, modular)
- Modular inversion of polynomials
- Resultant
- XGCD (Pohst-Zassenhaus)
- Multimodular polynomial multiplication

- Rewritten karatsuba_trunc function
- Rewritten division functions
- Polynomial derivative
- Polynomial evaluation
- Polynomial composition
- Addition and subtraction of zmod_polys
- Sped up multiplication of zmod_polys
- Extended multiplication of zmod_polys to allow up to 63 bit moduli
- zmod_poly subpolynomials
- zmod_poly reverse
- Truncated multiplication for zmod_polys (left, right, classical and KS)
- Scalar multiplication
- Division for zmod_polys (divrem and div, classical, divide and conquer and newton)
- Series inversion for zmod_polys
- Series division for zmod_polys
- Resultant for zmod_polys
- GCD for zmod_polys including half-gcd
- Inversion modulo a polynomial for zmod_polys
- XGCD for zmod_polys
- Squarefree factorisation for zmod_polys
- Berlekamp factorisation for zmod_polys
- Irreducibility testing for zmod_polys
- Derivative for zmod_polys
- Middle product for zmod_polys (sped up newton division)
- addmod, submod and divmod for ulongs
- Sped up limb sized integer square root
- Partial factoring of ulongs
- z_randbits
- Pocklington-Lehmer primality testing
- BSPW pseudo-primality testing
- Fermat pseudo-primality testing
- Fast Legendre symbol computation
- Chinese remainder for fmpz
- Square root with remainder for fmpz
- Left and right shift for fmpz
- Reduction modulo a ulong for fmpz
- Montgomery redc, mulmod, divmod and mod for fmpz
- Multimodular reduction and CRT for fmpz
- fmpz_mulmod and fmpz_divmod

- `fmpz_invert` for inversion modulo an `fmpz`
- Dramatically sped up `gcd` for small `fmpz`s
- Computation of 1D, 2D and some 3D theta functions
- Example program for multiplying theta functions
- Test code now times test functions
- Quick and dirty timing function for profiler
- Tiny quadratic sieve for small one and two limb integers
- Completely rewritten self initialising multiple polynomial quadratic sieve
- Build fix for 64 bit OSX dylibs (reported by Michael Abshoff)

2008-12-25 – FLINT 1.0.21

- Fixed the Christmas bug reported by Michael Abshoff which causes a test failure in `fmpz_poly_gcd_modular` and a hang in `fmpz_poly_invmod_modular` on 32 bit machines

2008-12-13 – FLINT 1.0.20

- Fixed some bugs in conversion of `zmod_poly`'s to and from strings

2008-12-12 – FLINT 1.0.19

- Fixed a bug in `z_remove_precomp`

2008-12-05 – FLINT 1.0.18

- Fixed another bug in the `fmpz_poly_set_coeff_*` functions which resulted in dirty coefficients

2008-11-30 – FLINT 1.0.17

- Fixed a segfault caused by left shifting of polynomials with zero limbs allocated in division and pseudo division functions.
- Fixed a bound used in `fmpz_gcd_modular` to use a proven bound
- Fixed a bug in `fmpz_poly-profile` where the top bit of random coefficients of `n` bits was always set

2008-10-22 – FLINT 1.0.16

- Fixed a segfault when trying to truncate a polynomial to an longer length than it currently is, with the function `fmpz_poly_truncate` (reported by Craig Citro)

2008-10-15 – FLINT 1.0.15

- Fixed a bug which causes a segfault when setting a coefficient of the zero polynomial to zero
- Fixed build bug in longlong.h on s390 platform

2008-09-23 – FLINT 1.0.14

- Update long_extras and test code for the sake of new quadratic sieve (new functions in long_extras remain undocumented)
- Removed many bugs from tinyQS and mpQS and joined them into a single program for factoring integers

2008-07-13 – FLINT 1.0.13

- Fixed memory leaks and dirty memory issues in test code for numerous modules.
- Removed further issues with cache prefetching in mpn_extras.c

2008-07-11 – FLINT 1.0.12

- Removed some Opteron tuning flags which cause illegal instruction errors on Pentium4
- Fixed numerous memory leaks in fmpz_poly test code
- Fixed memory leak in fmpz_poly_power_trunc_n
- Fixed major memory leaks in fmpz_poly_xgcd_modular
- Rewrote __fmpz_poly_mul_karatrunc_recursive and _fmpz_poly_mul_karatsuba_trunc to “prove code” and got rid of some dirty memory issues
- Fixed some potential illegal memory accesses to do with cache prefetching in fmpz_poly.c

2008-07-09 – FLINT 1.0.11

- Fixed a bug in z_ll_mod_precomp on ia64 (reported by Michael Abshoff and William Stein)

2008-06-16 – FLINT 1.0.10

- integer gcd (this just wraps the GMP gcd code)
- polynomial content
- convert to and from FLINT and NTL integers and polynomials
- get a coefficient of a polynomial efficiently as a read only mpz_t
- print polynomials in a prettified format with a specified variable

2008-03-11 – FLINT 1.0.9

- Fixed a memory allocation bug in `fmpz_poly_power`

2008-02-15 – FLINT 1.0.8

- Fixed a bug in `fmpz_poly_right_shift` (reported by Kiran Kedlaya)

2008-01-22 – FLINT 1.0.7

- Made `F_mpn_mul` binary compatible with the way `mpn_mul` *operates* in practice.

2008-01-17 – FLINT 1.0.6

- Fixed an issue with `FLINT_BIT_COUNT` on certain machines (probably due to arithmetic shift issues)

2008-01-05 – FLINT 1.0.5

- Fixed some inline issues which cause problems because of the C99 inline rules (reported by David Harvey).
- Fixed a makefile issue reported (and solved) by David Harvey when *not* linking against NTL.

2008-01-04 – FLINT 1.0.4

- Fixed a bug in the `bernoulli_zmod` example program and associated polynomial `zmod` code which caused memory corruption.
- Fixed a bug in the `fmpz-test` code which manifested on 32 bit machines, reported by David Harvey.
- Fixed some bugs in the pari profiling code.

2007-12-16 – FLINT 1.0.3

- Fixed a bug in the polynomial memory management code which caused a segfault
- Fixed a bug in the pseudo division code which caused a block overrun

2007-12-10 – FLINT 1.0.2

- Rewrote tuning code for integer multiplication functions, making it more robust and fixing a bug which showed up on 32 bit machines (reported by Michael Abshoff and Jaap Spies). Factored the tuning code out into a number of macros.

2007-12-07 – FLINT 1.0.1

- Fixed a bug in `_fmpz_poly_maxbits1` on 32 bit machines, reported by Michael Abshoff and Carl Witty
- Removed some instances of `u_int64_t` and replaced them with `uint64_t`, reported by Michael Abshoff
- Replaced `sys/types.h` with `stdint.h`
- Added FLINT macros to documentation
- Corrected numerous typos in documentation

2007-12-02 – FLINT 1.0

- First version of FLINT, includes `fmpz_poly`, `fmpz` and `mpQS`

16.1.2 Antic version history

2021-06-24 – Antic 0.2.5

- TODO: list changes here

2021-04-15 – Antic 0.2.4

- TODO: list changes here

2020-12-11 – Antic 0.2.3

- TODO: list changes here

2020-06-30 – Antic 0.2.2

- TODO: list changes here

2020-06-16 – Antic 0.2.1

- TODO: list changes here

2019-02-12 – Antic 0.2

- Many bug fixes, standalone build system, continuous integration.

2013-05-12 – Antic 0.1

- First version of antic, including a qfb module for (positive definite) binary quadratic forms.

16.1.3 Calcium version history

2021-05-28 – Calcium 0.4

- Algebraic numbers
 - Fixed bug in special-casing of roots of unity in qqbar_root_ui.
 - Fixed qqbar_randtest with bits == 1.
 - Faster qqbar_cmp_re for nearby reals.
 - Faster qqbar polynomial evaluation and powering using linear algebra.
 - Improved qqbar_abs, qqbar_abs2 to produce cleaner enclosures.
 - Use a slightly better method to detect real numbers in qqbar_sgn_im.
 - Added qqbar_hash.
 - Added qqbar_get_fmpq, qqbar_get_fmpz.
 - Added qqbar_pow_fmpq, qqbar_pow_fmpz, qqbar_pow_si.
 - Added qqbar_numerator, qqbar_denominator.
- Basic arithmetic and elementary functions
 - Improved ca_condense_field: automatically demote to a simple number field when the only used extension number is algebraic.
 - Improved multivariate field arithmetic to automatically remove algebraic or redundant monomial factors from denominators.
 - Added ca_pow_si_arithmetic (guaranteed in-field exponentiation).
 - Added polynomial evaluation functions (ca_fmpz_poly_evaluate, ca_fmpq_poly_evaluate, ca_fmpz_poly_evaluate, ca_fmpz_mpoly_q_evaluate).
 - Added several helper functions (ca_is_special, ca_is_qq_elem, ca_is_qq_elem_zero, ca_is_qq_elem_one, ca_is_qq_elem_integer, ca_is_nf_elem, ca_is_cyclotomic_nf_elem, ca_is_generic_elem).
 - Added ca_rewrite_complex_normal_form.
 - Perform direct complex conjugation in cyclotomic fields.
 - Use ca_get_acb_raw instead of ca_get_acb when printing to avoid expensive recomputations.
 - Added alternative algorithms for various basic functions.
 - Deep complex conjugation.
 - Use complex conjugation in is_real, is_imaginary, is_negative_real.
 - Added functions for unsafe inversion for internal use.
 - Significantly stronger zero testing in mixed algebraic-transcendental fields.
 - Added ca_arg.
 - Added special case for testing equality between number field elements and rationals.
 - Added ca_sin_cos, ca_sin, ca_cos, ca_tan and variants.
 - Added ca_atan, ca_asin, ca_acos and variants.

- Added `ca_csgn`.
- Improved `ca_get_acb` and `ca_get_acb_accurate_parts` to fall back on exact zero tests when direct numerical evaluation does not give precise enclosures.
- Added `ca_get_decimal_str`.
- More automatic simplifications of logarithms (simplify logarithms of exponentials, square roots and powers raised to integer powers).
- More automatic simplifications of square roots (simplify square roots of exponentials, square roots and powers raised to integer powers).
- Improved order comparisons (`ca_check_ge` etc.) to handle special values and to fall back on strong equality tests.
- Fixed a test failure in the `ca_mat` module.
- Polynomials
 - Added `ca_poly_inv_series`, `ca_poly_div_series` (power series division).
 - Added `ca_poly_exp_series` (power series exponential).
 - Added `ca_poly_log_series` (power series logarithm).
 - Added `ca_poly_atan_series` (power series arctangent).
- Other
 - Added `fmpz_mpoly_q_used_vars`.
 - Remove useless `rpath` line from `configure` (reported by Julien Puydt).
 - Added missing declaration of `fexpr_hash`.
 - Fixed crashes on OS X in Python interface (contributed by deinst).
 - Fixed memory leaks in Python string conversions (contributed by deinst).
 - Reserve I, E for symbolic expressions in Python interface.

2021-04-23 – Calcium 0.3

- Symbolic expressions
 - Added the `fexpr` module for flat-packed unevaluated symbolic expressions.
 - LaTeX output.
 - Basic manipulation (construction, replacement, accessing subexpressions).
 - Numerical evaluation with Arb.
 - Expanded normal form.
 - Conversion methods for other types.
 - Enable LaTeX rendering of objects in Jupyter notebooks.
- Algebraic numbers
 - Fix a major performance issue (slow root refinement) that made Calcium as a whole far slower than necessary.
 - Added `qqbar_cmp_root_order`; sort polynomial roots consistently by default.
 - Added `qqbar_get_quadratic`.
 - Added `qqbar_equal_fmpq_poly_val` and use it to speed up checking guessed values.

- Conversion of `qqbar_t` to and from symbolic expression (`qqbar_set_fexpr`, `qqbar_get_fexpr_repr`, `qqbar_get_fexpr_root_nearest`, `qqbar_get_fexpr_root_indexed`, `qqbar_get_fexpr_formula`).
- Fixed bugs in `qqbar_cmpabs_re`, `cmpabs_im`.
- Optimized `qqbar_cmp_im` and `qqbar_cmpabs_im` for conjugates with mirror symmetry.
- Added `qqbar_pow` (taking a `qqbar` exponent).
- Special-case roots of unity in `qqbar_pow_ui`, `qqbar_root_ui`, `qqbar_abs` and `qqbar_abs2`.
- Wrapped `qqbar` in Python.
- Polynomials
 - Added several utility functions.
 - Optimized polynomial multiplication with rational entries.
 - Fast polynomial multiplication over number fields.
- Matrices
 - Fast matrix multiplication over number fields.
 - Right kernel (`ca_mat_right_kernel`).
 - Matrix diagonalization (`ca_mat_diagonalization`).
 - Jordan normal form (`ca_mat_jordan_form`, `ca_mat_jordan_transformation`, `ca_mat_jordan_blocks`).
 - Matrix exponential (`ca_mat_exp`).
 - Matrix logarithm (`ca_mat_log`).
 - Polynomial evaluation (`ca_mat_ca_poly_evaluate`).
 - Cofactor expansion algorithm for determinant and adjugate (`ca_mat_adjugate_cofactor`).
 - Added several utility functions.
 - Improved algorithm selection in `ca_mat_inv`.
 - Solving using the adjugate matrix.
 - Danilevsky characteristic polynomial algorithm (`ca_mat_charpoly_danilevsky`).
- Field elements
 - Use factoring in `ca_sqrt` to enable more simplifications.
 - Simplify square roots and logarithms of negative real numbers.
 - Optimized `ca_sub`.
 - Conversion of `ca_t` to and from symbolic expressions (`ca_set_fexpr`, `ca_get_fexpr`).
 - Added function for assigning elements between context objects (`ca_transfer`).
 - Fixed a possible memory corruption bug when Vieta's formulas are used.
 - Optimized constructing square roots of rational numbers.
- Other
 - Added demonstration notebook to documentation.
 - Fixed OSX compatibility in Python wrapper (contributed by deinst).
 - Fixed bug in `calcium_write_acb`.
 - Fixed bug in `fmpz_mpoly_vec_set_primitive_unique` (contributed by gbunting).

2020-10-16 – Calcium 0.2

- Basic arithmetic and expression simplification
 - Use Gr bner basis for reduction ideals, making simplification much more robust.
 - Compute all linear relations with LLL simultaneously instead of piecemeal.
 - Make monomial ordering configurable (default is lex as before).
 - Use Vieta’s formulas to simplify expressions involving conjugate algebraic numbers.
 - Denest exponentials of symbolic logarithms.
 - Denest logarithms of symbolic powers and square roots.
 - Denest powers of symbolic powers.
 - Simplify exponentials that evaluate to roots of unity.
 - Simplify logarithms of roots of unity.
 - Improve ideal reduction to avoid some unnecessary GCD computations.
- Python wrapper
 - Calcium now includes a minimal ctypes-based Python wrapper for testing.
- New `ca_mat` module for matrices
 - Mostly using naive basecase algorithms.
 - Matrix arithmetic, basic manipulation.
 - Construction of special matrices (Hilbert, Pascal, Stirling, DFT).
 - LU factorization.
 - Fraction-free LU decomposition.
 - Nonsingular solving and inverse.
 - Reduced row echelon form.
 - Rank.
 - Trace and determinant.
 - Characteristic polynomial.
 - Computation of eigenvalues with multiplicities.
- New `ca_poly` module for polynomials
 - Mostly using naive basecase algorithms.
 - Polynomial arithmetic, basic manipulation.
 - Polynomial division.
 - Evaluation and composition.
 - Derivative and integral.
 - GCD (Euclidean algorithm).
 - Squarefree factorization.
 - Computation of roots with multiplicities.
 - Construction from given roots.
- New `ca_vec` module for vectors.
 - Memory management and basic scalar operations.
- Bug fixes

- Fix bug in powering number field elements.
 - Fix bug in `qqbar_log_pi_i`.
 - Fix aliasing bug in `ca_pow`.
- New basic functions
 - Conversion from double: `ca_set_d`, `ca_set_d_d`.
 - Special functions: `ca_erf`, `ca_erfi`, `ca_erfc`, with algebraic relations.
 - Special functions: `ca_gamma` (incomplete simplification algorithms).
- New `utils_flint` module for FLINT utilities
 - Vectors of multivariate polynomials.
 - Construction of elementary symmetric polynomials.
 - Gröbner basis computation (naive Buchberger algorithm).
- Documentation and presentation
 - Various improvements to the documentation.
 - DFT example program.

2020-09-08 – Calcium 0.1

- Initial test release.
- `ca` module (exact real and complex numbers).
- `fmpz_mpoly_q` module (multivariate rational functions over $\mathbb{Q}$).
- `qqbar` module (algebraic numbers represented by minimal polynomials).
- Example programs.

16.1.4 Arb version history

2022-06-29 – Arb 2.23.0

- Performance
 - Multithreaded numerical integration.
 - Multithreaded binary splitting computation of mathematical constants.
 - Multithreaded computation of Bernoulli numbers.
 - Multithreaded computation of Euler numbers.
 - Multithreaded refinement of Riemann zeta zeros.
 - Multithreaded `complex_plot` example program.
 - Multithreaded elementary functions.
 - Multithreaded computation of Hilbert class polynomials.
 - Improved multithreaded partition function.
 - Use FLINT's FFT multiplication instead of GMP in appropriate ranges.
 - New, faster algorithm for elementary functions between roughly 10^3 and 10^6 digits.
 - Faster computation of log using Newton-like iteration instead of using MPFR.
 - Faster computation of atan using Newton-like iteration instead of the bit-burst algorithm.

- Fix performance bug in `atan()` leading to quadratic running time with large arguments in high precision.
- Optimized high-precision complex squaring.
- Added internal function `arb_flint_get_num_available_threads()` to improve tuning for multithreaded algorithms
- Fixed performance bug making `erf()` slower at high precision with multiple threads.
- Features
 - Implemented the Lerch transcendent (`acb_dirichlet_lerch_phi()`).
 - `fpwrap` wrapper for Lerch transcendent (contributed by Valentin Boettcher).
 - Added a rudimentary module for Gaussian integers (`fmpz.h`).
 - Added `zeta_zeros` example program (contributed by D.H.J. Polymath).
 - Added functions for simultaneous high-precision computation of logarithms of primes and arctangents for primitive angles.
 - Added `bernoulli`, `class_poly`, `functions_benchmark` example programs for benchmarking use.
 - Multiplying a signed number by an infinity yields an infinity instead of $[0 +/- \infty]$ (contributed by Erik Postma).
- Miscellaneous
 - Deprecated doubles version of partition function.
 - Fix crash in `erf` on some systems including `mips64el` (reported by Julien Puydt).
 - Fixed MINGW64 build (contributed by Massoud Mazar).
 - Avoid deprecated FLINT function `n_gcd_full`.
 - Documentation fixes.

2022-01-25 – Arb 2.22.1

- Fixed bugs causing some hypergeometric functions hang or crash for some input on various non-x86 architectures.
- Fixed a minor bug in `acb_hypgeom_m` (NaN result sometimes only set the real part to NaN).

2022-01-15 – Arb 2.22.0

- Special functions
 - Use numerical integration in some cases to compute the hypergeometric functions $0F1$, $1F1$, U , $2F1$, incomplete gamma and beta, modified Bessel, etc. with real parameters and argument, improving performance and accuracy when the parameters are large.
 - Much faster computation of Bernoulli numbers using hybrid numerical-modular algorithm (modular code adapted from `bernmm` by David Harvey).
 - Faster computation of Euler numbers using hybrid algorithm; added `arb_fmpz_euler_number_ui`.
 - Added inverse error function (`arb_hypgeom_erfinv`, `arb_hypgeom_erfcinv`).
 - New (faster, more accurate) implementations of real error functions (`arb_hypgeom_erf`, `arb_hypgeom_erfc`) and trigonometric integrals (`arb_hypgeom_si`, `arb_hypgeom_ci`).
 - Added `acb_dirichlet_l_fmpq` and `acb_dirichlet_l_fmpq_afe`: reduced-complexity evaluation of L-functions at rational points.

- Added functions for computing primorials (`arb_primorial_ui`, `arb_primorial_nth_ui`).
- New, highly optimized internal code for real hypergeometric series (`arb_hypgeom_sum_fmpq_arb`, etc.; currently only used in some functions).
- Fix `arb_fpwrap_double_hypgeom_2f1` which computed the wrong thing.
- Core arithmetic and functions
 - Faster implementation of `arb_ui_pow_ui`.
 - Added `arb_fma_si`, `arb_fma_fmpz`.
 - Added `arf_equal_ui`, `arf_equal_d`.
 - Added `arf_get_str`.
 - Use arb-based printing code instead of MPFR in `arf_printd` and `mag_printd` so that large exponents work.
 - Fixed bug in `arb_get_str` that caused loss of precision when printing more than about 10^6 digits.
 - Allow negative exponents in `mag_pow_fmpz`.
 - Added the `double_interval` module for fast machine-precision interval arithmetic (experimental, intended for internal use).

2021-10-20 – Arb 2.21.1

- Fixed 32-bit test failures for `arb_hypgeom_gamma_fmpq`.
- Added `pow` function to the `fpwrap` module.
- Added missing header file includes.
- Do not encode the library version in the SONAME on Android (contributed by Andreas Enge).

2021-09-25 – Arb 2.21.0

- Experimental new `arb_fpwrap` module: accurate floating-point wrappers of Arb mathematical functions (supersedes the external `arbcmath.h`).
- Fixed memory leak in `arf_load_file` (reported by Dave Platt).
- New and faster gamma function code.
- Most gamma function internals are now located in the `arb_hypgeom` and `acb_hypgeom` modules. The user-facing functions (`arb_gamma`, etc.) are still available under the old names for compatibility. The internal algorithms for rising factorials (binary splitting, etc.) have been moved without aliases.
- Added `arb_fma`, `arb_fma_arf`, `arb_fma_ui` (like `addmul`, but take a separate input and output).
- Slightly faster internal Bernoulli number generation for small n .
- Better enclosures for `acb_barnes_g` at negative reals.
- Added Graeffe transforms (`arb_poly_graeffe_transform`, `acb_poly_graeffe_transform`) (contributed by Matthias Gessinger).
- Fixed conflict with musl libc (reported by Gonzalo Tornaría).
- Added `acb_add_error_arb` (contributed by Albin Ahlbäck).

2021-07-25 – Arb 2.20.0

- FLINT 2.8 support.
- Change `arb_get_str` with `ARB_STR_NO_RADIUS`: `[+/- 1.20e-15]` now prints as `0e-14`.
- Uniformly distributed random number functions `arf_urandom`, `arb_urandom` (contributed by Albin Ahlbäck).
- Use quasilinear algorithm in `arb_gamma_fmpz` for all small fractions.
- Added derivative of Weierstrass elliptic function (`acb_elliptic_p_prime`) (contributed by Daniel Schultz).
- Added dot products with integer coefficients: `arb_dot_fmpz`, `arb_dot_siui`, `arb_dot_uiui`, `arb_dot_si`, `arb_dot_ui`, `acb_dot_fmpz`, `acb_dot_siui`, `acb_dot_uiui`, `acb_dot_si`, `acb_dot_ui`.
- Faster `arb_fmpz_poly_evaluate_arb` and `arb_fmpz_poly_evaluate_acb`.
- Explicitly guarantee that roots are isolated in `arb_fmpz_poly_complex_roots` (could previously theoretically fail when using the deflation hack).
- Use `GNUInstallDirs` in `CMakeLists.txt` to support standard GNU installation directories (contributed by Michael Orlitzky).
- Fixed bug for aliased multiplication of window matrices (contributed by David Berghaus).
- Documentation fixes (contributed by Joel Dahne, Hanno Rein).

2020-12-06 – Arb 2.19.0

- Significant improvements to the implementation of Platt's algorithm for computing Riemann zeta function zeros at large height (contributed by p15-git-acc).
- Better criterion for selecting asymptotic expansion of incomplete gamma function (contributed by p15-git-acc).
- Multithreaded `acb_dft` for power-of-two lengths (contributed by p15-git-acc).
- Added `acb_csc_pi`, `arb_csc_pi` (contributed by p15-git-acc).
- Fixed segfault in `acb_mat_eig_simple_rump` when called with `L` non-NULL and `R` NULL (contributed by p15-git-acc).
- Fixed bug in `acb_real_abs` (contributed by Joel Dahne).
- Changed several functions to more consistently return infinities instead of NaNs where reasonable (contributed by p15-git-acc).
- Added Fransen-Robinson as an integral example (contributed by p15-git-acc).
- Cleaned up makefile (contributed by p15-git-acc).
- Fixed several typos and some omitted functions in the documentation (contributed by Joel-Dahne, p15-git-acc).

2020-06-25 – Arb 2.18.1

- Support MinGW64.
- Added version numbers (`__ARB_VERSION`, `__ARB_RELEASE`, `ARB_VERSION`) to `arb.h`.

2020-06-09 – Arb 2.18.0

- General
 - FLINT 2.6 support.
 - Several build system improvements (contributed by Isuru Fernando).
 - Changed `arf_get_mpf` to return an MPFR underflow/overflow result (rounding to 0 or infinity with the right sign and MPFR overflow flags) instead of throwing `flint_abort()` if the exponent is out of bounds for MPFR.
 - Documentation and type corrections (contributed by Joel Dahne).
- Arithmetic
 - The number of iterations per precision level in `arb_fmpz_poly_complex_roots` has been tweaked to avoid extreme slowdown for some polynomials with closely clustered roots.
 - Added `arb_contains_interior`, `acb_contains_interior`.
- Special functions
 - Fixed unsafe shifts causing Dirichlet characters for certain moduli exceeding 32 bits to crash.
 - Added `acb_agm` for computing the arithmetic-geometric mean of two complex numbers.
 - `acb_elliptic_rj` now uses a slow fallback algorithm in cases where Carlson's algorithm is not known to be valid. This fixes instances where `acb_elliptic_pi`, `acb_elliptic_pi_inc` and `acb_elliptic_rj` previously ended up on the wrong branch. Users should be cautioned that the new version can give worse enclosures and sometimes fails to converge in some cases where the old algorithm did (the `pi` flag for `acb_elliptic_pi_inc` is useful as a workaround).
 - Optimized some special cases in `acb_hurwitz_zeta`.

2019-10-16 – Arb 2.17.0

- General
 - Added exact serialization methods (`arb_dump_str`, `arb_load_str`, `arb_dump_file`, `arb_load_file`, `arf_dump_str`, `arf_load_str`, `arf_dump_file`, `arf_load_file`, `mag_dump_str`, `mag_load_str`, `mag_dump_file`, `mag_load_file`) (contributed by Julian R  th).
 - Removed many obsolete `fmpz` methods and de-inlined several helper functions to slightly improve compile time and library size.
 - Fixed a namespace clash for an internal function (contributed by Julian R  th).
 - Added the helper function `arb_sgn_nonzero`.
 - Added the helper function `acb_rel_one_accuracy_bits`.
- Riemann zeta function
 - Added a function for efficiently computing individual zeros of the Riemann zeta function using Turing's method (`acb_dirichlet_zeta_zero`) (contributed by D.H.J. Polymath).
 - Added a function for counting zeros of the Riemann zeta function up to given height using Turing's method (`acb_dirichlet_zeta_nzeros`) (contributed by D.H.J. Polymath).
 - Added the Backlund S function (`acb_dirichlet_backlund_s`).

- Added a function for computing Gram points (`acb_dirichlet_gram_point`).
- Added `acb_dirichlet_zeta_deriv_bound` for quickly bounding the derivative of the Riemann zeta function.
- Fast multi-evaluation of the Riemann zeta function using Platt’s algorithm (`acb_dirichlet_platt_multieval`) (contributed by D.H.J. Polymath).
- Other special functions
 - Improved the algorithm in `acb_hypgeom_u` to estimate precision loss more accurately.
 - Implemented Coulomb wave functions (`acb_hypgeom_coulomb`, `acb_hypgeom_coulomb_series` and other functions).
 - Faster algorithm for Catalan’s constant.
 - Added `acb_modular_theta_series`.
 - Added `arb_poly_sinc_pi_series` (contributed by D.H.J. Polymath).
 - Improved tuning in `acb_hypgeom_pfq_series_sum` for higher derivatives at high precision (reported by Mark Watkins).

2018-12-07 – Arb 2.16.0

- Linear algebra and arithmetic
 - Added `acb_mat_approx_eig_qr` for approximate computation of eigenvalues and eigenvectors of complex matrices.
 - Added `acb_mat_eig_enclosure_rump` implementing Rump’s algorithm for certification of eigenvalue-eigenvector pairs as well as clusters.
 - Added `acb_mat_eig_simple_rump` for certified diagonalization of matrices with simple eigenvalues.
 - Added `acb_mat_eig_simple_vdhoeven_mourrain`, `acb_mat_eig_simple` for fast certified diagonalization of matrices with simple eigenvalues.
 - Added `acb_mat_eig_multiple_rump`, `acb_mat_eig_multiple` for certified computation of eigenvalues with possible overlap.
 - Added `acb_mat_eig_global_enclosure` for fast global inclusion of eigenvalues without isolation.
 - Added `arb_mat_companion`, `acb_mat_companion` for constructing companion matrices.
 - Added several `arb_mat` and `acb_mat` helper functions: `indeterminate`, `is_exact`, `is_zero`, `is_finite`, `is_triu`, `is_tril`, `is_diag`, `diag_prod`.
 - Added `arb_mat_approx_inv`, `acb_mat_approx_inv`.
 - Optimized `arb_mat_mul_block` by using `arb_dot` when the blocks are small.
 - Added `acb_get_mid`.
 - Updated `hilbert_matrix` example program.

2018-10-25 – Arb 2.15.1

- Fixed precision issue leading to spurious NaN results in incomplete elliptic integrals

2018-09-18 – Arb 2.15.0

- Arithmetic
 - Added `arb_dot` and `acb_dot` for efficient evaluation of dot products.
 - Added `arb_approx_dot` and `acb_approx_dot` for efficient evaluation of dot products without error bounds.
 - Converted loops to `arb_dot` and `acb_dot` in the `arb_poly` and `acb_poly` methods `mul_low_classical`, `inv_series`, `div_series`, `exp_series_basecase`, `sin_cos_series_basecase`, `sinh_cosh_series_basecase`, `evaluate_rectangular`, `evaluate2_rectangular`, `revert_series_lagrange_fast`. Also changed the algorithm cutoffs for `mul_low`, `exp_series`, `sin_cos_series`, `sinh_cosh_series`.
 - Converted loops to `arb_dot` and `acb_dot` in the `arb_mat` and `acb_mat` methods `mul_classical`, `mul_threaded`, `solve_tril`, `solve_triu`, `charpoly`. Also changed the algorithm cutoffs for `mul`, `solve_tril`, `solve_triu`.
 - Converted loops to `arb_approx_dot` and `acb_approx_dot` in the `arb_mat` and `acb_mat` methods `approx_solve_tril`, `approx_solve_triu`. Also changed the algorithm cutoffs.
 - Added `arb_mat_approx_mul` and `acb_mat_approx_mul` for matrix multiplication without error bounds.
- Miscellaneous
 - Added `arb_hypgeom_airy_zero` for computing zeros of Airy functions.
 - Added `arb_hypgeom_dilog` wrapper.
 - Optimized `arb_const_pi` and `arb_const_log2` by using a static table at low precision, giving a small speedup and avoiding common recomputation when starting threads.
 - Optimized `mag_set_ui_2exp_si`.
 - Remove obsolete and unused function `_arb_vec_dot`.
 - Converted some inline functions to ordinary functions to reduce library size.
 - Fixed `acb_dirichlet_stieltjes` to use the integration algorithm also when $a \neq 1$.
 - Fixed test failure for `acb_dirichlet_stieltjes` on ARM64 (reported by Gianfranco Costamagna). Special thanks to Julien Puydt for assistance with debugging.
 - Fixed crash in `acb_dft_bluestein` with zero length (reported by Gianfranco Costamagna).

2018-07-22 – Arb 2.14.0

- Linear algebra
 - Faster and more accurate real matrix multiplication using block decomposition, scaling, and multiplying via FLINT integer matrices in combination with safe use of doubles for radius matrix multiplications.
 - Faster and more accurate complex matrix multiplication by reordering and taking advantage of real matrix multiplication.
 - The new multiplication algorithm methods (`arb_mat_mul_block`, `acb_mat_mul_reorder`) are used automatically by the main multiplication methods.

- Faster and more accurate LU factorization by using a block recursive algorithm that takes advantage of matrix multiplication. Added separate algorithm methods `(arb/acb)_mat_lu_(recursive/classical)` with an automatic algorithm choice in the default `lu` methods.
- Added methods `(arb/acb)_mat_solve_(tril/triu)` (and variants) for solving upper or lower triangular systems using a block recursive algorithm taking advantage of matrix multiplication.
- Improved linear solving and inverse for large well-conditioned matrices by using a preconditioning algorithm. Added separate solving algorithm methods `(arb/acb)_mat_solve_(lu/precond)` with an automatic algorithm choice in the default `solve` methods (contributed by anonymous user `arbguest`).
- Improved determinants using a preconditioning algorithm. Added separate determinant algorithm methods `(arb/acb)_mat_det_(lu/precond)` with an automatic algorithm choice in the default `det` methods.
- Added automatic detection of triangular matrices in `arb_mat_det` and `acb_mat_det`.
- Added `arb_mat_solve_preapprox` which allows certifying a precomputed approximate solution (contributed by anonymous user `arbguest`).
- Added methods for constructing various useful test matrices: `arb_mat_ones`, `arb_mat_hilbert`, `arb_mat_pascal`, `arb_mat_stirling`, `arb_mat_dct`, `acb_mat_ones`, `acb_mat_dft`.
- Added support for window matrices `(arb/acb)_mat_window_init/clear`.
- Changed random test matrix generation `(arb/acb)_mat_randtest` to produce sparse matrices with higher probability.
- Added `acb_mat_conjugate` and `acb_mat_conjugate_transpose`.
- Arithmetic and elementary functions
 - Improved `arb_sin_cos`, `arb_sin` and `arb_cos` to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.
 - Improved `arb_sinh_cosh`, `arb_sinh` and `arb_cosh` to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.
 - Improved `arb_exp_invexp` and `arb_expm1` to produce more accurate enclosures for wide input intervals. The working precision is also reduced automatically based on the accuracy of the input to improve efficiency.
 - Improved `acb_rsqr` to produce more accurate enclosures for wide intervals.
 - Made `mag_add_ui_lower` public.
 - Added `mag_sinh`, `mag_cosh`, `mag_sinh_lower`, `mag_cosh_lower`.
 - Fixed minor precision loss near -1 in `arb_log_hypot` and `acb_log`.
 - Return imaginary numbers with exact zero real part when possible in `acb_acos` and `acb_acosh` (contributed by Ralf Stephan).
 - Improved special cases in `arb_set_interval_arf` (reported by Marc Mezzarobba).
- Special functions
 - Added a function for computing isolated generalized Stieltjes constants (`acb_dirichlet_stieltjes`).
 - Added scaled versions of Bessel functions (`acb_hypgeom_bessel_i_scaled`, `acb_hypgeom_bessel_k_scaled`).
 - The interface for the internal methods computing Bessel functions (`i_asymp`, `k_asymp`, etc.) has been changed to accommodate computing scaled versions.

- Added Riemann xi function (`acb_dirichlet_xi`) (contributed by D.H.J Polymath).
- Fixed infinite error bounds in the Riemann zeta function when evaluating at a ball containing zero centered in the left plane (contributed by D.H.J Polymath).
- Fixed precision loss in Airy functions with huge input and high precision.
- Legendre functions of the first kind (`legendre_p`): handle inexact integer $a+b-c$ in 2F1 better (contributed by Joel Dahne).
- Example programs and documentation
 - Added more color functions to `complex_plot.c`.
 - Added more example integrals suggested by Nicolas Brisebarre and Bruno Salvy to `integrals.c`
 - Changed Sphinx style and redesigned the documentation front page.
 - Miscellaneous documentation cleanups.
 - Added documentation page about contributing.
- Other
 - Fixed a crash on some systems when calling `acb_dft` methods with a length of zero.
 - Fixed issue with setting `rpath` in `configure` (contributed by Vincent Delecroix).

2018-02-23 – Arb 2.13.0

- Major bugs
 - Fixed rounding direction in `arb_get_abs_lbound_arf()` which in some cases could result in an invalid lower bound being returned, and added forgotten test code for this and related functions (reported by deinst). Although this bug could lead to incorrect results, it probably had limited impact in practice (explaining why it was not caught indirectly by other test code) since a single rounding in the wrong direction in this operation generally will be dwarfed by multiple roundings in the correct direction in surrounding operations.
- Important notes about bounds
 - Many functions have been modified to compute tighter enclosures when the input balls are wide. In most cases the bounds should be improved, but there may be some regressions. Bug reports about any significant regressions are welcome.
 - Division by zero in `arb_div()` has been changed to return `[NaN +/- inf]` instead of `[+/- inf]`. This change might be reverted in the future if it proves to be too inconvenient. In either case, users should only assume that division by zero produces something non-finite, and user code that depends on division by zero to produce `[0 +/- inf]` should be modified to handle zero-containing denominators as a separate case.
- Improvements to arithmetic and elementary functions
 - Faster implementation of `acb_get_mag_lower()`.
 - Optimized `arb_get_mag_lower()`, `arb_get_mag_lower_nonnegative()`.
 - Added `arb_set_interval_mag()` and `arb_set_interval_neg_pos_mag()` for constructing an `arb_t` from a pair of `mag_t` endpoints.
 - Added `mag_const_pi_lower()`, `mag_atan()`, `mag_atan_lower()`.
 - Added `mag_div_lower()`, `mag_inv()`, `mag_inv_lower()`.
 - Added `mag_sqrt_lower()` and `mag_rsqrt_lower()`.
 - Added `mag_log()`, `mag_log_lower()`, `mag_neg_log()`, `mag_neg_log_lower()`.
 - Added `mag_exp_lower()`, `mag_expinv_lower()` and tweaked `mag_exp()`.

- Added `mag_pow_fmpz_lower()`, `mag_get_fmpz()`, `mag_get_fmpz_lower()`.
- Improved `arb_exp()` for wide input.
- Improved `arb_log()` for wide input.
- Improved `arb_sqrt()` for wide input.
- Improved `arb_rsqrtd()` for wide input.
- Improved `arb_div()` for wide input.
- Improved `arb_atan()` for wide input and slightly optimized `arb_atan2()` for input spanning multiple signs.
- Improved `acb_rsqrtd()` for wide input and improved stability of this function generally in the left half plane.
- Added `arb_log_hypot()` and improved `acb_log()` for wide input.
- Slightly optimized trigonometric functions (`acb_sin()`, `acb_sin_pi()`, `acb_cos()`, `acb_cos_pi()`, `acb_sin_cos()`, `acb_sin_cos_pi()`) for pure real or imaginary input.
- Special functions
 - Slightly improved bounds for gamma function (`arb_gamma()`, `acb_gamma()`, `arb_rgamma()`, `acb_rgamma()`) for wide input.
 - Improved bounds for Airy functions for wide input.
 - Simplifications to code for computing Gauss period minimal polynomials (contributed by Jean-Pierre Flori).
 - Optimized `arb_hypgeom_legendre_p_ui()` further by avoiding divisions in the basecase recurrence and computing the prefactor more quickly in the asymptotic series (contributed by Marc Mezzarobba).
 - Small further optimization of `arb_hypgeom_legendre_p_ui_root()` (contributed by Marc Mezzarobba).
 - Improved derivative bounds for Legendre polynomials (contributed by Marc Mezzarobba).
- Numerical integration
 - Increased default quadrature `deg_limit` at low precision to improve performance for integration of functions without singularities near the path.
 - Added several more integrals to `examples/integrals.c`
 - Added utility functions `acb_real_abs()`, `acb_real_sgn()`, `acb_real_heaviside()`, `acb_real_floor()`, `acb_real_ceil()`, `acb_real_min()`, `acb_real_max()`, `acb_real_sqrtpos()`, useful for numerical integration.
 - Added utility functions `acb_sqrt_analytic()`, `acb_rsqrtd_analytic()`, `acb_log_analytic()`, `acb_pow_analytic()` with branch cut detection, useful for numerical integration.
- Build system and compatibility issues
 - Removed `-Wl` flag from `Makefile.subdirs` to fix “-r and -pie may not be used together” compilation error on some newer Linux distributions (reported by many users).
 - Fixed broken test code for `l_vec_hurwitz` which resulted in spurious failures on 32-bit systems (originally reported by Thierry Monteil on Sage trac).
 - Avoid using deprecated MPFR function `mpfr_root()` with MPFR versions $\geq 4.0.0$.
 - Remark: the recently released MPFR 4.0.0 has a bug in `mpfr_div()` leading to test failures in Arb (though not affecting correctness of Arb itself). Users should make sure to install the patched version MPFR 4.0.1.
 - Added missing C++ include guards in `arb_fmpz_poly.h` and `dlog.h` (reported by Marc Mezzarobba).

- Fixed Travis builds on Mac OS again (contributed by Isuru Fernando).
- Added missing declaration for `arb_bell_ui()` (reported by numsys).

2017-11-29 – Arb 2.12.0

- Numerical integration
 - Added a new function (`acb_calc_integrate`) for rigorous numerical integration using adaptive subdivision and Gauss-Legendre quadrature. This largely obsoletes the old integration code using Taylor series.
 - Added new `integrals.c` example program (old example program moved to `integrals_taylor.c`).
- Discrete Fourier transforms
 - Added `acb_dft` module with various FFT algorithm implementations, including top level $O(n \log n)$ `acb_dft` and `acb_dft_inverse` functions (contributed by Pascal Molin).
- Legendre polynomials
 - Added `arb_hypgeom_legendre_p_ui` for fast and accurate evaluation of Legendre polynomials. This is also used automatically by the Legendre functions, where it is substantially faster and gives better error bounds than the generic algorithm.
 - Added `arb_hypgeom_legendre_p_ui_root` for fast computation of Legendre polynomial roots and Gauss-Legendre quadrature nodes (used internally by the new integration code).
 - Added `arb_hypgeom_central_bin_ui` for fast computation of central binomial coefficients (used internally for Legendre polynomials).
- Dirichlet L-functions and zeta functions
 - Fixed a bug in the Riemann zeta function involving a too small error bound in the implementation of the Riemann-Siegel formula for inexact input. This bug could result in a too small enclosure when evaluating the Riemann zeta function at an argument of large imaginary height without also computing derivatives, if the input interval was very wide.
 - Add `acb_dirichlet_zeta_jet`; also made computation of the first derivative of Riemann zeta function use the Riemann-Siegel formula where appropriate.
 - Added `acb_dirichlet_l_vec_hurwitz` for fast simultaneous evaluation of Dirichlet L-functions for multiple characters using Hurwitz zeta function and FFT (contributed by Pascal Molin).
 - Simplified interface for using `hurwitz_precomp` functions.
 - Added `lcentral.c` example program (contributed by Pascal Molin).
 - Improved error bounds when evaluating Dirichlet L-functions using Euler product.
- Elementary functions
 - Faster custom implementation of `sin`, `cos` at 4600 bits and above instead of using MPFR (30-40% asymptotic improvement, up to a factor two speedup).
 - Faster code for `exp` between 4600 and 19000 bits.
 - Improved error bounds for `acb_atan` by using derivative.
 - Improved error bounds for `arb_sinh_cosh`, `arb_sinh` and `arb_cosh` when the input has a small midpoint and large radius.
 - Added reciprocal trigonometric and hyperbolic functions (`arb_sec`, `arb_csc`, `arb_sech`, `arb_csch`, `acb_sec`, `acb_csc`, `acb_sech`, `acb_csch`).
 - Changed the interface of `_acb_vec_unit_roots` to take an extra length parameter (compatibility-breaking change).

- Improved `arb_pow` and `acb_pow` with an inexact base and a negative integer or negative half-integer exponent; the inverse is now computed before performing binary exponentiation in this case to avoid spurious blow-up.
- Elliptic functions
 - Improved Jacobi theta functions to reduce the argument modulo the lattice parameter, greatly improving speed and numerical stability for large input.
 - Optimized `arb_agm` by using a final series expansion and using special code for wide intervals.
 - Optimized `acb_agm1` by using a final series expansion and using special code for positive real input.
 - Optimized derivative of AGM for high precision by using a central difference instead of a forward difference.
 - Optimized `acb_elliptic_rf` and `acb_elliptic_rj` for high precision by using a variable length series expansion.
- Other
 - Fixed incorrect handling of subnormals in `arf_set_d`.
 - Added `mag_bin_uiui` for bounding binomial coefficients.
 - Added `mag_set_d_lower`, `mag_sqrt_lower`, `mag_set_d_2exp_fmpz_lower`.
 - Implemented multithreaded complex matrix multiplication.
 - Optimized `arb_rel_accuracy_bits` by adding fast path.
 - Fixed a spurious floating-point exception (division by zero) in the `t-gauss_period_minpoly` test program triggered by new code optimizations in recent versions of GCC that are unsafe together with FLINT inline assembly functions (a workaround was added to the test code, and a proper fix for the assembly code has been added to FLINT).

2017-07-10 – Arb 2.11.1

- Avoid use of a function that was unavailable in the latest public FLINT release

2017-07-09 – Arb 2.11.0

- Special functions
 - Added the Lambert W function (`arb_lambertw`, `acb_lambertw`, `arb_poly_lambertw_series`, `acb_poly_lambertw_series`). All complex branches and evaluation of derivatives are supported.
 - Added the `acb_expm1` method, complementing `arb_expm1`.
 - Added `arb_sinc_pi`, `acb_sinc_pi`.
 - Optimized handling of more special cases in the Hurwitz zeta function.
- Polynomials
 - Added the `arb_fmpz_poly` module to provide Arb methods for FLINT integer polynomials.
 - Added methods for evaluating an `fmpz_poly` at `arb_t` and `acb_t` arguments.
 - Added `arb_fmpz_poly_complex_roots` for computing the real and complex roots of an integer polynomial, turning the functionality previously available in the `poly_roots.c` example program into a proper library function.
 - Added a method (`arb_fmpz_poly_gauss_period_minpoly`) for constructing minimal polynomials of Gaussian periods.

- Added `arb_poly_product_roots_complex` for constructing a real polynomial from complex conjugate roots.
- Miscellaneous
 - Fixed test code in the `dirichlet` module for 32-bit systems (contributed by Pascal Molin).
 - Use `flint_abort()` instead of `abort()` (contributed by Tommy Hofmann).
 - Fixed the static library install path (contributed by François Bissey).
 - Made `arb_nonnegative_part()` a publicly documented method.
 - Arb now requires FLINT version 2.5 or later.

2017-02-27 – Arb 2.10.0

- General
 - Changed a large number of methods from inline functions to normal functions, substantially reducing the size of the built library.
 - Fixed a few minor memory leaks (missing `clear()` calls).
- Basic arithmetic
 - Added `arb_is_int_2exp_si` and `acb_is_int_2exp_si`.
 - Added `arf_sosq` for computing x^2+y^2 of floating-point numbers.
 - Improved error bounds for complex square roots in the left half plane.
 - Improved error bounds for complex reciprocal (`acb_inv`) and division.
 - Added the internal helper `mag_get_d_log2_approx` as a public method.
- Elliptic functions and integrals
 - New module `acb_elliptic.h` for elliptic functions and integrals.
 - Added complete elliptic integral of the third kind.
 - Added Legendre incomplete elliptic integrals (first, second, third kinds).
 - Added Carlson symmetric incomplete elliptic integrals (RF, RC, RG, RJ, RD).
 - Added Weierstrass elliptic zeta and sigma functions.
 - Added inverse Weierstrass elliptic p-function.
 - Added utility functions for computing the Weierstrass invariants and lattice roots.
 - Improved computation of derivatives of Jacobi theta functions by using modular transformations, and added a main evaluation function (`acb_modular_theta_jet`).
 - Improved detection of pure real or pure imaginary parts in various cases of evaluating theta and modular functions.
- Other special functions
 - New, far more efficient implementation of the dilogarithm function (`acb_polylog` with $s = 2$).
 - Fixed an issue in the Hurwitz zeta function leading to unreasonable slowdown for certain complex input.
 - Added `acb_poly_exp_pi_i_series`.
 - Added `arb_poly_log1p_series`, `acb_poly_log1p_series`.

2016-12-02 – Arb 2.9.0

- License
 - Changed license from GPL to LGPL.
- Build system and compatibility
 - Fixed FLINT includes to use flint/foo.h instead of foo.h, simplifying compilation on many systems.
 - Added another alias for the dynamic library to fix make check on certain systems (contributed by Andreas Enge).
 - Travis CI support (contributed by Isuru Fernando).
 - Added support for ARB_TEST_MULTIPLIER environment variable to control the number of test iterations.
 - Support building with CMake (contributed by Isuru Fernando).
 - Support building with MSVC on Windows (contributed by Isuru Fernando).
 - Fixed unsafe use of FLINT_ABS for slong -> ulong conversion in arf.h, which caused failures on MIPS and ARM systems.
- Basic arithmetic and methods
 - Fixed mag_addmul(x,x,x) with x having a mantissa of all ones. This could produce a non-normalized mag_t value, potentially leading to incorrect results in arb and acb level arithmetic. This bug was caught by new test code, and fortunately would have been hard to trigger accidentally.
 - Added fasth paths for error bound calculations in arb_sqrt and arb_div, speeding up these operations significantly at low precision
 - Added support for round-to-nearest in all arf methods.
 - Added fprintf methods (contributed by Alex Griffing).
 - Added acb_printn and acb_fprintn methods to match arb_printn.
 - Added arb_equal_si and acb_equal_si.
 - Added arb_can_round_mpfr.
 - Added arb_get_ubound_arf, arb_get_lbound_arf (contributed by Tommy Hofmann).
 - Added sign function (arb_sgn).
 - Added complex sign functions (acb_sgn, acb_csgn).
 - Rewrote arb_contains_fmpq to make the test exact.
 - Optimized mag_get_fmpq.
 - Optimized arf_get_fmpz and added more robust test code.
 - Rewrote arb_get_unique_fmpz and arb_get_interval_fmpz_2exp, reducing overhead, making them more robust with huge exponents, and documenting their behavior more carefully.
 - Optimized arb_union.
 - Optimized arf_is_int, arf_is_int_2exp_si and changed these from inline to normal functions.
 - Added mag_const_pi, mag_sub, mag_expinv.
 - Optimized binary-to-decimal conversion for huge exponents by using exponential function instead of binary powering.
 - Added arb_intersection (contributed by Alex Griffing).
 - Added arb_min, arb_max (contributed by Alex Griffing).

- Fixed a bug in `arb_log` and in test code on 64-bit Windows due to unsafe use of MPFR which only uses 32-bit exponents on Win64.
- Improved some test functions to reduce the chance of reporting spurious failures.
- Added squaring functions (`arb_sqr`, `acb_sqr`) (contributed by Ricky Farr).
- Added `arf_frexp`.
- Added `arf_cmp_si`, `arf_cmp_ui`, `arf_cmp_d`.
- Added methods to count allocated bytes (`arb_allocated_bytes`, `_arb_vec_allocated_bytes`, etc.).
- Added methods to predict memory usage for large vectors (`_arb/_acb_vec_estimate_allocated_bytes`).
- Changed `clear()` methods from inline to normal functions, giving 8% faster compilation and 25% smaller `libarb.so`.
- Added `acb_unit_root` and `_acb_vec_unit_roots` (contributed by Pascal Molin).
- Polynomials
 - Added `sinh` and `cosh` functions of power series (`arb/acb_poly_sinh/cosh_series` and `sinh_cosh_series`).
 - Use basecase series inversion algorithm to improve speed and error bounds in `arb/acb_poly_inv_series`.
 - Added functions for fast polynomial Taylor shift (`arb_poly_taylor_shift`, `acb_poly_taylor_shift` and variants).
 - Fast handling of special cases in polynomial composition.
 - Added `acb_poly` scalar `mul` and `div` convenience methods (contributed by Alex Griffing).
 - Added `set_trunc`, `set_trunc_round` convenience methods.
 - Added `add_series`, `sub_series` methods for truncating addition.
 - Added polynomial `is_zero`, `is_one`, `is_x`, `valuation` convenience methods.
 - Added hack to `arb_poly_mullo` and `acb_poly_mullo` to avoid overhead when doing an in-place multiplication with length at most 2.
 - Added binomial and Borel transform methods for `acb_poly`.
- Matrices
 - Added Cholesky decomposition plus solving and inverse for positive definite matrices (`arb_mat_cho`, `arb_mat_spd_solve`, `arb_mat_spd_inv` and related methods) (contributed by Alex Griffing).
 - Added LDL decomposition and inverse and solving based on LDL decomposition for real matrices (`arb_mat_ldl`, `arb_mat_solve_ldl_precomp`, `arb_mat_inv_ldl_precomp`) (contributed by Alex Griffing).
 - Improved the entrywise error bounds in matrix exponential computation to preserve sparsity and give exact entries where possible in many cases (contributed by Alex Griffing).
 - Added public functions for computing the truncated matrix exponential Taylor series (`arb_mat_exp_taylor_sum`, `acb_mat_exp_taylor_sum`).
 - Added functions related to sparsity structure (`arb_mat_entrywise_is_zero`, `arb_mat_count_is_zero`, etc.) (contributed by Alex Griffing).
 - Entrywise multiplication (`arb_mat_mul_entrywise`, `acb_mat_mul_entrywise`) (contributed by Alex Griffing).
 - Added `is_empty` and `is_square` convenience methods (contributed by Alex Griffing).

- Added the `bool_mat` helper module for matrices over the boolean semiring (contributed by Alex Griffing).
- Added Frobenius norm computation (contributed by Alex Griffing).
- Miscellaneous special functions
 - Added evaluation of Bernoulli polynomials (`arb_bernoulli_poly_ui`, `acb_bernoulli_poly_ui`).
 - Added convenience function for evaluation of huge Bernoulli numbers (`arb_bernoulli_fmpz`).
 - Added Euler numbers (`arb_euler_number_ui`, `arb_euler_number_fmpz`).
 - Added fast approximate partition function (`arb_partitions_fmpz/ui`).
 - Optimized partition function for $n < 1000$ by using recurrence for the low 64 bits.
 - Improved the worst-case error bound in `arb_atan`.
 - Added `arb_log_base_ui`.
 - Added complex sinc function (`acb_sinc`).
 - Special handling of $z = 1$ when computing polylogarithms.
 - Fixed `agm(-1,-1)` to output 0 instead of indeterminate.
 - Made working precision in `arb_gamma` and `acb_gamma` more sensitive to the input accuracy.
- Hypergeometric functions
 - Compute `erf` and `erfc` without cancellation problems for large or complex z .
 - Avoid re-computing the square root of π in several places.
 - Added generalized hypergeometric function (`acb_hypgeom_pfq`).
 - Implement binary splitting and rectangular splitting for evaluation of hypergeometric series with a power series parameter, greatly speeding up `Y_n`, `K_n` and other functions at high precision, as well as speeding up high-order parameter derivatives.
 - Use binary splitting more aggressively in `acb_hypgeom_pfq_sum` to reduce error bound inflation.
 - Asymptotic expansions of hypergeometric functions: more accurate parameter selection, and better handling of terminating cases.
 - Tweaked algorithm selection and working precision in `acb_hypgeom_m`.
 - Avoid dividing by the denominator of the next term in `acb_hypgeom_sum`, which would lead to a division by zero when evaluating hypergeometric polynomials.
 - Fixed a bug in hypergeometric series evaluation resulting in near-integers not being skipped in some cases, leading to unnecessary loss of precision.
 - Added series expansions of Airy functions (`acb_hypgeom_airy_series`, `acb_hypgeom_airy_jet`).
 - Fixed a case where Airy functions accidentally chose the worst algorithm instead of the best one.
 - Added functions for computing `erf`, `erfc`, `erfi` of power series in the `acb_hypgeom` module.
 - Added series expansion of the logarithmic integral (`acb_hypgeom_li_series`).
 - Added Fresnel integrals (`acb_hypgeom_fresnel`, `acb_hypgeom_fresnel_series`).
 - Added the lower incomplete gamma function (`acb_hypgeom_gamma_lower`) (contributed by Alex Griffing).
 - Added series expansion of the lower incomplete gamma function (`acb_hypgeom_gamma_lower_series`) (contributed by Alex Griffing).

- Added support for computing the regularized incomplete gamma functions.
- Use slightly sharper error bound for analytic continuation of $2F1$.
- Added support for computing finite limits of $2F1$ with inexact parameters differing by integers.
- Added the incomplete beta function (`acb_hypgeom_beta_lower`, `acb_hypgeom_beta_lower_series`)
- Improved `acb_hypgeom_u` to use a division-avoiding algorithm for small polynomial cases.
- Added `arb_hypgeom` module, wrapping the complex hypergeometric functions for more convenient use with the `arb_t` type.
- Dirichlet L-functions and Riemann zeta function
 - New module `dirichlet` for working algebraically with Dirichlet groups and characters (contributed by Pascal Molin).
 - New module `acb_dirichlet` for numerical evaluation of Dirichlet characters and L-functions (contributed by Pascal Molin).
 - Efficient representation and manipulation of Dirichlet characters using the Conrey representation (contributed by Pascal Molin).
 - New module `dlog` for word-size discrete logarithm evaluation, used to support algorithms on Dirichlet characters (contributed by Pascal Molin).
 - Methods for properties, evaluation, iteration, pairing, lift, lowering etc. of Dirichlet characters (contributed by Pascal Molin).
 - Added `acb_dirichlet_roots` methods for fast evaluation of many roots of unity (contributed by Pascal Molin).
 - Added `acb_dirichlet_hurwitz_precomp` methods for fast multi-evaluation of the Hurwitz zeta function for many parameter values.
 - Added methods for computing Gauss, Jacobi and theta sums over Dirichlet characters (contributed by Pascal Molin).
 - Added methods (`acb_dirichlet_l`, `acb_dirichlet_l_jet`, `acb_dirichlet_l_series`) for evaluation of Dirichlet L-functions and their derivatives.
 - Implemented multiple algorithms for evaluation of Dirichlet L-functions depending on the argument (Hurwitz zeta function decomposition, Euler product, functional equation).
 - Added methods (`acb_dirichlet_hardy_z`, `acb_dirichlet_hardy_z_series`, etc.) for computing the Hardy Z-function corresponding to a Dirichlet L-function.
 - Added fast bound for Hurwitz zeta function (`mag_hurwitz_zeta_uiui`).
 - Improved parameter selection in Hurwitz zeta function to target relative instead of absolute error for large positive s .
 - Improved parameter selection in Hurwitz zeta function to avoid computing unnecessary Bernoulli numbers for large imaginary s .
 - Added Dirichlet eta function (`acb_dirichlet_eta`).
 - Implemented the Riemann-Siegel formula for faster evaluation of the Riemann zeta function at large height.
 - Added smooth-index algorithm for the main sum when evaluating the Riemann zeta function, avoiding the high memory usage of the full sieving algorithm when the number of terms gets huge.
 - Improved tuning for using the Euler product when computing the Riemann zeta function.
- Example programs
 - Added logistic map example program.

- Added lvalue example program.
- Improved `poly_roots` in several ways: identify roots that are exactly real, automatically perform squarefree factorization, use power hack, and allow specifying a product of polynomials as input on the command line.
- Housekeeping
 - New section in the documentation giving an introduction to ball arithmetic and using the library.
 - Tidied, documented and added test code for the `fmpz_extras` module.
 - Added proper documentation and test code for many helper methods.
 - Removed the obsolete `fmpqb` module entirely.
 - Documented more algorithms and formulas.
 - Clarified integer overflow issues and use of `ARF_PREC_EXACT` in the documentation.
 - Added `.gitignore` file.
 - Miscellaneous improvements to the documentation.

2015-12-31 – Arb 2.8.1

- Fixed 32-bit test failure for the Laguerre function.
- Made the Laguerre function indeterminate at negative integer orders, to be consistent with the test code.

2015-12-29 – Arb 2.8.0

- Compatibility and build system
 - Windows64 support (contributed by Bill Hart).
 - Fixed a bug that broke basic arithmetic on targets where FLINT uses fallback code instead of assembly code, such as PPC64 (contributed by Jeroen Demeyer).
 - Fixed configure to use `EXTRA_SHARED_FLAGS/LDFLAGS`, and other build system fixes (contributed by Tommy Hofmann, Bill Hart).
 - Added soname versioning (contributed by Julien Puydt).
 - Fixed test code on MinGW (contributed by Hrvoje Abraham).
 - Miscellaneous fixes to simplify interfacing Arb from Julia.
- Arithmetic and elementary functions
 - Fixed `arf_get_d` to handle underflow/overflow correctly and to support round-to-nearest.
 - Added more complex inverse hyperbolic functions (`acb_asin`, `acb_acos`, `acb_asinh`, `acb_acosh`, `acb_atanh`).
 - Added `arb_contains_int` and `acb_contains_int` for testing whether an interval contains any integer.
 - Added `acb_quadratic_roots_fmpz`.
 - Improved `arb_sinh` to use a more accurate formula for $x < 0$.
 - Added sinc function (`arb_sinc`) (contributed by Alex Griffing).
 - Fixed bug in `arb_exp` affecting convergence for huge input.
 - Faster implementation of `arb_div_2expm1_ui`.

- Added `mag_root`, `mag_geom_series`.
- Improved and added test code for `arb_add_error` functions.
- Changed `arb_pow` and `acb_pow` to make `pow(0, positive) = 0` instead of `nan`.
- Improved `acb_sqrt` to return finite output for finite input straddling the branch cut.
- Improved `arb_set_interval_arf` so that `[inf, inf] = inf` instead of an infinite interval.
- Added computation of Bell numbers (`arb_bell_fmpz`).
- Added `arb_power_sum_vec` for computing power sums using Bernoulli numbers.
- Added computation of the Fujiwara root bound for `acb_poly`.
- Added code to identify all the real roots of a real polynomial (`acb_poly_validate_real_roots`).
- Added several convenient assignment functions, including `arb_set_d`, `acb_set_d`, `acb_set_d_d`, `acb_set_fmpz_fmpz` (contributed by Ricky Farr).
- Added many accessor functions (`_arb/acb_vec_entry_ptr`, `arb_get_mid/rad_arb`, `acb_real/imag_ptr`, `arb_mid/rad_ptr`, `acb_get_real/imag`).
- Added missing functions `acb_add_si`, `acb_sub_si`.
- Renamed `arb_root` to `arb_root_ui` (keeping alias) and added `acb_root_ui`.
- Special functions
 - Implemented the Gauss hypergeometric function ${}_2F_1$ and its regularized version.
 - Fixed two bugs in `acb_hypgeom_pfq_series_direct` discovered while implementing ${}_2F_1$. In rare cases, these could lead to incorrect values for functions depending on parameter derivatives of hypergeometric series.
 - * The first bug involved incorrect handling of negative integer parameters. The bug only affected ${}_2F_1$ and higher functions; it did not affect correctness of any previously implemented functions that relied on `acb_hypgeom_pfq_series_direct` (such as Bessel Y and K functions of integer order).
 - * The second bug involved a too small bound being computed for the sum of a geometric series. The geometric series bound is nearly tight for ${}_2F_1$, and the incorrect version caused immediate test failures for that function. Theoretically, this bug affected correctness of some previously-implemented functions that relied on `acb_hypgeom_pfq_series_direct` (such as Bessel Y and K functions of integer order), but since the geometric bound is not as tight in those cases, those functions were still reliable in practice (no failing test case has been found).
 - Implemented Airy functions and their derivatives (`acb_hypgeom_airy`).
 - Implemented the confluent hypergeometric function ${}_0F_1$ (`acb_hypgeom_0f1`).
 - Implemented associated Legendre functions P and Q.
 - Implemented Chebyshev, Jacobi, Gegenbauer, Laguerre, Hermite functions.
 - Implemented spherical harmonics.
 - Added function for computing Bessel J and Y functions simultaneously.
 - Added rising factorials for non-integer n (`arb_rising`, `acb_rising`).
 - Made rising factorials use gamma function for large integer n .
 - Faster algorithm for theta constants and Dedekind eta function at very high precision.
 - Fixed `erf` to give finite values instead of $\pm\infty$ for big imaginary input.
 - Improved `acb_zeta` (and `arb_zeta`) to automatically use fast code for integer zeta values.
 - Added double factorial (`arb_doublefac_ui`).

- Added code for generating Hilbert class polynomials (`acb_modular_hilbert_class_poly`).
- Matrices
 - Added faster matrix squaring (`arb/acb_mat_sqr`) (contributed by Alex Griffing).
 - Added matrix trace (`arb/acb_mat_trace`) (contributed by Alex Griffing).
 - Added `arb/acb_mat_set_round_fmpz_mat`, `acb_mat_set(_round)_arb_mat` (contributed by Tommy Hofmann).
 - Added `arb/acb_mat_transpose` (contributed by Tommy Hofmann).
 - Added comparison methods `arb/acb_mat_eq/ne` (contributed by Tommy Hofmann).
- Other
 - Added `complex_plot` example program.
 - Added Airy functions to `real_roots` example program.
 - Other minor patches were contributed by Alexander Kobel, Marc Mezzarobba, Julien Puydt.
 - Removed obsolete file `config.h`.

2015-07-14 – Arb 2.7.0

- Hypergeometric functions
 - Implemented Bessel I and Y functions (`acb_hypgeom_bessel_i`, `acb_hypgeom_bessel_y`).
 - Fixed bug in Bessel K function giving the wrong branch for negative real arguments.
 - Added code for evaluating complex hypergeometric series binary splitting.
 - Added code for evaluating complex hypergeometric series using fast multipoint evaluation.
- Gamma related functions
 - Implemented the Barnes G-function and its continuous logarithm (`acb_barnes_g`, `acb_log_barnes_g`).
 - Implemented the generalized polygamma function (`acb_polygamma`).
 - Implemented the reflection formula for the logarithmic gamma function (`acb_lgamma`, `acb_poly_lgamma_series`).
 - Implemented the digamma function of power series (`arb_poly_digamma_series`, `acb_poly_digamma_series`).
 - Improved `acb_poly_zeta_series` to produce exact zero imaginary parts in most cases when the result should be real-valued.
 - Made the real logarithmic gamma function (`arb_lgamma`, `arb_poly_lgamma_series`) abort more quickly for negative input.
- Elementary functions
 - Added `arb_exp_expinv` and `acb_exp_expinv` functions for simultaneously computing $\exp(x)$, $\exp(-x)$.
 - Improved `acb_tan`, `acb_tan_pi`, `acb_cot` and `acb_cot_pi` for input with large imaginary parts.
 - Added complex hyperbolic functions (`acb_sinh`, `acb_cosh`, `acb_sinh_cosh`, `acb_tanh`, `acb_coth`).
 - Added `acb_log_sin_pi` for computing the logarithmic sine function without branch cuts away from the real line.
 - Added `arb_poly_cot_pi_series`, `acb_poly_cot_pi_series`.

- Added `arf_root` and improved speed of `arb_root`.
- Tuned algorithm selection in `arb_pow_fmpq`.
- Other
 - Added documentation for `arb` and `acb` vector functions.

2015-04-19 – Arb 2.6.0

- Special functions
 - Added the Bessel K function.
 - Added the confluent hypergeometric functions M and U.
 - Added exponential, trigonometric and logarithmic integrals `ei`, `si`, `shi`, `ci`, `chi`, `li`.
 - Added the complete elliptic integral of the second kind E.
 - Added support for computing hypergeometric functions with power series as parameters.
 - Fixed special cases in Bessel J function returning useless output.
 - Fixed precision of zeta function accidentally being capped at 7000 digits (bug in 2.5).
 - Special-cased real input in the gamma functions for complex types.
 - Fixed exp of huge numbers outputting unnecessarily useless intervals.
 - Fixed broken code in `erf` that sometimes gave useless output.
 - Made selection of number of terms in hypergeometric series more robust.
- Polynomials and power series.
 - Added `sin_pi`, `cos_pi` and `sin_cos_pi` for real and complex power series.
 - Speeded up series reciprocal and division for `length = 2`.
 - Added `add_si` methods for polynomials.
 - Made `inv_series` and `div_series` with zero input produce indeterminates instead of aborting.
 - Added `arb_poly_majorant`, `acb_poly_majorant`.
- Basic functions
 - Added comparison methods `arb_eq`, `arb_ne`, `arb_lt`, `arb_le`, `arb_gt`, `arb_ge`, `acb_eq`, `acb_ne`.
 - Added `acb_rel_accuracy_bits` and improved the real version.
 - Fixed precision of constants like `pi` behaving more nondeterministically than necessary.
 - Fixed `arf_get_mag_lower(nan)` to output 0 instead of `inf`.
- Other
 - Removed call to `fmpq_dedekind_sum` which only exists in the git version of flint.
 - Fixed a test code bug that could cause crashes on some systems.
 - Added fix for static build on OS X (thanks Marcello Seri).
 - Miscellaneous corrections to the documentation.

2015-01-28 – Arb 2.5.0

- String conversion
 - Added `arb_set_str`.
 - Added `arb_get_str` and `arb_printn` for pretty-printed rigorous decimal output.
 - Added helper functions for binary to decimal conversion.
- Core arithmetic
 - Improved speed of division when using GMP instead of MPIR.
 - Improved complex division with a small denominator.
 - Removed a little bit of overhead for complex squaring.
- Special functions
 - Faster code for `atan` at very high precision, used instead of `mpfr_atan`.
 - Optimized elementary functions slightly for small input.
 - Added modified error functions `erfc` and `erfi`.
 - Added the generalized exponential integral.
 - Added the upper incomplete gamma function.
 - Implemented the complete elliptic integral of the first kind.
 - Implemented the arithmetic-geometric mean of complex numbers.
 - Optimized `arb_digamma` for small integers.
 - Made `mag_log_ui`, `mag_binpow_uiui` and `mag_polylog_tail` proper functions.
 - Added `pow`, `agm`, `erf`, `elliptic_k`, `elliptic_p` as functions of complex power series.
 - Added incomplete gamma function of complex power series.
 - Improved code for bounding complex rising factorials (the old code could potentially have given wrong results in degenerate cases).
 - Added `arb_sqrt1pm1`, `arb_atanh`, `arb_asinh`, `arb_atanh`.
 - Added `arb_log1p`, `acb_log1p`, `acb_atan`.
 - Added `arb_hurwitz_zeta`.
 - Improved parameter selection in the Hurwitz zeta function to try to avoid stalling when given enormous input.
 - Optimized `sqrt` and `rsqrt` of power series when given a binomial as input.
 - Made `arb_bernoulli_ui(264-2)` not crash.
 - Fixed `rgamma` of negative integers returning indeterminate.
- Polynomials and matrices
 - Added characteristic polynomial computation for real and complex matrices.
 - Added polynomial `set_round` methods.
 - Added `is_real` methods for more types.
 - Added more `get_unique_fmpz` methods.
 - Added code for generating Swinnerton-Dyer polynomials.
 - Improved error bounding in `det()` and `exp()` of complex matrices to recognize when the result is real-valued.
 - Changed polynomial `divrem` to return success/fail instead of aborting on divide by zero.

- Miscellaneous
 - Added logo to documentation.
 - Made inlined functions build as part of the library.
 - Silenced a clang warning.
 - Made `_acb_vec_sort_pretty` a library function.

2014-11-15 – Arb 2.4.0

- Arithmetic and core functions
 - Made evaluation of `sin`, `cos` and `exp` at medium precision faster using the `sqrt` trick.
 - Optimized `arb_sinh` and `arb_sinh_cosh`.
 - Optimized complex division with a small denominator.
 - Optimized cubing of complex numbers.
 - Added `floor` and `ceil` functions for the `arf` and `arb` types.
 - Added `acb_poly` powering functions.
 - Added `acb_exp_pi_i`.
 - Added functions for evaluation of Chebyshev polynomials.
 - Fixed `arb_div` to output `nan` for input containing `nan`.
- Added a module `acb_hypgeom` for hypergeometric functions
 - Evaluation of the generalized hypergeometric function in convergent cases.
 - Evaluation of confluent hypergeometric functions using asymptotic expansions.
 - The Bessel function of the first kind for complex input.
 - The error function for complex input.
- Added a module `acb_modular` for modular forms and elliptic functions
 - Support for working with modular transformations.
 - Mapping a point to the fundamental domain.
 - Evaluation of Jacobi theta functions and their series expansions.
 - The Dedekind eta function.
 - The `j`-invariant and the modular `lambda` and `delta` function.
 - Eisenstein series.
 - The Weierstrass elliptic function and its series expansion.
- Miscellaneous
 - Fixed `mag_print` printing a too large exponent.
 - Fixed `printd` methods to use a fallback instead of aborting when printing numbers too large for MPFR.
 - Added version number string (`arb_version`).
 - Various additions to the documentation.

2014-09-25 – Arb 2.3.0

- Removed most of the legacy (Arb 1.x) modules.
- Updated build scripts, hopefully fixing various issues.
- New implementations of `arb_sin`, `arb_cos`, `arb_sin_cos`, `arb_atan`, `arb_log`, `arb_exp`, `arb_expm1`, much faster up to a few thousand bits.
- Ported the bit-burst code for high-precision exponentials to the arb type.
- Speeded up `arb_log_ui_from_prev`.
- Added `mag_exp`, `mag_expm1`, `mag_exp_tail`, `mag_pow_fmpz`.
- Improved various mag functions.
- Added `arb_get/set_interval_mpf`, `arb_get_interval_arf`, and improved `arb_set_interval_arf`.
- Improved `arf_get_fmpz`.
- Prettier printing of complex numbers with negative imaginary part.
- Changed some frequently-used functions from inline to non-inline to reduce code size.

2014-08-01 – Arb 2.2.0

- Added functions for computing polylogarithms and order expansions of polylogarithms, with support for real and complex s , z .
- Added a missing cast affecting C++ compatibility.
- Generalized `powsum` functions to allow a geometric factor.
- Improved `powsum` functions slightly when the exponent is an integer.
- Faster `arb_log_ui_from_prev`.
- Added `mag_sqrt` and `mag_rsqr` functions.
- Fixed various minor bugs and added missing tests and documentation entries.

2014-06-20 – Arb 2.1.0

- Ported most of the remaining functions to the new arb/acb types, including:
 - Elementary functions (`log`, `atan`, etc.).
 - Hypergeometric series summation.
 - The gamma function.
 - The Riemann zeta function and related functions.
 - Bernoulli numbers.
 - The partition function.
 - The calculus modules (rigorous real root isolation, rigorous numerical integration of complex-valued functions).
 - Example programs.
- Added several missing utility functions to the arf and mag modules.

2014-05-27 – Arb 2.0.0

- New modules `mag`, `arf`, `arb`, `arb_poly`, `arb_mat`, `acb`, `acb_poly`, `acb_mat` for higher-performance ball arithmetic.
- `Poly_roots2` and `hilbert_matrix2` example programs.
- Vector dot product and norm functions (contributed by Abhinav Baid).

2014-05-03 – Arb 1.1.0

- Faster and more accurate error bounds for polynomial multiplication (error bounds are now always as good as with classical multiplication, and multiplying high-degree polynomials with approximately equal coefficients now has proper quasilinear complexity).
- Faster and much less memory-hungry exponentials at very high precision.
- Improved the partition function to support `n` bigger than a single word, and enabled the possibility to use two threads for the computation.
- Fixed a bug in floating-point arithmetic that caused a too small bound for the rounding error to be reported when the result of an inexact operation was rounded up to a power of two (this bug did not affect the correctness of ball arithmetic, because operations on ball midpoints always round down).
- Minor optimizations to floating-point arithmetic.
- Improved argument reduction of the digamma function and short series expansions of the rising factorial.
- Removed the holonomic module for now, as it did not really do anything very useful.

2013-12-21 – Arb 1.0.0

- New example programs directory
 - `poly_roots` example program.
 - `real_roots` example program.
 - `pi_digits` example program.
 - `hilbert_matrix` example program.
 - `keiper_li` example program.
- New `fmprb_calc` module for calculus with real functions
 - Bisection-based root isolation.
 - Asymptotically fast Newton root refinement.
- New `fmpcb_calc` module for calculus with complex functions
 - Numerical integration using Taylor series.
- Scalar functions
 - Simplified `fmprb_const_euler` using published error bound.
 - Added `fmprb_inv`.
 - Added `fmprb_trim`, `fmpcb_trim`.
 - Added `fmpcb_rsqrt` (complex reciprocal square root).
 - Fixed bug in `fmprb_sqrtpos` with nonfinite input.
 - Slightly improved `fmprb powering` code.

- Added various functions for bounding fmprs by powers of two.
 - Added `fmpr_is_int`.
- Polynomials and power series
 - Implemented scaling to speed up blockwise multiplication.
 - Slightly faster basecase power series exponentials.
 - Improved `sin/cos/tan/exp` for short power series.
 - Added complex `sqrt_series`, `rsqrt_series`.
 - Implemented the Riemann-Siegel Z and theta functions for real power series.
 - Added `fmprb_poly_pow_series`, `fmprb_poly_pow_ui` and related methods.
 - Added `fmprb/fmpcb_poly_contains_fmpz_poly`.
 - Faster composition by monomials.
 - Implemented Borel transform and binomial transform for real power series.
- Matrices
 - Implemented matrix exponentials.
 - Multithreaded `fmprb_mat_mul`.
 - Added matrix infinity norm functions.
 - Added some more matrix-scalar functions.
 - Added matrix contains and overlaps methods.
- Zeta function evaluation
 - Multithreaded power sum evaluation.
 - Faster parameter selection when computing many derivatives.
 - Implemented binary splitting to speed up computing many derivatives.
- Miscellaneous
 - Corrections for C++ compatibility (contributed by Jonathan Bober).
 - Several minor bugfixes and test code enhancements.

2013-08-07 – Arb 0.7

- Floating-point and ball functions
 - Documented, added test code, and fixed bugs for various operations involving a ball containing an infinity or NaN.
 - Added reciprocal square root functions (`fmpr_rsqrt`, `fmprb_rsqrt`) based on `mpfr_rec_sqrt`.
 - Faster high-precision division by not computing an explicit remainder.
 - Slightly faster computation of π by using new reciprocal square root and division code.
 - Added an `fmpr` function for approximate division to speed up certain radius operations.
 - Added `fmpr_set_d` for conversion from double.
 - Allow use of doubles to optionally compute the partition function faster but without an error bound.
 - Bypass `mpfr` overflow when computing the exponential function to extremely high precision (approximately 1 billion digits).

- Made `fmpcb_exp` faster for large numbers at extremely high precision by skipping the $\log(2)$ removal.
- Made `fmpcb_lgamma` faster at high precision by speeding up the argument reduction branch computation.
- Added `fmpcb_asin`, `fmpcb_acos`.
- Added various other utility functions to the `fmpcb` module.
- Added a function for computing the Glaisher constant.
- Optimized evaluation of the Riemann zeta function at high precision.
- Polynomials and power series
 - Made squaring of polynomials faster than generic multiplication.
 - Implemented power series reversion (various algorithms) for the `fmpcb_poly` type.
 - Added many `fmpcb_poly` utility functions (shifting, truncating, setting/getting coefficients, etc.).
 - Improved power series division when either operand is short
 - Improved power series logarithm when the input is short.
 - Improved power series exponential to use the basecase algorithm for short input regardless of the output size.
 - Added power series square root and reciprocal square root.
 - Added `atan`, `tan`, `sin`, `cos`, `sin_cos`, `asin`, `acos` `fmpcb_poly` power series functions.
 - Added Newton iteration macros to simplify various functions.
 - Added gamma functions of real and complex power series (`[fmpcb/fmpcb]_poly_[gamma/rgamma/lgamma]_series`).
 - Added wrappers for computing the Hurwitz zeta function of a power series (`[fmpcb/fmpcb]_poly_zeta_series`).
 - Implemented sieving and other optimizations to improve performance for evaluating the zeta function of a short power series.
 - Improved power series composition when the inner series is linear.
 - Added many `fmpcb_poly` versions of nearly all `fmpcb_poly` functions.
 - Improved speed and stability of series composition/reversion by balancing the power table exponents.
- Other
 - Added support for freeing all cached data by calling `flint_cleanup()`.
 - Introduced `fmpcb_ptr`, `fmpcb_srcptr`, `fmpcb_ptr`, `fmpcb_srcptr` typedefs for cleaner function signatures.
 - Various bug fixes and general cleanup.

2013-05-31 – Arb 0.6

- Made fast polynomial multiplication over the reals numerically stable by using a blockwise algorithm.
- Disabled default use of the Gauss formula for multiplication of complex polynomials, to improve numerical stability.
- Added division and remainder for complex polynomials.
- Added fast multipoint evaluation and interpolation for complex polynomials.
- Added missing `fmprb_poly_sub` and `fmpcb_poly_sub` functions.
- Faster exponentials (`fmprb_exp` and dependent functions) at low precision, using precomputation.
- Rewrote `fmpr_add` and `fmpr_sub` using `mpn` level code, improving efficiency at low precision.
- Ported the partition function implementation from `flint` (using ball arithmetic in all steps of the calculation to guarantee correctness).
- Ported algorithm for computing the cosine minimal polynomial from `flint` (using ball arithmetic to guarantee correctness).
- Support using GMP instead of MPIR.
- Only use thread-local storage when enabled in `flint`.
- Slightly faster error bounding for the zeta function.
- Added some other helper functions.

2013-03-28 – Arb 0.5

- Arithmetic and elementary functions
 - Added `fmpr_get_fmpz`, `fmpr_get_si`.
 - Fixed accuracy problem with `fmprb_div_2expm1`.
 - Special-cased squaring of complex numbers.
 - Added various `fmpcb` convenience functions (`addmul_ui`, etc).
 - Optimized `fmpr_cmp_2exp_si` and `fmpr_cmpabs_2exp_si`, and added test code for comparison functions.
 - Added `fmprb_atan2`, also fixing a bug in `fmpcb_arg`.
 - Added `fmprb_sin_pi`, `cos_pi`, `sin_cos_pi`, etc.
 - Added `fmprb_sin_pi_fmpz` (etc.) using algebraic methods for fast evaluation of roots of unity.
 - Faster `fmprb_poly_evaluate` and `evaluate_fmpcb` using rectangular splitting.
 - Added `fmprb_poly_evaluate2`, `evaluate2_fmpcb` for simultaneously evaluating the derivative.
 - Added `fmprb_poly` root polishing code using near-optimal Newton steps (experimental).
 - Added `fmpr_root`, `fmprb_root` (currently based on MPFR).
 - Added `fmpr_min`, `fmpr_max`.
 - Added `fmprb_set_interval_fmpr`, `fmprb_union`.
 - Added `fmpr_bits`, `fmprb_bits`, `fmpcb_bits` for obtaining the mantissa width.
 - Added `fmprb_hypot`.
 - Added complex square roots.

- Improved `fmprb_log` to slightly improve speed, and properly support huge arguments.
- Fixed `exp`, `cosh`, `sinh` to work with huge arguments.
- Added `fmprb_expm1`.
- Fixed `sin`, `cos`, `atan` to work with huge arguments.
- Improved `fmprb_pow` and `fmpcb_pow`, including automatic detection of small integer and half-integer exponents.
- Added many more elementary functions: `fmprb_tan/cot/tanh/coth`, `fmpcb_tan/cot`, and `pi` versions.
- Added `fmprb const_e`, `const_log2`, `const_log10`, `const_catalan`.
- Fixed ball containment/overlap checking to work operate efficiently and correctly with huge exponents.
- Strengthened test code for many core operations.
- Special functions
 - Reorganized zeta function related code.
 - Faster evaluation of the Riemann zeta function via sieving.
 - Documented and improved efficiency of the zeta constant binary splitting code.
 - Calculate error bound in Borwein’s algorithm with `fmprs` instead of using doubles.
 - Optimized divisions in zeta evaluation via the Euler product.
 - Use functional equation for Riemann zeta function of a negative argument.
 - Compute single Bernoulli numbers using ball arithmetic instead of relying on the floating-point code in `flint`.
 - Initial code for evaluating the gamma function using its Taylor series.
 - Much faster rising factorials at high precision, using difference polynomials.
 - Much faster gamma function at high precision.
 - Added complex gamma function, log gamma function, and other versions.
 - Added `fmprb_agm` (real arithmetic-geometric mean).
 - Added `fmprb_gamma_fmpq`, supporting rapid computation of $\gamma(p/q)$ for $q = 1, 2, 3, 4, 6$.
 - Added real and complex digamma function.
 - Fixed unnecessary recomputation of Bernoulli numbers.
 - Optimized computation of Euler’s constant, and added proper error bounds.
 - Avoid reliance on doubles in the hypergeometric series tail bound.
 - Cleaned up factorials and binomials, computing factorials via gamma.
- Other
 - Added an `fmpz_extras` module to collect various internal `fmpz` helper functions.
 - Fixed detection of `flint` header files.
 - Fixed various other small bugs.

2013-01-26 – Arb 0.4

- Much faster `fmpr_mul`, `fmprb_mul` and `set_round`, resulting in general speed improvements.
- Code for computing the complex Hurwitz zeta function with derivatives.
- Fixed and documented error bounds for hypergeometric series.
- Better algorithm for series evaluation of the gamma function at a rational point.
- Much faster generation of Bernoulli numbers.
- Complex log, exp, pow, trigonometric functions (currently based on MPFR).
- Complex nth roots via Newton iteration.
- Added code for arithmetic on `fmpcb_polys`.
- Code for computing Khinchin's constant.
- Code for rising factorials of polynomials or power series
- Faster `sin_cos`.
- Better `div_2expm1`.
- Many other new helper functions.
- Improved thread safety.
- More test code for core operations.

2012-11-07 – Arb 0.3

- Converted documentation to Sphinx.
- New module `fmpcb` for ball interval arithmetic over the complex numbers
 - Conversions, utility functions and arithmetic operations.
- New module `fmpcb_mat` for matrices over the complex numbers
 - Conversions, utility functions and arithmetic operations.
 - Multiplication, LU decomposition, solving, inverse and determinant.
- New module `fmpcb_poly` for polynomials over the complex numbers
 - Root isolation for complex polynomials.
- New module `fmmpz_holonomic` for functions/sequences defined by linear differential/difference equations with polynomial coefficients
 - Functions for creating various special sequences and functions.
 - Some closure properties for sequences.
 - Taylor series expansion for differential equations.
 - Computing the nth entry of a sequence using binary splitting.
 - Computing the nth entry mod p using fast multipoint evaluation.
- Generic binary splitting code with automatic error bounding is now used for evaluating hypergeometric series.
- Matrix powering.
- Various other helper functions.

2012-09-29 – Arb 0.2

- Code for computing the gamma function (Karatsuba, Stirling's series).
- Rising factorials.
- Fast `exp_series` using Newton iteration.
- Improved multiplication of small polynomials by using classical multiplication.
- Implemented error propagation for square roots.
- Polynomial division (Newton-based).
- Polynomial evaluation (Horner) and composition (divide-and-conquer).
- Product trees, fast multipoint evaluation and interpolation (various algorithms).
- Power series composition (Horner, Brent-Kung).
- Added the `fmprb_mat` module for matrices of balls of real numbers.
- Matrix multiplication.
- Interval-aware LU decomposition, solving, inverse and determinant.
- Many helper functions and small bugfixes.

2012-09-14 – Arb 0.1

- 2012-08-05 - Began simplified rewrite.
- 2012-04-05 - Experimental ball and polynomial code (first commit).

BIBLIOGRAPHY

- [WQ3a] <http://functions.wolfram.com/07.11.26.0033.01>
- [WQ3b] <http://functions.wolfram.com/07.12.27.0014.01>
- [WQ3c] <http://functions.wolfram.com/07.12.26.0003.01>
- [WQ3d] <http://functions.wolfram.com/07.12.26.0088.01>
- [AbbottBronsteinMulders1999] Fast deterministic computation of determinants of dense matrices, ACM International Symposium on Symbolic and Algebraic Computation (1999)
- [Apostol1997] Apostol, Tom : Modular functions and Dirichlet series in number theory, Springer (1997)
- [Ari2011] J. Arias de Reyna, “High precision computation of Riemann’s zeta function by the Riemann-Siegel formula, I”, Mathematics of Computation 80 (2011), 995-1009
- [Ari2012] J. Arias de Reyna, “Programs for Riemann’s zeta function”, (J. A. J. van Vonderen, Ed.) *Leven met getallen : liber amicorum ter gelegenheid van de pensionering van Herman te Riele* CWI (2012) 102-112, <https://ir.cwi.nl/pub/19724>
- [Arn2010] J. Arndt, *Matters Computational*, Springer (2010), <https://www.jjj.de/fxt/#fxtbook>
- [Arn2012] J. Arndt, “On computing the generalized Lambert series”, <https://arxiv.org/abs/1202.6525>
- [ArnoldMonagan2011] Arnold, Andrew and Monagan, Michael : Calculating cyclotomic polynomials, Mathematics of Computation 80:276 (2011) 2359–2379
- [BBC1997] D. H. Bailey, J. M. Borwein and R. E. Crandall, “On the Khintchine constant”, Mathematics of Computation 66 (1997) 417-431
- [BBC2000] J. Borwein, D. M. Bradley and R. E. Crandall, “Computational strategies for the Riemann zeta function”, Journal of Computational and Applied Mathematics 121 (2000) 247-296
- [BBK2014] D. H. Bailey, J. M. Borwein and A. D. Kaiser. “Automated simplification of large symbolic expressions”. Journal of Symbolic Computation Volume 60, January 2014, Pages 120-136. <https://doi.org/10.1016/j.jsc.2013.09.001>
- [BD1992] D. Buchmann and S. Düllmann. “Distributed class group computation.” Informatik: Festschrift zum 60. Geburtstag von Günter Hotz (1992): 69-79.
- [BF2020] F. Beukers and J. Forsgård. “Gamma-evaluations of hypergeometric series”. Preprint, 2020. <https://arxiv.org/abs/2004.08117>
- [BCOSSS2007] A. Bostan, F. Chyzak, F. Ollivier, B. Salvy, É. Schost, A. Sedoglavic. “Fast computation of power series solutions of systems of differential equations”. Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2007) 1012-1021
- [BFSS2006] A. Bostan, P. Flajolet, B. Salvy and É. Schost. “Fast computation of special resultants”. Journal of Symbolic Computation, 41(1):1–29, January 2006. <https://doi.org/10.1016/j.jsc.2005.07.001>

- [BJ2013] R. P. Brent and F. Johansson, “A bound for the error term in the Brent-McMillan algorithm”, preprint (2013), <https://arxiv.org/abs/1312.0039>
- [BM1980] R. P. Brent and E. M. McMillan, “Some new algorithms for high-precision computation of Euler’s constant”, *Mathematics of Computation* 34 (1980) 305-312.
- [BZ1992] J. Borwein and I. Zucker, “Fast evaluation of the gamma function for small rational fractions using complete elliptic integrals of the first kind”, *IMA Journal of Numerical Analysis* 12 (1992) 519-526
- [BZ2011] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press (2011), <http://www.loria.fr/~zimmerma/mca/pub226.html>
- [BaiWag1980] Robert Baillie; Samuel S. Wagstaff, Jr. (October 1980). “Lucas Pseudoprimes”. *Mathematics of Computation*. 35 (152): 1391–1417.
- [BerTas2010] D. Berend and T. Tassa : Improved bounds on Bell numbers and on moments of sums of random variables, *Probability and Mathematical Statistics* vol. 30 (2010) 185–205
- [Bin1996] D. A. Bini, “Numerical computation of polynomial zeros by means of Aberth’s method”. *Numerical algorithms* 13 (1996): 179-200. <https://doi.org/10.1007/BF02207694>
- [Blo2009] R. Bloemen, “Even faster zeta(2n) calculation!”, <https://web.archive.org/web/20141101133659/http://xn-2-umb.com/09/11/even-faster-zeta-calculation>
- [Bodrato2010] Bodrato, Marco : A Strassen-like Matrix Multiplication Suited for Squaring and Higher Power Computation. *Proceedings of the ISSAC 2010 München, Germany*, 25-28 July, 2010
- [Boe2020] H. Boehm. “Towards an API for the real numbers”. *PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2020, Pages 562-576. <https://doi.org/10.1145/3385412.3386037>
- [Bog2012] I. Bogaert, B. Michiels and J. Fostier, “O(1) computation of Legendre polynomials and Gauss-Legendre nodes and weights for parallel computing”, *SIAM Journal on Scientific Computing* 34:3 (2012), C83-C101
- [Bol1887] O. Bolza, “Darstellung der rationalen ganzen Invarianten der Binärform sechsten Grades durch die Nullwerthe der zugehörigen Theta-Functionen”, *Math. Ann.* 30:4 (1887), 478–495. <https://doi.org/10.1007/BF01444091>
- [Bor1987] P. Borwein, “Reduced complexity evaluation of hypergeometric functions”, *Journal of Approximation Theory* 50:3 (1987)
- [Bor2000] P. Borwein, “An Efficient Algorithm for the Riemann Zeta Function”, *Constructive experimental and nonlinear analysis*, CMS Conference Proc. 27 (2000) 29-34, <http://www.cecm.sfu.ca/personal/pborwein/PAPERS/P155.pdf>
- [Bre1978] R. P. Brent, “A Fortran multiple-precision arithmetic package”, *ACM Transactions on Mathematical Software*, 4(1):57–70, March 1978.
- [Bre1979] R. P. Brent, “On the Zeros of the Riemann Zeta Function in the Critical Strip”, *Mathematics of Computation* 33 (1979), 1361-1372, <https://doi.org/10.1090/S0025-5718-1979-0537983-2>
- [Bre2010] R. P. Brent, “Ramanujan and Euler’s Constant”, http://www.maths.anu.edu.au/~brent/pd/Euler_CARMA_10.pdf
- [BrentKung1978] Brent, R. P. and Kung, H. T. : Fast Algorithms for Manipulating Formal Power Series, *J. ACM* 25:4 (1978) 581–595
- [BuhlerCrandallSompolski1992] Buhler, J.P. and Crandall, R.E. and Sompolski, R.W. : Irregular primes to one million : *Math. Comp.* 59:2000 (1992) 717–722
- [CFG2017] F. Cléry, C. Faber, and G. van der Geer. “Covariants of binary sextics and vector-valued Siegel modular forms of genus two”, *Math. Ann.* 369 (2017), 1649–1669. <https://doi.org/10.1007/s00208-016-1510-2>

- [CFG2019] F. Cléry, C. Faber, and G. van der Geer. “Covariants of binary sextics and modular forms of degree 2 with character”, *Math. Comp.* 88 (2019), 2423–2441. <https://doi.org/10.1090/mcom/3412>
- [CGHJK1996] R. M. Corless, G. H. Gonnet, D. E. Hare, D. J. Jeffrey and D. E. Knuth, “On the Lambert W function”, *Advances in Computational Mathematics*, 5(1) (1996), 329–359
- [CP2005] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, second edition, Springer (2005).
- [Car1995] B. C. Carlson, “Numerical computation of real or complex elliptic integrals”. *Numerical Algorithms*, 10(1):13–26 (1995).
- [Car2004] J. Carette. “Understanding expression simplification.” *ISSAC ‘04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pp. 72–79. 2004. <https://doi.org/10.1145/1005285.1005298>
- [Chen2003] Zhuo Chen and John Greene : Some Comments on Baillie–PSW Pseudoprimes, *The Fibonacci Quarterly* 41:4 (2003) 334–344
- [Cho1999] T. Chow. “What is a closed-form number?”. *The American Mathematical Monthly* Volume 106, 1999 - Issue 5. <https://doi.org/10.1080/00029890.1999.12005066>
- [Coh1996] Cohen, Henri : *A course in computational algebraic number theory*, Springer, 1996
- [Coh2000] H. Cohen. *Advanced topics in computational number theory*. Springer, 2000. <https://doi.org/10.1007/978-1-4419-8489-0>
- [Col1971] Collins, George E. : *The Calculation of Multivariate Polynomial Resultants*, *SYMSAC ‘71*, ACM 1971 212–222
- [CraPom2005] Richard Crandall and Carl Pomerance: *Prime numbers: a computational perspective*. 2005.
- [DHBHS2004] B. Deconinck, M. Heil, A. Bobenko, M. van Hoeij, and M. Schmies, “Computing Riemann theta functions”, *Math. Comp.* 73:247 (2004), 1417–1442. <https://arxiv.org/abs/nlin/0206009>
- [DYF1999] A. Dzieciol, S. Yngve and P. O. Fröman, “Coulomb wave functions with complex values of the variable and the parameters”, *J. Math. Phys.* 40, 6145 (1999), <https://doi.org/10.1063/1.533083>
- [DelegliseNicolasZimmermann2009] Deleglise, Marc and Niclas, Jean-Louis and Zimmermann, Paul : Landau’s function for one million billions, *J. Théor. Nombres Bordeaux* 20:3 (2009) 625–671
- [DomKanTro1987] Domich, P. D. and Kannan, R. and Trotter, L. E. Jr. : Hermite Normal Form Computation Using Modulo Determinant Arithmetic, *Math. Operations Res.* (12) 1987 50–59
- [Dup2006] R. Dupont. “Moyenne arithmético-géométrique, suites de Borchardt et applications.” *These de doctorat, École polytechnique, Palaiseau* (2006). http://www.lix.polytechnique.fr/Labo/Regis.Dupont/these_soutenance.pdf
- [Dus1999] P. Dusart, “The k^{th} prime is greater than $k(\ln k + \ln \ln k - 1)$ for $k \geq 2$,” *Math. Comp.*, 68:225 (January 1999) 411–415.
- [EHJ2016] A. Enge, W. Hart and F. Johansson, “Short addition sequences for theta functions”, preprint (2016), <https://arxiv.org/abs/1608.06810>
- [EM2004] O. Espinosa and V. Moll, “A generalized polygamma function”, *Integral Transforms and Special Functions* (2004), 101–115.
- [EK2025] N. D. Elkies and J. Kieffer, “A uniform quasi-linear time algorithm for evaluating theta functions in any dimension”, in preparation.
- [FeiGal2013] J. Feitsma and W. Galway, “Tables of pseudoprimes and related data”, <http://www.cecm.sfu.ca/Pseudoprimes/index-2-to-64.html>
- [Fie2007] C. Fieker, “Sparse representation for cyclotomic fields”. *Experiment. Math.* Volume 16, Issue 4 (2007), 493–500. <https://doi.org/10.1080/10586458.2007.10129012>

- [FieHof2014] Fieker C. and Hofmann T.: “Computing in quotients of rings of integers” LMS Journal of Computation and Mathematics, 17(A), 349-365
- [Fil1992] S. Fillebrown, “Faster Computation of Bernoulli Numbers”, Journal of Algorithms 13 (1992) 431-445
- [ForJan2015] M. Forisek and J. Jancina, “Fast Primality Testing for Integers That Fit into a Machine Word”, SOFSEM 2015, <https://ceur-ws.org/Vol-1326/020-Forisek.pdf>
- [GCL1992] K. O. Geddes, S. R. Czapor and G. Labahn. *Algorithms for computer algebra*. Springer, 1992. <https://doi.org/10.1007/b102438>
- [GG2003] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, second edition, Cambridge University Press (2003)
- [GS2003] X. Gourdon and P. Sebah, “Numerical evaluation of the Riemann Zeta-function” (2003), <http://numbers.computation.free.fr/Constants/Miscellaneous/zetaevaluations.pdf>
- [GVL1996] G. H. Golub and C. F. Van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press (1996).
- [Gas2018] D. Gaspard, “Connection formulas between Coulomb wave functions” (2018), <https://arxiv.org/abs/1804.10976>
- [Gly2010] D. G. Glynn, “The permanent of a square matrix”, European Journal of Combinatorics, Volume 31, Issue 7, October 2010, Pages 1887-1891, <https://doi.org/10.1016/j.ejc.2010.01.010>
- [Gos1974] R. W. Gosper, “Acceleration of series”, MIT AI Memo no.304, (March-1974). <https://dspace.mit.edu/handle/1721.1/6088>
- [Got1959] E. Gottschling, “Explizite Bestimmung der Randflächen es Fundamentalbereiches der Modulgruppe zweiten Grades”, Math. Annalen 138 (1959), 103-124. <https://doi.org/10.1007/BF01342938>
- [GowWag2008] Jason Gower and Sam Wagstaff : “Square form factoring” Math. Comp. 77, 2008, pp 551-588, <https://doi.org/10.1090/S0025-5718-07-02010-8>
- [GraMol2010] Torbjörn Granlund and Niels Möller : Improved Division by Invariant Integers, <https://gmplib.org/~tege/division-paper.pdf>
- [GraMon1994] Törbjorn Granlund and Peter L. Montgomery : Division by Invariant Integers using Multiplication <https://gmplib.org/~tege/divcnst-pldi94.pdf>
- [HM2017] J. van der Hoeven and B. Mourrain. “Efficient certification of numeric solutions to eigenproblems”, MACIS 2017, 81-94, (2017), <https://hal.archives-ouvertes.fr/hal-01579079>
- [HS1967] E. Hansen and R. Smith, “Interval Arithmetic in Matrix Computations, Part II”, SIAM Journal of Numerical Analysis, 4(1):1-9 (1967). <https://doi.org/10.1137/0704001>
- [HZ2004] G. Hanrot and P. Zimmermann, “Newton Iteration Revisited” (2004), <http://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz>
- [HanZim2004] Guillaume Hanrot and Paul Zimmermann : Newton Iteration Revisited (2004) <https://www.loria.fr/~zimmerma/papers/fastnewton.ps.gz>
- [Har2010] D. Harvey, “A multimodular algorithm for computing Bernoulli numbers” (2010), Mathematics of Computation 79.272: 2361-2370
- [HZ2011] D. Harvey and P. Zimmermann, “Short division of long integers” (2011), Proceedings of the 20th Symposium on Computer Arithmetic (ARITH-20), July 25-27, 2011, pages 7-14. <https://web.maths.unsw.edu.au/~davidharvey/research/shortdiv.pdf>
- [Har2012] Hart, William B.. (2012) A one line factoring algorithm. Journal of the Australian Mathematical Society, Volume 92 (Number 1). pp. 61-69.
- [Har2015] W. B. Hart. “ANTIC: Algebraic number theory in C”. Computeralgebra-Rundbrief: Vol. 56, 2015

- [Har2018] W. B. Hart. “Algebraic number theory”. Unpublished manuscript, 2018.
- [Hart2010] W. B. Hart. “Fast library for number theory: an introduction.” International Congress on Mathematical Software. Springer, Berlin, Heidelberg, 2010. https://doi.org/10.1007/978-3-642-15582-6_18
- [Hen1956] Peter Henrici : “A Subroutine for Computations with Rational Numbers” J. ACM (1956), <https://doi.org/10.1145/320815.320818>
- [Hoe2001] J. van der Hoeven. “Fast evaluation of holonomic functions near and in regular singularities”, Journal of Symbolic Computation, 31(6):717-743 (2001).
- [Hoe2009] J. van der Hoeven, “Ball arithmetic”, Technical Report, HAL 00432152 (2009), <http://www.texmacs.org/joris/ball/ball-abs.html>
- [Hor1972] Ellis Horowitz : “Algorithms for Rational Function Arithmetic Operations” Annual ACM Symposium on Theory of Computing: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (Denver) (1972), <https://doi.org/10.1145/800152.804903>
- [Iliopoulos1989] Iliopoulos, C. S., Worst-Case Complexity Bounds on Algorithms for Computing the Canonical Structure of Finite Abelian Groups and the Hermite and Smith Normal Forms of an Integer Matrix : SIAM J. Computation 18:4 (1989) 658
- [Igu1972] J.-I. Igusa. *Theta functions*, Springer, 1972. <https://doi.org/10.1007/978-3-642-65315-5>
- [Igu1979] J.-I. Igusa, “On the ring of modular forms of degree two over $\mathbb{Z}$ ”, Amer. J. Math. 101:1 (1979), 149–183. <https://doi.org/10.2307/2373943>
- [JB2018] F. Johansson and I. Blagouchine. “Computing Stieltjes constants using complex integration”, preprint (2018), <https://arxiv.org/abs/1804.01679>
- [JM2018] F. Johansson and M. Mezzarobba, “Fast and rigorous arbitrary-precision computation of Gauss-Legendre quadrature nodes and weights”, preprint (2018), <https://arxiv.org/abs/1802.03948>
- [JR1999] D. Jeffrey and A. D. Rich. “Simplifying square roots of square roots by denesting”. Computer Algebra Systems: A Practical Guide, M.J. Wester, Ed., Wiley 1999.
- [Joh2012] F. Johansson, “Efficient implementation of the Hardy-Ramanujan-Rademacher formula”, LMS Journal of Computation and Mathematics, Volume 15 (2012), 341-359, <http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=8710297>
- [Joh2013] F. Johansson, “Rigorous high-precision computation of the Hurwitz zeta function and its derivatives”, Numerical Algorithms, <http://arxiv.org/abs/1309.2877> <https://doi.org/10.1007/s11075-014-9893-1>
- [Joh2014a] F. Johansson, *Fast and rigorous computation of special functions to high precision*, PhD thesis, RISC, Johannes Kepler University, Linz, 2014. <https://fredrikj.net/thesis/>
- [Joh2014b] F. Johansson, “Evaluating parametric holonomic sequences using rectangular splitting”, IS-SAC 2014, 256-263. <https://doi.org/10.1145/2608628.2608629>
- [Joh2014c] F. Johansson, “Efficient implementation of elementary functions in the medium-precision range”, <https://arxiv.org/abs/1410.7176>
- [Joh2015] F. Johansson, “Computing Bell numbers”, <https://fredrikj.net/blog/2015/08/computing-bell-numbers/>
- [Joh2015b] F. Johansson, “A fast algorithm for reversion of power series”, Math. Comp. 84 (2015), 475-484, <http://doi.org/10.1090/S0025-5718-2014-02857-3>
- [Joh2016] F. Johansson, “Computing hypergeometric functions rigorously”, preprint (2016), <https://arxiv.org/abs/1606.06977>
- [Joh2017] F. Johansson. “Arb: efficient arbitrary-precision midpoint-radius interval arithmetic”. IEEE Transactions on Computers, vol 66, issue 8, 2017, pp. 1281-1292. <https://doi.org/10.1109/TC.2017.2690633>

- [Joh2017a] F. Johansson. “Arb: efficient arbitrary-precision midpoint-radius interval arithmetic”, IEEE Transactions on Computers, 66(8):1281-1292 (2017). <https://doi.org/10.1109/TC.2017.2690633>
- [Joh2017b] F. Johansson, “Computing the Lambert W function in arbitrary-precision complex interval arithmetic”, preprint (2017), <https://arxiv.org/abs/1705.03266>
- [Joh2018a] F. Johansson, “Numerical integration in arbitrary-precision ball arithmetic”, preprint (2018), <https://arxiv.org/abs/1802.07942>
- [Joh2018b] F. Johansson and others, “mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.1.0)”, December 2018. <https://mpmath.org/>
- [JvdP2002] M. J. Jacobson Jr. and A. J. van der Poorten. “Computational aspects of NUCOMP.” In International Algorithmic Number Theory Symposium, pp. 120-133. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.
- [Kahan1991] Kahan, William: Computing a Real Cube Root. <https://csclub.uwaterloo.ca/~pbarfuss/qbrt.pdf>
- [KanBac1979] Kannan, R. and Bachem, A. : Polynomial algorithms for computing and the Smith and Hermite normal forms of an integer matrix, SIAM J. Computation vol. 9 (1979) 499–507
- [Kar1998] E. A. Karatsuba, “Fast evaluation of the Hurwitz zeta function and Dirichlet L-series”, Problems of Information Transmission 34:4 (1998), 342-353, http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=425&option_lang=eng
- [Knu1997] Knuth, D. E. The Art of Computer Programming, volume 2: Seminumerical algorithms, 1997
- [Kob2010] A. Kobel, “Certified Complex Numerical Root Finding”, Seminar on Computational Geometry and Geometric Computing (2010), http://www.mpi-inf.mpg.de/departments/d1/teaching/ss10/Seminar_CGGC/Slides/02_Kobel_NRS.pdf
- [Kri2013] A. Krishnamoorthy and D. Menon, “Matrix Inversion Using Cholesky Decomposition” Proc. of the International Conference on Signal Processing Algorithms, Architectures, Arrangements, and Applications (SPA-2013), pp. 70-72, 2013.
- [LT2016] H. Labrande and E. Thomé, “Computing theta functions in quasi-linear time in genus 2 and above”, ANTS XII, Kaiserslautern, LMS J. Comp. Math 19 (2016), 163–177. <https://doi.org/10.1112/S1461157016000309>
- [Leh1970] R. S. Lehman, “On the Distribution of Zeros of the Riemann Zeta-Function”, Proc. of the London Mathematical Society 20(3) (1970), 303-320, <https://doi.org/10.1112/plms/s3-20.2.303>
- [LukPatWil1996] R. F. Lukes and C. D. Patterson and H. C. Williams “Some results on pseudosquares” Math. Comp. 1996, no. 65, 361–372
- [Lüb2004] F. Lübeck, “Conway polynomials for finite fields”, RTWH Aachen, <https://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/index.html>, (accessed 2024-01-12)
- [ManWright1994] Y.-K. Man and F. J. Wright. “Fast polynomial dispersion computation and its application to indefinite summation”. Proceedings of the International Symposium on Symbolic and Algebraic Computation (1994), 175-180. <https://doi.org/10.1145/190347.190413>
- [MN2019] P. Molin and C. Neurohr, “Computing period matrices and the Abel–Jacobi map of superelliptic curves”, Math. Comp. 88:316 (2019), 847–888.
- [MP2006] M. Monagan and R. Pearce. “Rational simplification modulo a polynomial ideal”. Proceedings of the 2006 international symposium on Symbolic and algebraic computation - ISSAC ‘06. <https://doi.org/10.1145/1145768.1145809>
- [MPFR2012] The MPFR team, “MPFR Algorithms” (2012), <https://www.mpfr.org/algo.html>
- [MasRob1996] J. Massias and G. Robin, “Bornes effectives pour certaines fonctions concernant les nombres premiers,” J. Theorie Nombres Bordeaux, 8 (1996) 215-242.

- [Mic2007] N. Michel, “Precise Coulomb wave functions for a wide range of complex l , η and z ”, Computer Physics Communications, Volume 176, Issue 3, (2007), 232-249, <https://doi.org/10.1016/j.cpc.2006.10.004>
- [Miy2010] S. Miyajima, “Fast enclosure for all eigenvalues in generalized eigenvalue problems”, Journal of Computational and Applied Mathematics, 233 (2010), 2994-3004, <https://doi.org/10.1016/j.cam.2009.11.048>
- [Mos1971] J. Moses. “Algebraic simplification - a guide for the perplexed”. Proceedings of the second ACM symposium on Symbolic and algebraic manipulation (1971), 282-304. <https://doi.org/10.1145/362637.362648>
- [Mul2000] Thom Mulders : On Short Multiplications and Divisions, AAECC vol. 11 (2000) 69–88
- [Mum1983] D. Mumford, *Tata Lectures on Theta I*, Birkhäuser, 1983. <https://doi.org/10.1007/978-1-4899-2843-6>
- [Mum1984] D. Mumford, *Tata Lectures on Theta II*, Birkhäuser, 1984. <https://doi.org/10.1007/978-0-8176-4578-6>
- [NIST2012] National Institute of Standards and Technology, *Digital Library of Mathematical Functions* (2012), <https://dlmf.nist.gov/>
- [NakTurWil1997] Nakos, George and Turner, Peter and Williams, Robert : Fraction-free algorithms for linear and polynomial equations, ACM SIGSAM Bull. 31 (1997) 3 11–19
- [NW1978] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms for Computers and Calculators*. 2nd Edition, Academic Press, 1978.
- [Olv1997] F. Olver, *Asymptotics and special functions*, AKP Classics, AK Peters Ltd., Wellesley, MA, 1997. Reprint of the 1974 original.
- [PP2010] K. H. Pilehrood and T. H. Pilehrood. “Series acceleration formulas for beta values”, Discrete Mathematics and Theoretical Computer Science, DMTCS, 12 (2) (2010), 223-236, <https://hal.inria.fr/hal-00990465/>
- [PS1973] M. S. Paterson and L. J. Stockmeyer, “On the number of nonscalar multiplications necessary to evaluate polynomials”, SIAM J. Comput (1973)
- [PS1991] G. Pittaluga and L. Sacripante, “Inequalities for the zeros of the Airy functions”, SIAM J. Math. Anal. 22:1 (1991), 260-267.
- [Paterson1973] Michael S. Paterson and Larry J. Stockmeyer : On the number of nonscalar multiplications necessary to evaluate polynomials, SIAM Journal on Computing (1973)
- [PernetStein2010] Pernet, C. and Stein, W. : Fast computation of Hermite normal forms of random integer matrices ,J. Number Theory 130:17 (2010) 1675–1683
- [Pet1999] K. Petras, “On the computation of the Gauss-Legendre quadrature formula with a given precision”, Journal of Computational and Applied Mathematics 112 (1999), 253-267
- [Pla2011] D. J. Platt, “Computing degree 1 L-functions rigorously”, Ph.D. Thesis, University of Bristol (2011), <https://people.maths.bris.ac.uk/~madjp/thesis5.pdf>
- [Pla2017] D. J. Platt, “Isolating some non-trivial zeros of zeta”, Mathematics of Computation 86 (2017), 2449-2467, <https://doi.org/10.1090/mcom/3198>
- [RF1994] D. Richardson and J. Fitch. “The identity problem for elementary functions and constants”. IS-SAC ‘94: Proceedings of the international symposium on Symbolic and algebraic computation, August 1994, 285-290. <https://doi.org/10.1145/190347.190429>
- [Rad1973] H. Rademacher, *Topics in analytic number theory*, Springer, 1973.
- [Rademacher1937] Rademacher, Hans : On the partition function $p(n)$ Proc. London Math. Soc vol. 43 (1937) 241–254
- [Ric1992] D. Richardson. “The elementary constant problem”. ISSAC ‘92: Papers from the international symposium on Symbolic and algebraic computation, August 1992, 108-116. <https://doi.org/10.1145/143242.143284>

- [Ric1995] D. Richardson. “A simplified method of recognizing zero among elementary constants”. ISSAC ‘95: Proceedings of the 1995 international symposium on Symbolic and algebraic computation, April 1995, 104-109. <https://doi.org/10.1145/220346.220360>
- [Ric1997] D. Richardson. “How to recognize zero”. Journal of Symbolic Computation 24.6 (1997): 627-645. <https://doi.org/10.1006/jSCO.1997.0157>
- [Ric2007] D. Richardson. “Zero tests for constants in simple scientific computation”. Mathematics in Computer Science volume 1, pages 21-37 (2007). <https://doi.org/10.1007/s11786-007-0002-x>
- [Ric2009] D. Richardson. “Recognising zero among implicitly defined elementary numbers”. Preprint, 2009.
- [Rosowski2023] Rosowski, Andreas. “Fast commutative matrix algorithms”. Journal of Symbolic Computation 114 (2023): 302–321.
- [RosSch1962] Rosser, J. Barkley; Schoenfeld, Lowell: Approximate formulas for some functions of prime numbers. Illinois J. Math. 6 (1962), no. 1, 64–94.
- [Rum2010] S. M. Rump, “Verification methods: Rigorous results using floating-point arithmetic”, Acta Numerica 19 (2010), 287-449.
- [Rys1963] H. J. Ryser, *Combinatorial Mathematics*, The Carus Mathematical Monographs #14, The Mathematical Association of America (1963).
- [Smi2001] D. M. Smith, “Algorithm: Fortran 90 Software for Floating-Point Multiple Precision Arithmetic, Gamma and Related Functions”, Transactions on Mathematical Software 27 (2001) 377-387, <http://myweb.lmu.edu/dmsmith/toms2001.pdf>
- [SorWeb2016] Sorenson, Jonathan and Webster, Jonathan : Strong pseudoprimes to twelve prime bases. Math. Comp. 86 (2017), 985-1003, <https://doi.org/10.1090/mcom/3134>
- [Ste2002] A. Steel. “A new scheme for computing with algebraically closed fields”. In: Fieker C., Kohel D.R. (eds) Algorithmic Number Theory. ANTS 2002. Lecture Notes in Computer Science, vol 2369. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45455-1_38
- [Ste2010] A. Steel. “Computing with algebraically closed fields”. Journal of Symbolic Computation 45 (2010) 342-372. <https://doi.org/10.1016/j.jSC.2009.09.005>
- [Stehle2010] Stehlé, Damien : Floating-Point LLL: Theoretical and Practical Aspects, in Nguyen, Phong Q. and Vallée, Brigitte : The LLL Algorithm: Survey and Applications (2010) 179–213
- [Stein2007] Stein, William A.: Modular forms, a computational approach. American Mathematical Society. 2007
- [StoMul1998] Storjohann, Arne and Mulders, Thom : Fast algorithms for linear algebra modulo N : Algorithms—{ESA} ‘98 (Venice), Lecture Notes in Comput. Sci. 1461 139–150
- [Str2014] M. Streng, “Computing Igusa class polynomials”, Math. Comp. 83:285 (2014), 275–309. <https://doi.org/10.1090/S0025-5718-2013-02712-3>
- [Str1997] A. Strzebonski. “Computing in the field of complex algebraic numbers”. Journal of Symbolic Computation (1997) 24, 647-656. <https://doi.org/10.1006/jSCO.1997.0158>
- [Str2012] A. Strzebonski. “Real root isolation for exp-log-arctan functions”. Journal of Symbolic Computation 47 (2012) 282–314. <https://doi.org/10.1016/j.jSC.2011.11.004>
- [Sut2007] A. V. Sutherland. “Order computations in generic groups.” PhD diss., Massachusetts Institute of Technology, 2007.
- [Tak2000] D. Takahashi, “A fast algorithm for computing large Fibonacci numbers”, Information Processing Letters 75 (2000) 243-246, <http://www.ii.uni.wroc.pl/~lorys/IPL/article75-6-1.pdf>
- [ThullYap1990] Thull, K. and Yap, C. : A Unified Approach to HGCD Algorithms for Polynomials and Integers, (1990)

- [Tre2008] L. N. Trefethen, “Is Gauss Quadrature Better than Clenshaw-Curtis?”, SIAM Review, 50:1 (2008), 67-87, <https://doi.org/10.1137/060659831>
- [Tru2011] T. S. Trudgian, “Improvements to Turing’s method”, Mathematics of Computation 80 (2011), 2259-2279, <https://doi.org/10.1090/S0025-5718-2011-02470-1>
- [Tru2014] T. S. Trudgian, “An improved upper bound for the argument of the Riemann zeta-function on the critical line II”, Journal of Number Theory 134 (2014), 280-292, <https://doi.org/10.1016/j.jnt.2013.07.017>
- [Tur1953] A. M. Turing, “Some Calculations of the Riemann Zeta-Function”, Proc. of the London Mathematical Society 3(3) (1953), 99-117, <https://doi.org/10.1112/plms/s3-3.1.99>
- [Villard2007] Villard, Gilles : Certification of the QR Factor R and of Lattice Basis Reducedness, In proceedings of ACM International Symposium on Symbolic and Algebraic Computation (2007) 361–368 ACM Press.
- [Waksman1970] Waksman, Abraham, “On Winograd’s algorithm for inner products”, IEEE Transactions on Computers 19.04 (1970): 360-361.
- [WaktinsZeitlin1993] Watkins, W. and Zeitlin, J. : The minimal polynomial of $\cos(2\pi/n)$ The American Mathematical Monthly 100:5 (1993) 471–474
- [Wei2000] A. Weilert, “ $(1+i)$ -ary GCD computation in $\mathbb{Z}[i]$ as an analogue to the binary GCD algorithm”, Journal of Symbolic Computation 30.5 (2000): 605-617, <https://doi.org/10.1006/jsco.2000.0422>
- [Whiteman1956] Whiteman, A. L. : A sum connected with the series for the partition function, Pacific Journal of Mathematics 6:1 (1956) 159–176
- [Zip1985] R. Zippel. “Simplification of expressions involving radicals”. Journal of Symbolic Computation (1985) 1, 189-210. [https://doi.org/10.1016/S0747-7171\(85\)80014-6](https://doi.org/10.1016/S0747-7171(85)80014-6)
- [Zun2023] J. Zuniga, “Catalan’s constant fast convergent series”, <https://mathoverflow.net/q/424055>
- [Zun2023b] J. Zuniga, “Are these fast convergent series for $\log(2)$ and $\log(3)$ already known and proven?”, <https://math.stackexchange.com/q/4854073>
- [vHP2012] M. van Hoeij and V. Pal. “Isomorphisms of algebraic number fields”. Journal de Théorie des Nombres de Bordeaux, Vol. 24, No. 2 (2012), pp. 293-305. <https://doi.org/10.2307/43973105>
- [vdH1995] J. van der Hoeven, “Automatic numerical expansions”. Proc. of the conference Real numbers and computers (1995), 261-274. <https://www.texmacs.org/joris/ane/ane-abs.html>
- [vdH2006] J. van der Hoeven, “Computations with effective real numbers”. Theoretical Computer Science, Volume 351, Issue 1, 14 February 2006, Pages 52-60. <https://doi.org/10.1016/j.tcs.2005.09.060>

Symbols

- `__FLINT_RELEASE` (*C macro*), 15
- `__FLINT_VERSION` (*C macro*), 15
- `__FLINT_VERSION_MINOR` (*C macro*), 15
- `__FLINT_VERSION_PATCHLEVEL` (*C macro*), 15
- `_acb_dirichlet_definite_hardy_z` (*C function*), 791
- `_acb_dirichlet_euler_product_real_ui` (*C function*), 789
- `_acb_dirichlet_exact_zeta_nzeros` (*C function*), 792
- `_acb_dirichlet_hardy_theta_series` (*C function*), 790
- `_acb_dirichlet_hardy_z_series` (*C function*), 791
- `_acb_dirichlet_isolate_gram_hardy_z_zero` (*C function*), 791
- `_acb_dirichlet_isolate_rosser_hardy_z_zero` (*C function*), 791
- `_acb_dirichlet_isolate_turing_hardy_z_zero` (*C function*), 791
- `_acb_dirichlet_l_series` (*C function*), 790
- `_acb_dirichlet_platt_local_hardy_z_zeros` (*C function*), 793
- `_acb_dirichlet_refine_hardy_z_zero` (*C function*), 792
- `_acb_elliptic_k_series` (*C function*), 742
- `_acb_elliptic_p_series` (*C function*), 746
- `_acb_hypgeom_airy_series` (*C function*), 724
- `_acb_hypgeom_beta_lower_series` (*C function*), 726
- `_acb_hypgeom_chi_series` (*C function*), 728
- `_acb_hypgeom_ci_series` (*C function*), 727
- `_acb_hypgeom_coulomb_series` (*C function*), 725
- `_acb_hypgeom_ei_series` (*C function*), 727
- `_acb_hypgeom_erf_series` (*C function*), 720
- `_acb_hypgeom_erfc_series` (*C function*), 721
- `_acb_hypgeom_erfi_series` (*C function*), 721
- `_acb_hypgeom_fresnel_series` (*C function*), 721
- `_acb_hypgeom_gamma_lower_series` (*C function*), 726
- `_acb_hypgeom_gamma_upper_series` (*C function*), 725
- `_acb_hypgeom_li_series` (*C function*), 728
- `_acb_hypgeom_shi_series` (*C function*), 728
- `_acb_hypgeom_si_series` (*C function*), 727
- `_acb_mat_charpoly` (*C function*), 712
- `_acb_mat_companion` (*C function*), 712
- `_acb_mat_diag_prod` (*C function*), 712
- `_acb_mat_vector_mul_col` (*C function*), 709
- `_acb_mat_vector_mul_row` (*C function*), 709
- `_acb_modular_theta_series` (*C function*), 751
- `_acb_poly_add` (*C function*), 674
- `_acb_poly_agm1_series` (*C function*), 686
- `_acb_poly_atan_series` (*C function*), 681
- `_acb_poly_binomial_transform` (*C function*), 679
- `_acb_poly_binomial_transform_basecase` (*C function*), 679
- `_acb_poly_binomial_transform_convolution` (*C function*), 679
- `_acb_poly_borel_transform` (*C function*), 679
- `_acb_poly_compose` (*C function*), 676
- `_acb_poly_compose_series` (*C function*), 676
- `_acb_poly_cos_pi_series` (*C function*), 682
- `_acb_poly_cos_series` (*C function*), 682
- `_acb_poly_cosh_series` (*C function*), 683
- `_acb_poly_cot_pi_series` (*C function*), 682
- `_acb_poly_derivative` (*C function*), 679
- `_acb_poly_digamma_series` (*C function*), 683
- `_acb_poly_div` (*C function*), 676
- `_acb_poly_div_root` (*C function*), 676
- `_acb_poly_div_series` (*C function*), 676
- `_acb_poly_divrem` (*C function*), 676
- `_acb_poly_elliptic_k_series` (*C function*), 686
- `_acb_poly_elliptic_p_series` (*C function*), 686
- `_acb_poly_erf_series` (*C function*), 685
- `_acb_poly_evaluate` (*C function*), 677
- `_acb_poly_evaluate2` (*C function*), 677
- `_acb_poly_evaluate2_horner` (*C function*), 677
- `_acb_poly_evaluate2_rectangular` (*C function*), 677
- `_acb_poly_evaluate_horner` (*C function*), 677
- `_acb_poly_evaluate_rectangular` (*C function*), 677
- `_acb_poly_evaluate_vec_fast` (*C function*), 678
- `_acb_poly_evaluate_vec_fast_precomp` (*C function*), 678
- `_acb_poly_evaluate_vec_iter` (*C function*), 678
- `_acb_poly_exp_pi_i_series` (*C function*), 681
- `_acb_poly_exp_series` (*C function*), 681
- `_acb_poly_exp_series_basecase` (*C function*),

681
_acb_poly_find_roots (*C function*), 686
_acb_poly_find_roots_double (*C function*), 687
_acb_poly_gamma_series (*C function*), 683
_acb_poly_graeffe_transform (*C function*), 679
_acb_poly_integral (*C function*), 679
_acb_poly_interpolate_barycentric (*C function*), 678
_acb_poly_interpolate_fast (*C function*), 678
_acb_poly_interpolate_fast_precomp (*C function*), 678
_acb_poly_interpolate_newton (*C function*), 678
_acb_poly_interpolation_weights (*C function*), 678
_acb_poly_inv_borel_transform (*C function*), 679
_acb_poly_inv_series (*C function*), 675
_acb_poly_lambertw_series (*C function*), 683
_acb_poly_lgamma_series (*C function*), 683
_acb_poly_loglp_series (*C function*), 681
_acb_poly_log_series (*C function*), 681
_acb_poly_majorant (*C function*), 674
_acb_poly_mul (*C function*), 675
_acb_poly_mullow (*C function*), 674
_acb_poly_mullow_classical (*C function*), 674
_acb_poly_mullow_transpose (*C function*), 674
_acb_poly_mullow_transpose_gauss (*C function*), 674
_acb_poly_mulmid (*C function*), 675
_acb_poly_mulmid_classical (*C function*), 675
_acb_poly_mulmid_transpose (*C function*), 675
_acb_poly_normalise (*C function*), 671
_acb_poly_nth_derivative (*C function*), 679
_acb_poly_overlaps (*C function*), 673
_acb_poly_polylog_cpx (*C function*), 685
_acb_poly_polylog_cpx_small (*C function*), 685
_acb_poly_polylog_cpx_zeta (*C function*), 685
_acb_poly_polylog_series (*C function*), 685
_acb_poly_pow_acb_series (*C function*), 680
_acb_poly_pow_series (*C function*), 680
_acb_poly_pow_ui (*C function*), 680
_acb_poly_pow_ui_trunc_binexp (*C function*), 680
_acb_poly_powsum_one_series_sieved (*C function*), 684
_acb_poly_powsum_series_naive (*C function*), 684
_acb_poly_powsum_series_naive_threaded (*C function*), 684
_acb_poly_product_roots (*C function*), 677
_acb_poly_rem (*C function*), 676
_acb_poly_revert_series (*C function*), 676
_acb_poly_rgamma_series (*C function*), 683
_acb_poly_rising_ui_series (*C function*), 683
_acb_poly_root_bound_fujiwara (*C function*), 686
_acb_poly_root_inclusion (*C function*), 686
_acb_poly_rsqrts_series (*C function*), 681
_acb_poly_set_length (*C function*), 671
_acb_poly_shift_left (*C function*), 672
_acb_poly_shift_right (*C function*), 672
_acb_poly_sin_cos_pi_series (*C function*), 682
_acb_poly_sin_cos_series (*C function*), 681
_acb_poly_sin_pi_series (*C function*), 682
_acb_poly_sin_series (*C function*), 682
_acb_poly_sinc_pi_series (*C function*), 683
_acb_poly_sinc_series (*C function*), 683
_acb_poly_sinh_cosh_series (*C function*), 682
_acb_poly_sinh_cosh_series_basecase (*C function*), 682
_acb_poly_sinh_cosh_series_exponential (*C function*), 682
_acb_poly_sinh_series (*C function*), 682
_acb_poly_sqrt_series (*C function*), 680
_acb_poly_sub (*C function*), 674
_acb_poly_tan_series (*C function*), 682
_acb_poly_taylor_shift (*C function*), 676
_acb_poly_tree_alloc (*C function*), 677
_acb_poly_tree_build (*C function*), 677
_acb_poly_tree_free (*C function*), 677
_acb_poly_validate_real_roots (*C function*), 687
_acb_poly_validate_roots (*C function*), 686
_acb_poly_zeta_cpx_series (*C function*), 685
_acb_poly_zeta_em_bound (*C function*), 684
_acb_poly_zeta_em_bound1 (*C function*), 684
_acb_poly_zeta_em_choose_param (*C function*), 684
_acb_poly_zeta_em_sum (*C function*), 684
_acb_poly_zeta_em_tail_bsplint (*C function*), 684
_acb_poly_zeta_em_tail_naive (*C function*), 684
_acb_poly_zeta_series (*C function*), 685
_acb_vec_add (*C function*), 653
_acb_vec_add_error_arf_vec (*C function*), 654
_acb_vec_add_error_mag_vec (*C function*), 654
_acb_vec_allocated_bytes (*C function*), 639
_acb_vec_bits (*C function*), 654
_acb_vec_clear (*C function*), 639
_acb_vec_contains (*C function*), 653
_acb_vec_equal (*C function*), 653
_acb_vec_estimate_allocated_bytes (*C function*), 639
_acb_vec_get_imag (*C function*), 653
_acb_vec_get_real (*C function*), 653
_acb_vec_get_unique_fmpz_vec (*C function*), 654
_acb_vec_indeterminate (*C function*), 654
_acb_vec_init (*C function*), 639
_acb_vec_is_finite (*C function*), 653
_acb_vec_is_real (*C function*), 653
_acb_vec_is_zero (*C function*), 653
_acb_vec_neg (*C function*), 653
_acb_vec_overlaps (*C function*), 653

_acb_vec_printd (*C function*), 654
 _acb_vec_printn (*C function*), 654
 _acb_vec_scalar_addmul (*C function*), 653
 _acb_vec_scalar_div (*C function*), 654
 _acb_vec_scalar_div_arb (*C function*), 654
 _acb_vec_scalar_div_fmpz (*C function*), 654
 _acb_vec_scalar_div_ui (*C function*), 654
 _acb_vec_scalar_mul (*C function*), 653
 _acb_vec_scalar_mul_2exp_si (*C function*), 654
 _acb_vec_scalar_mul_arb (*C function*), 654
 _acb_vec_scalar_mul_fmpz (*C function*), 654
 _acb_vec_scalar_mul_onei (*C function*), 654
 _acb_vec_scalar_mul_ui (*C function*), 653
 _acb_vec_scalar_submul (*C function*), 653
 _acb_vec_set (*C function*), 653
 _acb_vec_set_powers (*C function*), 654
 _acb_vec_set_real_imag (*C function*), 653
 _acb_vec_set_round (*C function*), 653
 _acb_vec_sort_pretty (*C function*), 654
 _acb_vec_sqr (*C function*), 654
 _acb_vec_sub (*C function*), 653
 _acb_vec_swap (*C function*), 653
 _acb_vec_trim (*C function*), 654
 _acb_vec_unit_roots (*C function*), 654
 _acb_vec_zero (*C function*), 653
 _aprcl_config (*C type*), 287
 _aprcl_is_prime_gauss (*C function*), 287
 _aprcl_is_prime_jacobi (*C function*), 287
 _arb_atan_gauss_p_ensure_cached (*C function*), 636
 _arb_atan_sum_bs_powtab (*C function*), 635
 _arb_atan_sum_bs_simple (*C function*), 635
 _arb_atan_taylor_naive (*C function*), 634
 _arb_atan_taylor_rs (*C function*), 634
 _arb_exp_sum_bs_powtab (*C function*), 635
 _arb_exp_sum_bs_simple (*C function*), 635
 _arb_exp_taylor_bound (*C function*), 634
 _arb_exp_taylor_naive (*C function*), 634
 _arb_exp_taylor_rs (*C function*), 634
 _arb_fmpz_poly_evaluate_acb (*C function*), 688
 _arb_fmpz_poly_evaluate_acb_horner (*C function*), 688
 _arb_fmpz_poly_evaluate_acb_rectangular (*C function*), 688
 _arb_fmpz_poly_evaluate_arb (*C function*), 688
 _arb_fmpz_poly_evaluate_arb_horner (*C function*), 688
 _arb_fmpz_poly_evaluate_arb_rectangular (*C function*), 688
 _arb_gamma_upper_fmpz_step_bsplitt (*C function*), 737
 _arb_get_mpn_fixed_mod_log2 (*C function*), 634
 _arb_get_mpn_fixed_mod_pi4 (*C function*), 634
 _arb_hypgeom_airy_series (*C function*), 739
 _arb_hypgeom_beta_lower_series (*C function*), 737
 _arb_hypgeom_chi_series (*C function*), 738
 _arb_hypgeom_ci_2f3 (*C function*), 738
 _arb_hypgeom_ci_asymp (*C function*), 738
 _arb_hypgeom_ci_series (*C function*), 738
 _arb_hypgeom_coulomb_series (*C function*), 740
 _arb_hypgeom_ei_series (*C function*), 738
 _arb_hypgeom_erf_series (*C function*), 735
 _arb_hypgeom_erfc_series (*C function*), 735
 _arb_hypgeom_erfi_series (*C function*), 735
 _arb_hypgeom_fresnel_series (*C function*), 736
 _arb_hypgeom_gamma_lower_fmpz_0_bsplitt (*C function*), 737
 _arb_hypgeom_gamma_lower_fmpz_0_choose_N (*C function*), 737
 _arb_hypgeom_gamma_lower_series (*C function*), 736
 _arb_hypgeom_gamma_lower_sum_rs_1 (*C function*), 737
 _arb_hypgeom_gamma_stirling_term_bounds (*C function*), 733
 _arb_hypgeom_gamma_upper_fmpz_inf_bsplitt (*C function*), 737
 _arb_hypgeom_gamma_upper_fmpz_inf_choose_N (*C function*), 737
 _arb_hypgeom_gamma_upper_series (*C function*), 736
 _arb_hypgeom_gamma_upper_singular_si_bsplitt (*C function*), 737
 _arb_hypgeom_gamma_upper_singular_si_choose_N (*C function*), 737
 _arb_hypgeom_gamma_upper_sum_rs_1 (*C function*), 737
 _arb_hypgeom_li_series (*C function*), 739
 _arb_hypgeom_rising_coeffs_1 (*C function*), 733
 _arb_hypgeom_rising_coeffs_2 (*C function*), 733
 _arb_hypgeom_rising_coeffs_fmpz (*C function*), 733
 _arb_hypgeom_shi_series (*C function*), 738
 _arb_hypgeom_si_1f2 (*C function*), 738
 _arb_hypgeom_si_asymp (*C function*), 738
 _arb_hypgeom_si_series (*C function*), 738
 _arb_log_p_ensure_cached (*C function*), 636
 _arb_mat_addmul_rad_mag_fast (*C function*), 698
 _arb_mat_charpoly (*C function*), 702
 _arb_mat_cholesky_banachiewicz (*C function*), 701
 _arb_mat_companion (*C function*), 702
 _arb_mat_diag_prod (*C function*), 703
 _arb_mat_ldl_golub_and_van_loan (*C function*), 702
 _arb_mat_ldl_inplace (*C function*), 702
 _arb_mat_vector_mul_col (*C function*), 699
 _arb_mat_vector_mul_row (*C function*), 699
 _arb_poly_acos_series (*C function*), 666
 _arb_poly_add (*C function*), 657
 _arb_poly_asin_series (*C function*), 666
 _arb_poly_atan_series (*C function*), 666

`_arb_poly_binomial_transform` (*C function*), 664

`_arb_poly_binomial_transform_basecase` (*C function*), 664

`_arb_poly_binomial_transform_convolution` (*C function*), 664

`_arb_poly_borel_transform` (*C function*), 664

`_arb_poly_compose` (*C function*), 660

`_arb_poly_compose_series` (*C function*), 660

`_arb_poly_cos_pi_series` (*C function*), 667

`_arb_poly_cos_series` (*C function*), 667

`_arb_poly_cosh_series` (*C function*), 668

`_arb_poly_cot_pi_series` (*C function*), 667

`_arb_poly_derivative` (*C function*), 663

`_arb_poly_digamma_series` (*C function*), 668

`_arb_poly_div` (*C function*), 659

`_arb_poly_div_root` (*C function*), 660

`_arb_poly_div_series` (*C function*), 659

`_arb_poly_divrem` (*C function*), 659

`_arb_poly_evaluate` (*C function*), 660

`_arb_poly_evaluate2` (*C function*), 661

`_arb_poly_evaluate2_acb` (*C function*), 661

`_arb_poly_evaluate2_acb_horner` (*C function*), 661

`_arb_poly_evaluate2_acb_rectangular` (*C function*), 661

`_arb_poly_evaluate2_horner` (*C function*), 661

`_arb_poly_evaluate2_rectangular` (*C function*), 661

`_arb_poly_evaluate_acb` (*C function*), 661

`_arb_poly_evaluate_acb_horner` (*C function*), 661

`_arb_poly_evaluate_acb_rectangular` (*C function*), 661

`_arb_poly_evaluate_horner` (*C function*), 660

`_arb_poly_evaluate_rectangular` (*C function*), 660

`_arb_poly_evaluate_vec_fast` (*C function*), 662

`_arb_poly_evaluate_vec_fast_precomp` (*C function*), 662

`_arb_poly_evaluate_vec_iter` (*C function*), 662

`_arb_poly_exp_series` (*C function*), 666

`_arb_poly_exp_series_basecase` (*C function*), 666

`_arb_poly_gamma_series` (*C function*), 668

`_arb_poly_graeffe_transform` (*C function*), 664

`_arb_poly_integral` (*C function*), 663

`_arb_poly_interpolate_barycentric` (*C function*), 663

`_arb_poly_interpolate_fast` (*C function*), 663

`_arb_poly_interpolate_fast_precomp` (*C function*), 663

`_arb_poly_interpolate_newton` (*C function*), 663

`_arb_poly_interpolation_weights` (*C function*), 663

`_arb_poly_inv_borel_transform` (*C function*), 664

`_arb_poly_inv_series` (*C function*), 659

`_arb_poly_lambertw_series` (*C function*), 668

`_arb_poly_lgamma_series` (*C function*), 668

`_arb_poly_log1p_series` (*C function*), 666

`_arb_poly_log_series` (*C function*), 666

`_arb_poly_majorant` (*C function*), 657

`_arb_poly_mul` (*C function*), 659

`_arb_poly_mullo` (*C function*), 658

`_arb_poly_mullo_block` (*C function*), 658

`_arb_poly_mullo_classical` (*C function*), 658

`_arb_poly_mulmid` (*C function*), 659

`_arb_poly_mulmid_block` (*C function*), 659

`_arb_poly_mulmid_classical` (*C function*), 659

`_arb_poly_newton_convergence_factor` (*C function*), 670

`_arb_poly_newton_refine_root` (*C function*), 670

`_arb_poly_newton_step` (*C function*), 670

`_arb_poly_normalise` (*C function*), 655

`_arb_poly_nth_derivative` (*C function*), 663

`_arb_poly_overlaps` (*C function*), 657

`_arb_poly_pow_arb_series` (*C function*), 665

`_arb_poly_pow_series` (*C function*), 665

`_arb_poly_pow_ui` (*C function*), 665

`_arb_poly_pow_ui_trunc_binexp` (*C function*), 665

`_arb_poly_product_roots` (*C function*), 662

`_arb_poly_product_roots_complex` (*C function*), 662

`_arb_poly_rem` (*C function*), 659

`_arb_poly_revert_series` (*C function*), 660

`_arb_poly_rgamma_series` (*C function*), 668

`_arb_poly_riemann_siegel_theta_series` (*C function*), 669

`_arb_poly_riemann_siegel_z_series` (*C function*), 669

`_arb_poly_rising_ui_series` (*C function*), 668

`_arb_poly_root_bound_fujiwara` (*C function*), 670

`_arb_poly_rsqr` (*C function*), 665

`_arb_poly_set_length` (*C function*), 655

`_arb_poly_shift_left` (*C function*), 656

`_arb_poly_shift_right` (*C function*), 656

`_arb_poly_sin_cos_pi_series` (*C function*), 667

`_arb_poly_sin_cos_series` (*C function*), 666

`_arb_poly_sin_pi_series` (*C function*), 667

`_arb_poly_sin_series` (*C function*), 667

`_arb_poly_sinc_pi_series` (*C function*), 668

`_arb_poly_sinc_series` (*C function*), 668

`_arb_poly_sinh_cosh_series` (*C function*), 667

`_arb_poly_sinh_cosh_series_basecase` (*C function*), 667

`_arb_poly_sinh_cosh_series_exponential` (*C function*), 667

`_arb_poly_sinh_series` (*C function*), 667

`_arb_poly_sqrt_series` (*C function*), 665

`_arb_poly_sub` (*C function*), 657

`_arb_poly_swinnerton_dyer_ui` (*C function*), 670
`_arb_poly_tan_series` (*C function*), 667
`_arb_poly_taylor_shift` (*C function*), 660
`_arb_poly_tree_alloc` (*C function*), 662
`_arb_poly_tree_build` (*C function*), 662
`_arb_poly_tree_free` (*C function*), 662
`_arb_sin_cos_taylor_naive` (*C function*), 634
`_arb_sin_cos_taylor_rs` (*C function*), 634
`_arb_vec_add` (*C function*), 637
`_arb_vec_add_error_arf_vec` (*C function*), 638
`_arb_vec_add_error_mag_vec` (*C function*), 638
`_arb_vec_allocated_bytes` (*C function*), 614
`_arb_vec_bits` (*C function*), 637
`_arb_vec_clear` (*C function*), 614
`_arb_vec_contains` (*C function*), 637
`_arb_vec_equal` (*C function*), 637
`_arb_vec_estimate_allocated_bytes` (*C function*), 614
`_arb_vec_get_mag` (*C function*), 637
`_arb_vec_get_unique_fmpz_vec` (*C function*), 638
`_arb_vec_indeterminate` (*C function*), 638
`_arb_vec_init` (*C function*), 614
`_arb_vec_is_finite` (*C function*), 637
`_arb_vec_is_zero` (*C function*), 637
`_arb_vec_neg` (*C function*), 637
`_arb_vec_overlaps` (*C function*), 637
`_arb_vec_printd` (*C function*), 638
`_arb_vec_printn` (*C function*), 638
`_arb_vec_scalar_addmul` (*C function*), 637
`_arb_vec_scalar_div` (*C function*), 637
`_arb_vec_scalar_mul` (*C function*), 637
`_arb_vec_scalar_mul_2exp_si` (*C function*), 637
`_arb_vec_scalar_mul_fmpz` (*C function*), 637
`_arb_vec_set` (*C function*), 637
`_arb_vec_set_powers` (*C function*), 637
`_arb_vec_set_round` (*C function*), 637
`_arb_vec_sub` (*C function*), 637
`_arb_vec_swap` (*C function*), 637
`_arb_vec_trim` (*C function*), 638
`_arb_vec_zero` (*C function*), 637
`_arf_get_integer_mpn` (*C function*), 610
`_arf_interval_vec_clear` (*C function*), 800
`_arf_interval_vec_init` (*C function*), 800
`_arf_set_mpn_fixed` (*C function*), 610
`_arf_set_round_mpn` (*C function*), 611
`_arf_set_round_ui` (*C function*), 610
`_arf_set_round_uiui` (*C function*), 610
`_arith_bernoulli_number` (*C function*), 294
`_arith_bernoulli_number_vec` (*C function*), 294
`_arith_bernoulli_number_vec_multi_mod` (*C function*), 294
`_arith_bernoulli_number_vec_recursive` (*C function*), 294
`_arith_harmonic_number` (*C function*), 291
`_bernoulli_fmpq_ui` (*C function*), 795
`_bernoulli_fmpq_ui_multi_mod` (*C function*), 795
`_bernoulli_fmpq_ui_zeta` (*C function*), 795
`_ca_make_field_element` (*C function*), 863
`_ca_make_fmpq` (*C function*), 863
`_ca_mat_ca_poly_evaluate` (*C function*), 876
`_ca_mat_charpoly` (*C function*), 880
`_ca_mat_charpoly_berkowitz` (*C function*), 880
`_ca_mat_charpoly_danilevsky` (*C function*), 880
`_ca_poly_add` (*C function*), 869
`_ca_poly_check_equal` (*C function*), 869
`_ca_poly_compose` (*C function*), 870
`_ca_poly_derivative` (*C function*), 871
`_ca_poly_div_series` (*C function*), 871
`_ca_poly_divrem` (*C function*), 870
`_ca_poly_divrem_basecase` (*C function*), 870
`_ca_poly_evaluate` (*C function*), 870
`_ca_poly_evaluate_horner` (*C function*), 870
`_ca_poly_exp_series` (*C function*), 871
`_ca_poly_gcd` (*C function*), 871
`_ca_poly_gcd_euclidean` (*C function*), 871
`_ca_poly_integral` (*C function*), 871
`_ca_poly_inv_series` (*C function*), 871
`_ca_poly_log_series` (*C function*), 871
`_ca_poly_mul` (*C function*), 869
`_ca_poly_mullo` (*C function*), 870
`_ca_poly_normalise` (*C function*), 867
`_ca_poly_pow_ui` (*C function*), 870
`_ca_poly_pow_ui_trunc` (*C function*), 870
`_ca_poly_reverse` (*C function*), 869
`_ca_poly_roots` (*C function*), 872
`_ca_poly_set_length` (*C function*), 867
`_ca_poly_set_roots` (*C function*), 872
`_ca_poly_shift_left` (*C function*), 869
`_ca_poly_shift_right` (*C function*), 869
`_ca_poly_sub` (*C function*), 869
`_ca_poly_vec_clear` (*C function*), 872
`_ca_poly_vec_fit_length` (*C function*), 872
`_ca_poly_vec_init` (*C function*), 872
`_ca_vec_add` (*C function*), 865
`_ca_vec_check_is_zero` (*C function*), 866
`_ca_vec_clear` (*C function*), 864
`_ca_vec_fit_length` (*C function*), 864
`_ca_vec_fmpz_vec_get_fmpz_vec_den` (*C function*), 866
`_ca_vec_fmpz_vec_is_fmpz_vec` (*C function*), 866
`_ca_vec_init` (*C function*), 864
`_ca_vec_is_fmpz_vec` (*C function*), 866
`_ca_vec_neg` (*C function*), 865
`_ca_vec_scalar_addmul_ca` (*C function*), 865
`_ca_vec_scalar_div_ca` (*C function*), 865
`_ca_vec_scalar_mul_ca` (*C function*), 865
`_ca_vec_scalar_submul_ca` (*C function*), 865
`_ca_vec_set` (*C function*), 865
`_ca_vec_set_fmpz_vec_div_fmpz` (*C function*), 866
`_ca_vec_sub` (*C function*), 865

`_ca_vec_swap` (*C function*), 864
`_ca_vec_zero` (*C function*), 865
`_d_vec_add` (*C function*), 1091
`_d_vec_approx_equal` (*C function*), 1090
`_d_vec_clear` (*C function*), 1090
`_d_vec_dot` (*C function*), 1091
`_d_vec_dot_heuristic` (*C function*), 1091
`_d_vec_dot_thrice` (*C function*), 1091
`_d_vec_equal` (*C function*), 1090
`_d_vec_init` (*C function*), 1090
`_d_vec_is_approx_zero` (*C function*), 1090
`_d_vec_is_zero` (*C function*), 1090
`_d_vec_mul_2exp` (*C function*), 1091
`_d_vec_norm` (*C function*), 1091
`_d_vec_randtest` (*C function*), 1090
`_d_vec_set` (*C function*), 1090
`_d_vec_sub` (*C function*), 1091
`_d_vec_zero` (*C function*), 1090
`_dirichlet_char_exp` (*C function*), 526
`_fexpr_vec_clear` (*C function*), 890
`_fexpr_vec_init` (*C function*), 890
`_fexpr_vec_sort_fast` (*C function*), 895
`_fft_mulmod_2exp1` (*C function*), 304
`_flint_mpn_addmod_2` (*C function*), 280
`_flint_mpn_mulhigh_basecase` (*C function*), 281
`_flint_mpn_mulhigh_n_mul` (*C function*), 281
`_flint_mpn_mulhigh_n_mulders` (*C function*), 281
`_flint_mpn_mulhigh_n_mulders_recursive` (*C function*), 281
`_flint_mpn_mullow_n` (*C function*), 282
`_flint_mpn_mullow_n_mul` (*C function*), 282
`_flint_mpn_mullow_n_mulders` (*C function*), 282
`_flint_mpn_mullow_n_mulders_recursive` (*C function*), 282
`_flint_mpn_sqrhigh_basecase` (*C function*), 281
`_flint_mpn_sqrhigh_mulders` (*C function*), 281
`_flint_mpn_sqrhigh_mulders_recursive` (*C function*), 281
`_flint_mpn_sqrhigh_sqr` (*C function*), 281
`_fmpq_add` (*C function*), 322
`_fmpq_add_fmpz` (*C function*), 322
`_fmpq_add_si` (*C function*), 322
`_fmpq_add_small` (*C function*), 323
`_fmpq_add_ui` (*C function*), 322
`_fmpq_addmul` (*C function*), 323
`_fmpq_canonicalise` (*C function*), 318
`_fmpq_div` (*C function*), 322
`_fmpq_fprint` (*C function*), 321
`_fmpq_gcd` (*C function*), 323
`_fmpq_gcd_cofactors` (*C function*), 323
`_fmpq_get_str` (*C function*), 320
`_fmpq_harmonic_ui` (*C function*), 326
`_fmpq_is_canonical` (*C function*), 318
`_fmpq_mat_charpoly` (*C function*), 336
`_fmpq_mat_minpoly` (*C function*), 336
`_fmpq_mod_fmpz` (*C function*), 324
`_fmpq_mul` (*C function*), 322
`_fmpq_mul_si` (*C function*), 322
`_fmpq_mul_small` (*C function*), 323
`_fmpq_mul_ui` (*C function*), 322
`_fmpq_next_calkin_wilf` (*C function*), 324
`_fmpq_next_minimal` (*C function*), 324
`_fmpq_next_signed_calkin_wilf` (*C function*), 325
`_fmpq_next_signed_minimal` (*C function*), 324
`_fmpq_poly_add` (*C function*), 342
`_fmpq_poly_add_can` (*C function*), 342
`_fmpq_poly_add_series` (*C function*), 342
`_fmpq_poly_add_series_can` (*C function*), 342
`_fmpq_poly_asin_series` (*C function*), 352
`_fmpq_poly_asinh_series` (*C function*), 352
`_fmpq_poly_atan_series` (*C function*), 351
`_fmpq_poly_atanh_series` (*C function*), 352
`_fmpq_poly_canonicalise` (*C function*), 337
`_fmpq_poly_cmp` (*C function*), 341
`_fmpq_poly_compose` (*C function*), 355
`_fmpq_poly_compose_series` (*C function*), 356
`_fmpq_poly_compose_series_brent_kung` (*C function*), 356
`_fmpq_poly_compose_series_horner` (*C function*), 356
`_fmpq_poly_content` (*C function*), 358
`_fmpq_poly_cos_series` (*C function*), 352
`_fmpq_poly_cosh_series` (*C function*), 353
`_fmpq_poly_derivative` (*C function*), 350
`_fmpq_poly_discriminant` (*C function*), 349
`_fmpq_poly_div` (*C function*), 346
`_fmpq_poly_div_series` (*C function*), 348
`_fmpq_poly_divides` (*C function*), 347
`_fmpq_poly_divrem` (*C function*), 346
`_fmpq_poly_equal_trunc` (*C function*), 341
`_fmpq_poly_evaluate_fmpq` (*C function*), 354
`_fmpq_poly_evaluate_fmpz` (*C function*), 354
`_fmpq_poly_exp_expinv_series` (*C function*), 351
`_fmpq_poly_exp_series` (*C function*), 351
`_fmpq_poly_fprint` (*C function*), 359
`_fmpq_poly_fprint_pretty` (*C function*), 359
`_fmpq_poly_gcd` (*C function*), 348
`_fmpq_poly_gegenbauer_c` (*C function*), 354
`_fmpq_poly_integral` (*C function*), 350
`_fmpq_poly_interpolate_barycentric` (*C function*), 355
`_fmpq_poly_interpolate_fast` (*C function*), 355
`_fmpq_poly_interpolate_fast_precomp` (*C function*), 355
`_fmpq_poly_interpolate_fmpq_vec` (*C function*), 355
`_fmpq_poly_interpolate_multi_mod` (*C function*), 355
`_fmpq_poly_interpolation_weights` (*C function*), 355
`_fmpq_poly_inv_series` (*C function*), 348
`_fmpq_poly_inv_series_newton` (*C function*), 348

`_fmpz_poly_invsqrt_series` (*C function*), 350
`_fmpz_poly_is_canonical` (*C function*), 338
`_fmpz_poly_is_monic` (*C function*), 358
`_fmpz_poly_laguerre_l` (*C function*), 354
`_fmpz_poly_lcm` (*C function*), 349
`_fmpz_poly_legendre_p` (*C function*), 354
`_fmpz_poly_log_series` (*C function*), 351
`_fmpz_poly_make_monic` (*C function*), 358
`_fmpz_poly_mul` (*C function*), 345
`_fmpz_poly_mullow` (*C function*), 345
`_fmpz_poly_normalise` (*C function*), 337
`_fmpz_poly_nth_derivative` (*C function*), 350
`_fmpz_poly_pow` (*C function*), 345
`_fmpz_poly_pow_trunc` (*C function*), 345
`_fmpz_poly_power_sums` (*C function*), 351
`_fmpz_poly_power_sums_to_poly` (*C function*), 351
`_fmpz_poly_powers_clear` (*C function*), 347
`_fmpz_poly_powers_precompute` (*C function*), 347
`_fmpz_poly_primitive_part` (*C function*), 358
`_fmpz_poly_print` (*C function*), 359
`_fmpz_poly_print_pretty` (*C function*), 359
`_fmpz_poly_rem` (*C function*), 346
`_fmpz_poly_rem_powers_precomp` (*C function*), 347
`_fmpz_poly_rescale` (*C function*), 355
`_fmpz_poly_resultant` (*C function*), 349
`_fmpz_poly_revert_series` (*C function*), 358
`_fmpz_poly_revert_series_lagrange` (*C function*), 357
`_fmpz_poly_revert_series_lagrange_fast` (*C function*), 357
`_fmpz_poly_revert_series_newton` (*C function*), 357
`_fmpz_poly_scalar_div_fmpz` (*C function*), 344
`_fmpz_poly_scalar_div_fmpz` (*C function*), 344
`_fmpz_poly_scalar_div_si` (*C function*), 344
`_fmpz_poly_scalar_div_ui` (*C function*), 344
`_fmpz_poly_scalar_mul_fmpz` (*C function*), 344
`_fmpz_poly_scalar_mul_fmpz` (*C function*), 343
`_fmpz_poly_scalar_mul_si` (*C function*), 343
`_fmpz_poly_scalar_mul_ui` (*C function*), 343
`_fmpz_poly_set_length` (*C function*), 337
`_fmpz_poly_set_str` (*C function*), 339
`_fmpz_poly_sin_cos_series` (*C function*), 353
`_fmpz_poly_sin_series` (*C function*), 352
`_fmpz_poly_sinh_cosh_series` (*C function*), 353
`_fmpz_poly_sinh_series` (*C function*), 353
`_fmpz_poly_sqrt_series` (*C function*), 350
`_fmpz_poly_sub` (*C function*), 342
`_fmpz_poly_sub_can` (*C function*), 342
`_fmpz_poly_sub_series` (*C function*), 343
`_fmpz_poly_sub_series_can` (*C function*), 343
`_fmpz_poly_tan_series` (*C function*), 352
`_fmpz_poly_tanh_series` (*C function*), 353
`_fmpz_poly_xgcd` (*C function*), 348
`_fmpz_pow_si` (*C function*), 323
`_fmpz_print` (*C function*), 321
`_fmpz_randbits` (*C function*), 321
`_fmpz_randtest` (*C function*), 321
`_fmpz_reconstruct_fmpz` (*C function*), 324
`_fmpz_reconstruct_fmpz_2` (*C function*), 324
`_fmpz_reconstruct_fmpz_2_naive` (*C function*), 324
`_fmpz_set_si` (*C function*), 319
`_fmpz_set_ui` (*C function*), 319
`_fmpz_simplest_between` (*C function*), 325
`_fmpz_sub` (*C function*), 322
`_fmpz_sub_fmpz` (*C function*), 322
`_fmpz_sub_si` (*C function*), 322
`_fmpz_sub_ui` (*C function*), 322
`_fmpz_submul` (*C function*), 323
`_fmpz_vec_clear` (*C function*), 327
`_fmpz_vec_dot` (*C function*), 328
`_fmpz_vec_equal` (*C function*), 327
`_fmpz_vec_fprint` (*C function*), 328
`_fmpz_vec_get_fmpz_vec_fmpz` (*C function*), 328
`_fmpz_vec_init` (*C function*), 327
`_fmpz_vec_max_height` (*C function*), 327
`_fmpz_vec_max_height_bits` (*C function*), 327
`_fmpz_vec_print` (*C function*), 328
`_fmpz_vec_randtest` (*C function*), 327
`_fmpz_vec_randtest_uniq_sorted` (*C function*), 327
`_fmpz_vec_set_fmpz_vec` (*C function*), 328
`_fmpz_vec_sort` (*C function*), 327
`_fmpz_cleanup` (*C function*), 156
`_fmpz_cleanup_mpz_content` (*C function*), 156
`_fmpz_clear_mpz` (*C function*), 156
`_fmpz_demote` (*C function*), 157
`_fmpz_demote_val` (*C function*), 157
`_fmpz_factor_append` (*C function*), 180
`_fmpz_factor_append_ui` (*C function*), 180
`_fmpz_is_canonical` (*C function*), 157
`_fmpz_mat_charpoly` (*C function*), 194
`_fmpz_mat_charpoly_berkowitz` (*C function*), 194
`_fmpz_mat_charpoly_modular` (*C function*), 194
`_fmpz_mat_minpoly` (*C function*), 194
`_fmpz_mat_minpoly_modular` (*C function*), 194
`_fmpz_mat_mul_double_word` (*C function*), 192
`_fmpz_mat_mul_multi_mod` (*C function*), 191
`_fmpz_mat_mul_small` (*C function*), 192
`_fmpz_mat_solve_dixon_den` (*C function*), 196
`_fmpz_mod_mat_mul_classical_threaded_op` (*C function*), 466
`_fmpz_mod_mat_mul_classical_threaded_pool_op` (*C function*), 466
`_fmpz_mod_mat_reduce` (*C function*), 464
`_fmpz_mod_mat_set_mod` (*C function*), 463
`_fmpz_mod_poly_add` (*C function*), 474
`_fmpz_mod_poly_compose` (*C function*), 490
`_fmpz_mod_poly_compose_mod` (*C function*), 491
`_fmpz_mod_poly_compose_mod_brent_kung` (*C function*), 491

```

_fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv499
  (C function), 492
_fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv_worker_minpoly_bm (C function), 487
  (C function), 492
_fmpz_mod_poly_compose_mod_brent_kung_preinv_fmpz_mod_poly_mul (C function), 476
  (C function), 492
_fmpz_mod_poly_compose_mod_brent_kung_vec_pre_fmpz_mod_poly_mulmid (C function), 476
  (C function), 492
_fmpz_mod_poly_compose_mod_brent_kung_vec_pre_fmpz_mod_poly_mulmod (C function), 476
  (C function), 493
_fmpz_mod_poly_compose_mod_horner (C function), 491
_fmpz_mod_poly_derivative (C function), 488
_fmpz_mod_poly_discriminant (C function), 488
_fmpz_mod_poly_div (C function), 481
_fmpz_mod_poly_div_newton_n_preinv (C function), 481
_fmpz_mod_poly_div_series (C function), 483
_fmpz_mod_poly_divides (C function), 483
_fmpz_mod_poly_divides_classical (C function), 483
_fmpz_mod_poly_divrem (C function), 482
_fmpz_mod_poly_divrem_basecase (C function), 480
_fmpz_mod_poly_divrem_newton_n_preinv (C function), 480
_fmpz_mod_poly_evaluate_fmpz (C function), 489
_fmpz_mod_poly_evaluate_fmpz_vec (C function), 489
_fmpz_mod_poly_evaluate_fmpz_vec_fast (C function), 489
_fmpz_mod_poly_evaluate_fmpz_vec_fast_precomp_fmpz_mod_poly_precompute_matrix_worker (C function), 489
_fmpz_mod_poly_evaluate_fmpz_vec_iter (C function), 489
_fmpz_mod_poly_fprint (C function), 495
_fmpz_mod_poly_gcd (C function), 484
_fmpz_mod_poly_gcd_euclidean_f (C function), 484
_fmpz_mod_poly_gcd_f (C function), 484
_fmpz_mod_poly_gcdinv (C function), 486
_fmpz_mod_poly_gcdinv_euclidean (C function), 485
_fmpz_mod_poly_gcdinv_euclidean_f (C function), 486
_fmpz_mod_poly_gcdinv_f (C function), 486
_fmpz_mod_poly_hgcd (C function), 484
_fmpz_mod_poly_interval_poly_worker (C function), 500
_fmpz_mod_poly_inv_series (C function), 483
_fmpz_mod_poly_invmod (C function), 486
_fmpz_mod_poly_invmod_f (C function), 486
_fmpz_mod_poly_invsqrt_series (C function), 490
_fmpz_mod_poly_is_squarefree (C function), 498
_fmpz_mod_poly_is_squarefree_f (C function),
_fmpz_mod_poly_minpoly (C function), 487
_fmpz_mod_poly_minpoly_hgcd (C function), 487
_fmpz_mod_poly_mul (C function), 476
_fmpz_mod_poly_mulmod (C function), 476
_fmpz_mod_poly_mulmid (C function), 476
_fmpz_mod_poly_neg (C function), 475
_fmpz_mod_poly_normalise (C function), 471
_fmpz_mod_poly_pow (C function), 477
_fmpz_mod_poly_pow_trunc (C function), 477
_fmpz_mod_poly_pow_trunc_binexp (C function), 478
_fmpz_mod_poly_powers_mod_preinv_naive (C function), 479
_fmpz_mod_poly_powers_mod_preinv_threaded_pool (C function), 479
_fmpz_mod_poly_powmod_fmpz_binexp (C function), 478
_fmpz_mod_poly_powmod_fmpz_binexp_preinv (C function), 478
_fmpz_mod_poly_powmod_ui_binexp (C function), 478
_fmpz_mod_poly_powmod_ui_binexp_preinv (C function), 478
_fmpz_mod_poly_powmod_x_fmpz_preinv (C function), 479
_fmpz_mod_poly_precompute_matrix (C function), 491
_fmpz_mod_poly_precompute_matrix_worker (C function), 491
_fmpz_mod_poly_product_roots_fmpz_vec (C function), 477
_fmpz_mod_poly_radix (C function), 495
_fmpz_mod_poly_radix_init (C function), 494
_fmpz_mod_poly_reduce (C function), 1082
_fmpz_mod_poly_reduce_matrix_mod_poly (C function), 491
_fmpz_mod_poly_rem (C function), 482
_fmpz_mod_poly_rem_basecase (C function), 481
_fmpz_mod_poly_resultant (C function), 488
_fmpz_mod_poly_scalar_div_fmpz (C function), 475
_fmpz_mod_poly_scalar_mul_fmpz (C function), 475
_fmpz_mod_poly_scalar_mul_ui (C function), 475
_fmpz_mod_poly_set_length (C function), 471
_fmpz_mod_poly_shift_left (C function), 474
_fmpz_mod_poly_shift_right (C function), 474
_fmpz_mod_poly_sqr (C function), 476
_fmpz_mod_poly_sqrt (C function), 490
_fmpz_mod_poly_sqrt_series (C function), 490
_fmpz_mod_poly_sub (C function), 475
_fmpz_mod_poly_tree_alloc (C function), 494

```

_fmpz_mod_poly_tree_build (*C function*), 494
 _fmpz_mod_poly_tree_free (*C function*), 494
 _fmpz_mod_poly_xgcd (*C function*), 485
 _fmpz_mod_poly_xgcd_euclidean_f (*C function*), 485
 _fmpz_mod_vec_add (*C function*), 462
 _fmpz_mod_vec_dot (*C function*), 462
 _fmpz_mod_vec_dot_rev (*C function*), 462
 _fmpz_mod_vec_mul (*C function*), 462
 _fmpz_mod_vec_neg (*C function*), 462
 _fmpz_mod_vec_scalar_addmul_fmpz_mod (*C function*), 462
 _fmpz_mod_vec_scalar_div_fmpz_mod (*C function*), 462
 _fmpz_mod_vec_scalar_mul_fmpz_mod (*C function*), 462
 _fmpz_mod_vec_set_fmpz_vec (*C function*), 462
 _fmpz_mod_vec_sub (*C function*), 462
 _fmpz_mpoly_div_monagan_pearce (*C function*), 272
 _fmpz_mpoly_divides_array (*C function*), 271
 _fmpz_mpoly_divides_monagan_pearce (*C function*), 272
 _fmpz_mpoly_divrem_array (*C function*), 273
 _fmpz_mpoly_divrem_ideal_monagan_pearce (*C function*), 273
 _fmpz_mpoly_divrem_monagan_pearce (*C function*), 272
 _fmpz_mpoly_fits_small (*C function*), 260
 _fmpz_mpoly_q_content (*C function*), 380
 _fmpz_new_mpz (*C function*), 156
 _fmpz_nm1_trial_factors (*C function*), 173
 _fmpz_np1_trial_factors (*C function*), 173
 _fmpz_poly_2norm (*C function*), 221
 _fmpz_poly_2norm_normalised_bits (*C function*), 221
 _fmpz_poly_CRT_ui (*C function*), 245
 _fmpz_poly_CRT_ui_precomp (*C function*), 245
 _fmpz_poly_add (*C function*), 212
 _fmpz_poly_bit_pack (*C function*), 214
 _fmpz_poly_bit_unpack (*C function*), 214
 _fmpz_poly_bit_unpack_unsigned (*C function*), 214
 _fmpz_poly_bound_roots (*C function*), 247
 _fmpz_poly_chebyshev_t (*C function*), 248
 _fmpz_poly_chebyshev_u (*C function*), 248
 _fmpz_poly_compose (*C function*), 237
 _fmpz_poly_compose_divconquer (*C function*), 237
 _fmpz_poly_compose_horner (*C function*), 237
 _fmpz_poly_compose_series (*C function*), 239
 _fmpz_poly_compose_series_brent_kung (*C function*), 238
 _fmpz_poly_compose_series_horner (*C function*), 238
 _fmpz_poly_content (*C function*), 224
 _fmpz_poly_cos_minpoly (*C function*), 248
 _fmpz_poly_cyclotomic (*C function*), 247
 _fmpz_poly_derivative (*C function*), 234
 _fmpz_poly_discriminant (*C function*), 224
 _fmpz_poly_div (*C function*), 228
 _fmpz_poly_div_basecase (*C function*), 226
 _fmpz_poly_div_divconquer (*C function*), 227
 _fmpz_poly_div_divconquer_recursive (*C function*), 227
 _fmpz_poly_div_preinv (*C function*), 229
 _fmpz_poly_div_root_fmpz (*C function*), 229
 _fmpz_poly_div_series (*C function*), 231
 _fmpz_poly_div_series_basecase (*C function*), 231
 _fmpz_poly_div_series_divconquer (*C function*), 231
 _fmpz_poly_divexact (*C function*), 229
 _fmpz_poly_divexact_root_fmpz (*C function*), 229
 _fmpz_poly_divides (*C function*), 230
 _fmpz_poly_divrem (*C function*), 226
 _fmpz_poly_divrem_basecase (*C function*), 225
 _fmpz_poly_divrem_divconquer (*C function*), 226
 _fmpz_poly_divrem_divconquer_recursive (*C function*), 225
 _fmpz_poly_divrem_preinv (*C function*), 229
 _fmpz_poly_divremlow_divconquer_recursive (*C function*), 227
 _fmpz_poly_eta_qexp (*C function*), 250
 _fmpz_poly_evaluate_divconquer_fmpz (*C function*), 234
 _fmpz_poly_evaluate_divconquer_fmpz (*C function*), 234
 _fmpz_poly_evaluate_fmpz (*C function*), 235
 _fmpz_poly_evaluate_fmpz (*C function*), 234
 _fmpz_poly_evaluate_horner_d (*C function*), 235
 _fmpz_poly_evaluate_horner_d_2exp (*C function*), 235
 _fmpz_poly_evaluate_horner_d_2exp2 (*C function*), 235
 _fmpz_poly_evaluate_horner_fmpz (*C function*), 234
 _fmpz_poly_evaluate_horner_fmpz (*C function*), 234
 _fmpz_poly_evaluate_mod (*C function*), 235
 _fmpz_poly_factor_cubic (*C function*), 258
 _fmpz_poly_factor_quadratic (*C function*), 258
 _fmpz_poly_factor_zassenhaus (*C function*), 258
 _fmpz_poly_fibonacci (*C function*), 249
 _fmpz_poly_fprint (*C function*), 244
 _fmpz_poly_fprint_pretty (*C function*), 244
 _fmpz_poly_gcd (*C function*), 222
 _fmpz_poly_gcd_heuristic (*C function*), 221
 _fmpz_poly_gcd_modular (*C function*), 222
 _fmpz_poly_gcd_subresultant (*C function*), 221
 _fmpz_poly_get_str (*C function*), 209
 _fmpz_poly_get_str_pretty (*C function*), 209

`_fmpz_poly_hensel_continue_lift` (*C function*), 243

`_fmpz_poly_hensel_start_lift` (*C function*), 243

`_fmpz_poly_hermite_h` (*C function*), 249

`_fmpz_poly_hermite_he` (*C function*), 249

`_fmpz_poly_interpolate` (*C function*), 236

`_fmpz_poly_interpolate_exact` (*C function*), 236

`_fmpz_poly_interpolate_exact_newton` (*C function*), 236

`_fmpz_poly_interpolate_multi_mod` (*C function*), 236

`_fmpz_poly_interpolate_newton` (*C function*), 236

`_fmpz_poly_inv_series` (*C function*), 231

`_fmpz_poly_inv_series_basecase` (*C function*), 231

`_fmpz_poly_inv_series_newton` (*C function*), 231

`_fmpz_poly_is_cyclotomic` (*C function*), 248

`_fmpz_poly_is_squarefree` (*C function*), 225

`_fmpz_poly_lcm` (*C function*), 223

`_fmpz_poly_legendre_pt` (*C function*), 248

`_fmpz_poly_monomial_to_newton` (*C function*), 236

`_fmpz_poly_mul` (*C function*), 216

`_fmpz_poly_mul_KS` (*C function*), 216

`_fmpz_poly_mul_SS` (*C function*), 216

`_fmpz_poly_mul_classical` (*C function*), 214

`_fmpz_poly_mul_karatsuba` (*C function*), 215

`_fmpz_poly_mul_mid_default_mpn_ctx` (*C function*), 306

`_fmpz_poly_mul_mid_mpn_ctx` (*C function*), 306

`_fmpz_poly_mulhigh` (*C function*), 217

`_fmpz_poly_mulhigh_classical` (*C function*), 215

`_fmpz_poly_mulhigh_karatsuba_n` (*C function*), 215

`_fmpz_poly_mulloy` (*C function*), 216

`_fmpz_poly_mulloy_KS` (*C function*), 216

`_fmpz_poly_mulloy_SS` (*C function*), 216

`_fmpz_poly_mulloy_SS_precache` (*C function*), 217

`_fmpz_poly_mulloy_classical` (*C function*), 214

`_fmpz_poly_mulloy_karatsuba_n` (*C function*), 215

`_fmpz_poly_mulmid` (*C function*), 215

`_fmpz_poly_mulmid_KS` (*C function*), 215

`_fmpz_poly_mulmid_SS` (*C function*), 215

`_fmpz_poly_mulmid_classical` (*C function*), 215

`_fmpz_poly_newton_to_monomial` (*C function*), 236

`_fmpz_poly_normalise` (*C function*), 208

`_fmpz_poly_nth_derivative` (*C function*), 234

`_fmpz_poly_num_real_roots` (*C function*), 247

`_fmpz_poly_num_real_roots_sturm` (*C function*), 247

`_fmpz_poly_pow` (*C function*), 220

`_fmpz_poly_pow_addchains` (*C function*), 219

`_fmpz_poly_pow_binexp` (*C function*), 220

`_fmpz_poly_pow_binomial` (*C function*), 219

`_fmpz_poly_pow_multinomial` (*C function*), 219

`_fmpz_poly_pow_small` (*C function*), 220

`_fmpz_poly_pow_trunc` (*C function*), 220

`_fmpz_poly_power_sums_naive` (*C function*), 241

`_fmpz_poly_power_sums_to_poly` (*C function*), 241

`_fmpz_poly_powers_clear` (*C function*), 230

`_fmpz_poly_powers_precompute` (*C function*), 230

`_fmpz_poly_preinvert` (*C function*), 229

`_fmpz_poly_primitive_part` (*C function*), 224

`_fmpz_poly_print` (*C function*), 244

`_fmpz_poly_print_pretty` (*C function*), 244

`_fmpz_poly_product_roots_fmpz_vec` (*C function*), 246

`_fmpz_poly_product_roots_fmpz_vec` (*C function*), 246

`_fmpz_poly_pseudo_div` (*C function*), 233

`_fmpz_poly_pseudo_divrem` (*C function*), 233

`_fmpz_poly_pseudo_divrem_basecase` (*C function*), 232

`_fmpz_poly_pseudo_divrem_cohen` (*C function*), 232

`_fmpz_poly_pseudo_divrem_divconquer` (*C function*), 232

`_fmpz_poly_pseudo_rem` (*C function*), 233

`_fmpz_poly_pseudo_rem_cohen` (*C function*), 233

`_fmpz_poly_reduce` (*C function*), 1082

`_fmpz_poly_rem` (*C function*), 228

`_fmpz_poly_rem_basecase` (*C function*), 228

`_fmpz_poly_rem_powers_precomp` (*C function*), 230

`_fmpz_poly_remove_content_2exp` (*C function*), 214

`_fmpz_poly_resultant` (*C function*), 224

`_fmpz_poly_resultant_euclidean` (*C function*), 224

`_fmpz_poly_resultant_modular` (*C function*), 223

`_fmpz_poly_reverse` (*C function*), 210

`_fmpz_poly_revert_series` (*C function*), 239

`_fmpz_poly_scale_2exp` (*C function*), 214

`_fmpz_poly_set_length` (*C function*), 208

`_fmpz_poly_set_str` (*C function*), 209

`_fmpz_poly_shift_left` (*C function*), 220

`_fmpz_poly_shift_right` (*C function*), 220

`_fmpz_poly_signature` (*C function*), 241

`_fmpz_poly_sqr` (*C function*), 218

`_fmpz_poly_sqr_KS` (*C function*), 218

`_fmpz_poly_sqr_classical` (*C function*), 218

`_fmpz_poly_sqr_karatsuba` (*C function*), 218

`_fmpz_poly_sqr_low` (*C function*), 219

`_fmpz_poly_sqr_low_KS` (*C function*), 218

`_fmpz_poly_sqr_low_classical` (*C function*), 219

_fmpz_poly_sqr_low_karatsuba_n (*C function*), 218
 _fmpz_poly_sqrt (*C function*), 240
 _fmpz_poly_sqrt_KS (*C function*), 240
 _fmpz_poly_sqrt_classical (*C function*), 239
 _fmpz_poly_sqrt_divconquer (*C function*), 240
 _fmpz_poly_sqrt_series (*C function*), 240
 _fmpz_poly_sqr_trem_classical (*C function*), 239
 _fmpz_poly_sqr_trem_divconquer (*C function*), 239
 _fmpz_poly_sub (*C function*), 212
 _fmpz_poly_swinnerton_dyer (*C function*), 248
 _fmpz_poly_taylor_shift (*C function*), 238
 _fmpz_poly_taylor_shift_divconquer (*C function*), 238
 _fmpz_poly_taylor_shift_horner (*C function*), 238
 _fmpz_poly_taylor_shift_multi_mod (*C function*), 238
 _fmpz_poly_theta_qexp (*C function*), 250
 _fmpz_poly_tree_alloc (*C function*), 246
 _fmpz_poly_tree_build_fmpz_vec (*C function*), 246
 _fmpz_poly_tree_free (*C function*), 246
 _fmpz_poly_xgcd (*C function*), 223
 _fmpz_poly_xgcd_modular (*C function*), 222
 _fmpz_promote (*C function*), 156
 _fmpz_promote_val (*C function*), 157
 _fmpz_remove (*C function*), 168
 _fmpz_rfacs (*C function*), 167
 _fmpz_set_si_small (*C function*), 818
 _fmpz_size (*C function*), 818
 _fmpz_sub_small (*C function*), 818
 _fmpz_ui_vec_prod (*C function*), 179
 _fmpz_vec_add (*C function*), 178
 _fmpz_vec_clear (*C function*), 175
 _fmpz_vec_content (*C function*), 180
 _fmpz_vec_content_chained (*C function*), 180
 _fmpz_vec_dot (*C function*), 180
 _fmpz_vec_dot_general (*C function*), 180
 _fmpz_vec_dot_general_naive (*C function*), 180
 _fmpz_vec_equal (*C function*), 177
 _fmpz_vec_fprint (*C function*), 176
 _fmpz_vec_fread (*C function*), 176
 _fmpz_vec_get_d_vec_2exp (*C function*), 177
 _fmpz_vec_get_fft (*C function*), 177
 _fmpz_vec_get_nmod_vec (*C function*), 177
 _fmpz_vec_height (*C function*), 176
 _fmpz_vec_height_index (*C function*), 176
 _fmpz_vec_init (*C function*), 175
 _fmpz_vec_is_zero (*C function*), 177
 _fmpz_vec_lcm (*C function*), 180
 _fmpz_vec_max (*C function*), 177
 _fmpz_vec_max_bits (*C function*), 175
 _fmpz_vec_max_bits_ref (*C function*), 175
 _fmpz_vec_max_inplace (*C function*), 177
 _fmpz_vec_max_limbs (*C function*), 176
 _fmpz_vec_neg (*C function*), 177
 _fmpz_vec_print (*C function*), 176
 _fmpz_vec_prod (*C function*), 179
 _fmpz_vec_randtest (*C function*), 175
 _fmpz_vec_randtest_unsigned (*C function*), 175
 _fmpz_vec_read (*C function*), 176
 _fmpz_vec_scalar_abs (*C function*), 177
 _fmpz_vec_scalar_addmul_fmpz (*C function*), 179
 _fmpz_vec_scalar_addmul_si (*C function*), 179
 _fmpz_vec_scalar_addmul_si_2exp (*C function*), 179
 _fmpz_vec_scalar_addmul_ui (*C function*), 179
 _fmpz_vec_scalar_divexact_fmpz (*C function*), 178
 _fmpz_vec_scalar_divexact_si (*C function*), 178
 _fmpz_vec_scalar_divexact_ui (*C function*), 178
 _fmpz_vec_scalar_fdiv_q_2exp (*C function*), 178
 _fmpz_vec_scalar_fdiv_q_fmpz (*C function*), 178
 _fmpz_vec_scalar_fdiv_q_si (*C function*), 178
 _fmpz_vec_scalar_fdiv_q_ui (*C function*), 178
 _fmpz_vec_scalar_fdiv_r_2exp (*C function*), 178
 _fmpz_vec_scalar_mod_fmpz (*C function*), 179
 _fmpz_vec_scalar_mul_2exp (*C function*), 178
 _fmpz_vec_scalar_mul_fmpz (*C function*), 178
 _fmpz_vec_scalar_mul_si (*C function*), 178
 _fmpz_vec_scalar_mul_ui (*C function*), 178
 _fmpz_vec_scalar_smod_fmpz (*C function*), 179
 _fmpz_vec_scalar_submul_fmpz (*C function*), 179
 _fmpz_vec_scalar_submul_si (*C function*), 179
 _fmpz_vec_scalar_submul_si_2exp (*C function*), 179
 _fmpz_vec_scalar_tdiv_q_2exp (*C function*), 179
 _fmpz_vec_scalar_tdiv_q_fmpz (*C function*), 178
 _fmpz_vec_scalar_tdiv_q_si (*C function*), 179
 _fmpz_vec_scalar_tdiv_q_ui (*C function*), 179
 _fmpz_vec_set (*C function*), 177
 _fmpz_vec_set_fft (*C function*), 177
 _fmpz_vec_set_nmod_vec (*C function*), 177
 _fmpz_vec_sort (*C function*), 178
 _fmpz_vec_sub (*C function*), 178
 _fmpz_vec_sum (*C function*), 179
 _fmpz_vec_sum_max_bits (*C function*), 175
 _fmpz_vec_swap (*C function*), 177
 _fmpz_vec_zero (*C function*), 177
 _fq_ctx_init_conway (*C function*), 915
 _fq_default_poly_set_length (*C function*), 959
 _fq_dense_reduce (*C function*), 916
 _fq_embed_gens_naive (*C function*), 971
 _fq_frobenius (*C function*), 920

`_fq_inv` (*C function*), 917
`_fq_nmod_ctx_init_conway_ui` (*C function*), 973
`_fq_nmod_dense_reduce` (*C function*), 974
`_fq_nmod_embed_gens_naive` (*C function*), 1011
`_fq_nmod_frobenius` (*C function*), 978
`_fq_nmod_inv` (*C function*), 975
`_fq_nmod_norm` (*C function*), 978
`_fq_nmod_poly_add` (*C function*), 991
`_fq_nmod_poly_compose` (*C function*), 1003
`_fq_nmod_poly_compose_mod` (*C function*), 1005
`_fq_nmod_poly_compose_mod_brent_kung` (*C function*), 1004
`_fq_nmod_poly_compose_mod_brent_kung_precomp` (*C function*), 1005
`_fq_nmod_poly_compose_mod_brent_kung_preinv` (*C function*), 1004
`_fq_nmod_poly_compose_mod_horner` (*C function*), 1003
`_fq_nmod_poly_compose_mod_horner_preinv` (*C function*), 1004
`_fq_nmod_poly_compose_mod_preinv` (*C function*), 1005
`_fq_nmod_poly_derivative` (*C function*), 1002
`_fq_nmod_poly_div` (*C function*), 999
`_fq_nmod_poly_div_newton_n_preinv` (*C function*), 999
`_fq_nmod_poly_div_series` (*C function*), 1000
`_fq_nmod_poly_divides` (*C function*), 1002
`_fq_nmod_poly_divrem` (*C function*), 998
`_fq_nmod_poly_divrem_newton_n_preinv` (*C function*), 1000
`_fq_nmod_poly_evaluate_fq_nmod` (*C function*), 1003
`_fq_nmod_poly_fprint` (*C function*), 1006
`_fq_nmod_poly_fprint_pretty` (*C function*), 1006
`_fq_nmod_poly_gcd` (*C function*), 1001
`_fq_nmod_poly_gcd_euclidean_f` (*C function*), 1001
`_fq_nmod_poly_get_str` (*C function*), 1007
`_fq_nmod_poly_get_str_pretty` (*C function*), 1007
`_fq_nmod_poly_hamming_weight` (*C function*), 998
`_fq_nmod_poly_inv_series` (*C function*), 1000
`_fq_nmod_poly_inv_series_newton` (*C function*), 1000
`_fq_nmod_poly_invsqrt_series` (*C function*), 1003
`_fq_nmod_poly_is_squarefree` (*C function*), 1009
`_fq_nmod_poly_make_monic` (*C function*), 989
`_fq_nmod_poly_mul` (*C function*), 993
`_fq_nmod_poly_mul_KS` (*C function*), 993
`_fq_nmod_poly_mul_classical` (*C function*), 992
`_fq_nmod_poly_mul_reorder` (*C function*), 992
`_fq_nmod_poly_mul_univariate` (*C function*), 992
`_fq_nmod_poly_mulhigh` (*C function*), 994
`_fq_nmod_poly_mulhigh_classical` (*C function*), 994
`_fq_nmod_poly_mulmod` (*C function*), 994
`_fq_nmod_poly_mulmod_KS` (*C function*), 993
`_fq_nmod_poly_mulmod_classical` (*C function*), 993
`_fq_nmod_poly_mulmod_univariate` (*C function*), 993
`_fq_nmod_poly_mulmod_preinv` (*C function*), 995
`_fq_nmod_poly_neg` (*C function*), 991
`_fq_nmod_poly_normalise` (*C function*), 988
`_fq_nmod_poly_normalise2` (*C function*), 988
`_fq_nmod_poly_pow` (*C function*), 996
`_fq_nmod_poly_pow_trunc` (*C function*), 997
`_fq_nmod_poly_pow_trunc_binexp` (*C function*), 997
`_fq_nmod_poly_powmod_fmpz_binexp` (*C function*), 996
`_fq_nmod_poly_powmod_fmpz_binexp_preinv` (*C function*), 996
`_fq_nmod_poly_powmod_fmpz_sliding_preinv` (*C function*), 997
`_fq_nmod_poly_powmod_ui_binexp` (*C function*), 996
`_fq_nmod_poly_powmod_ui_binexp_preinv` (*C function*), 996
`_fq_nmod_poly_powmod_x_fmpz_preinv` (*C function*), 997
`_fq_nmod_poly_precompute_matrix` (*C function*), 1005
`_fq_nmod_poly_print` (*C function*), 1006
`_fq_nmod_poly_print_pretty` (*C function*), 1006
`_fq_nmod_poly_reduce_matrix_mod_poly` (*C function*), 1005
`_fq_nmod_poly_rem` (*C function*), 999
`_fq_nmod_poly_reverse` (*C function*), 988
`_fq_nmod_poly_scalar_addmul_fq_nmod` (*C function*), 991
`_fq_nmod_poly_scalar_div_fq` (*C function*), 992
`_fq_nmod_poly_scalar_mul_fq_nmod` (*C function*), 991
`_fq_nmod_poly_scalar_submul_fq_nmod` (*C function*), 991
`_fq_nmod_poly_set` (*C function*), 989
`_fq_nmod_poly_set_length` (*C function*), 988
`_fq_nmod_poly_shift_left` (*C function*), 998
`_fq_nmod_poly_shift_right` (*C function*), 998
`_fq_nmod_poly_sqr` (*C function*), 995
`_fq_nmod_poly_sqr_KS` (*C function*), 995
`_fq_nmod_poly_sqr_classical` (*C function*), 995
`_fq_nmod_poly_sqrt` (*C function*), 1003
`_fq_nmod_poly_sqrt_series` (*C function*), 1003
`_fq_nmod_poly_sub` (*C function*), 991
`_fq_nmod_poly_xgcd` (*C function*), 1001
`_fq_nmod_poly_xgcd_euclidean_f` (*C function*), 1001

_fq_nmod_poly_zero (*C function*), 989
 _fq_nmod_pow (*C function*), 975
 _fq_nmod_reduce (*C function*), 974
 _fq_nmod_sparse_reduce (*C function*), 974
 _fq_nmod_trace (*C function*), 978
 _fq_nmod_vec_add (*C function*), 980
 _fq_nmod_vec_clear (*C function*), 979
 _fq_nmod_vec_dot (*C function*), 981
 _fq_nmod_vec_equal (*C function*), 980
 _fq_nmod_vec_fprint (*C function*), 979
 _fq_nmod_vec_init (*C function*), 979
 _fq_nmod_vec_is_zero (*C function*), 980
 _fq_nmod_vec_neg (*C function*), 980
 _fq_nmod_vec_print (*C function*), 979
 _fq_nmod_vec_randtest (*C function*), 979
 _fq_nmod_vec_scalar_addmul_fq_nmod (*C function*), 980
 _fq_nmod_vec_scalar_submul_fq_nmod (*C function*), 980
 _fq_nmod_vec_set (*C function*), 980
 _fq_nmod_vec_sub (*C function*), 980
 _fq_nmod_vec_swap (*C function*), 980
 _fq_nmod_vec_zero (*C function*), 980
 _fq_norm (*C function*), 920
 _fq_poly_add (*C function*), 943
 _fq_poly_compose (*C function*), 955
 _fq_poly_compose_mod (*C function*), 956
 _fq_poly_compose_mod_brent_kung (*C function*), 955
 _fq_poly_compose_mod_brent_kung_precomp_preinv (*C function*), 957
 _fq_poly_compose_mod_brent_kung_preinv (*C function*), 956
 _fq_poly_compose_mod_horner (*C function*), 955
 _fq_poly_compose_mod_horner_preinv (*C function*), 955
 _fq_poly_compose_mod_preinv (*C function*), 956
 _fq_poly_derivative (*C function*), 954
 _fq_poly_div (*C function*), 951
 _fq_poly_div_newton_n_preinv (*C function*), 951
 _fq_poly_div_series (*C function*), 952
 _fq_poly_divides (*C function*), 954
 _fq_poly_divrem (*C function*), 950
 _fq_poly_divrem_newton_n_preinv (*C function*), 951
 _fq_poly_evaluate_fq (*C function*), 955
 _fq_poly_fprint (*C function*), 957
 _fq_poly_fprint_pretty (*C function*), 957
 _fq_poly_gcd (*C function*), 952
 _fq_poly_gcd_euclidean_f (*C function*), 952
 _fq_poly_get_str (*C function*), 958
 _fq_poly_get_str_pretty (*C function*), 958
 _fq_poly_hamming_weight (*C function*), 950
 _fq_poly_inv_series (*C function*), 952
 _fq_poly_inv_series_newton (*C function*), 952
 _fq_poly_invsqrt_series (*C function*), 954
 _fq_poly_is_squarefree (*C function*), 967
 _fq_poly_make_monic (*C function*), 942
 _fq_poly_mul (*C function*), 945
 _fq_poly_mul_KS (*C function*), 945
 _fq_poly_mul_classical (*C function*), 944
 _fq_poly_mul_reorder (*C function*), 944
 _fq_poly_mul_univariate (*C function*), 945
 _fq_poly_mulhigh (*C function*), 946
 _fq_poly_mulhigh_classical (*C function*), 946
 _fq_poly_mullo (*C function*), 946
 _fq_poly_mullo_KS (*C function*), 946
 _fq_poly_mullo_classical (*C function*), 945
 _fq_poly_mullo_univariate (*C function*), 946
 _fq_poly_mulmod (*C function*), 947
 _fq_poly_mulmod_preinv (*C function*), 947
 _fq_poly_neg (*C function*), 944
 _fq_poly_normalise (*C function*), 941
 _fq_poly_normalise2 (*C function*), 941
 _fq_poly_pow (*C function*), 948
 _fq_poly_pow_trunc (*C function*), 949
 _fq_poly_pow_trunc_binexp (*C function*), 949
 _fq_poly_powmod_fmpz_binexp (*C function*), 948
 _fq_poly_powmod_fmpz_binexp_preinv (*C function*), 948
 _fq_poly_powmod_fmpz_sliding_preinv (*C function*), 949
 _fq_poly_powmod_ui_binexp (*C function*), 948
 _fq_poly_powmod_ui_binexp_preinv (*C function*), 948
 _fq_poly_powmod_x_fmpz_preinv (*C function*), 949
 _fq_poly_precompute_matrix (*C function*), 956
 _fq_poly_print (*C function*), 957
 _fq_poly_print_pretty (*C function*), 957
 _fq_poly_reduce_matrix_mod_poly (*C function*), 956
 _fq_poly_rem (*C function*), 951
 _fq_poly_reverse (*C function*), 941
 _fq_poly_scalar_addmul_fq (*C function*), 944
 _fq_poly_scalar_div_fq (*C function*), 944
 _fq_poly_scalar_mul_fq (*C function*), 944
 _fq_poly_scalar_submul_fq (*C function*), 944
 _fq_poly_set (*C function*), 942
 _fq_poly_set_length (*C function*), 941
 _fq_poly_shift_left (*C function*), 950
 _fq_poly_shift_right (*C function*), 950
 _fq_poly_sqr (*C function*), 948
 _fq_poly_sqr_KS (*C function*), 947
 _fq_poly_sqr_classical (*C function*), 947
 _fq_poly_sqr_reorder (*C function*), 947
 _fq_poly_sqrt (*C function*), 954
 _fq_poly_sqrt_series (*C function*), 954
 _fq_poly_sub (*C function*), 943
 _fq_poly_xgcd (*C function*), 953
 _fq_poly_xgcd_euclidean_f (*C function*), 953
 _fq_poly_zero (*C function*), 942
 _fq_pow (*C function*), 917
 _fq_reduce (*C function*), 917
 _fq_sparse_reduce (*C function*), 916

`_fq_trace` (*C function*), 920
`_fq_vec_add` (*C function*), 928
`_fq_vec_clear` (*C function*), 927
`_fq_vec_dot` (*C function*), 928
`_fq_vec_equal` (*C function*), 927
`_fq_vec_fprint` (*C function*), 927
`_fq_vec_init` (*C function*), 927
`_fq_vec_is_zero` (*C function*), 927
`_fq_vec_neg` (*C function*), 927
`_fq_vec_print` (*C function*), 927
`_fq_vec_randtest` (*C function*), 927
`_fq_vec_scalar_addmul_fq` (*C function*), 928
`_fq_vec_scalar_submul_fq` (*C function*), 928
`_fq_vec_set` (*C function*), 927
`_fq_vec_sub` (*C function*), 928
`_fq_vec_swap` (*C function*), 927
`_fq_vec_zero` (*C function*), 927
`_fq_zech_ctx_init_conway_ui` (*C function*), 1024
`_fq_zech_dense_reduce` (*C function*), 1025
`_fq_zech_embed_gens_naive` (*C function*), 1060
`_fq_zech_inv` (*C function*), 1026
`_fq_zech_poly_add` (*C function*), 1041
`_fq_zech_poly_compose` (*C function*), 1053
`_fq_zech_poly_compose_mod` (*C function*), 1054
`_fq_zech_poly_compose_mod_brent_kung` (*C function*), 1053
`_fq_zech_poly_compose_mod_brent_kung_precomp` (*C function*), 1055
`_fq_zech_poly_compose_mod_brent_kung_preinv` (*C function*), 1054
`_fq_zech_poly_compose_mod_horner` (*C function*), 1053
`_fq_zech_poly_compose_mod_horner_preinv` (*C function*), 1053
`_fq_zech_poly_compose_mod_preinv` (*C function*), 1054
`_fq_zech_poly_derivative` (*C function*), 1052
`_fq_zech_poly_div` (*C function*), 1048
`_fq_zech_poly_div_newton_n_preinv` (*C function*), 1048
`_fq_zech_poly_div_series` (*C function*), 1050
`_fq_zech_poly_divides` (*C function*), 1051
`_fq_zech_poly_divrem` (*C function*), 1048
`_fq_zech_poly_divrem_newton_n_preinv` (*C function*), 1049
`_fq_zech_poly_evaluate_fq_zech` (*C function*), 1052
`_fq_zech_poly_fprint` (*C function*), 1055
`_fq_zech_poly_fprint_pretty` (*C function*), 1055
`_fq_zech_poly_gcd` (*C function*), 1050
`_fq_zech_poly_gcd_euclidean_f` (*C function*), 1050
`_fq_zech_poly_get_str` (*C function*), 1056
`_fq_zech_poly_get_str_pretty` (*C function*), 1056
`_fq_zech_poly_hamming_weight` (*C function*), 1048
`_fq_zech_poly_inv_series` (*C function*), 1049
`_fq_zech_poly_inv_series_newton` (*C function*), 1049
`_fq_zech_poly_invsqrt_series` (*C function*), 1052
`_fq_zech_poly_is_squarefree` (*C function*), 1058
`_fq_zech_poly_make_monic` (*C function*), 1039
`_fq_zech_poly_mul` (*C function*), 1043
`_fq_zech_poly_mul_KS` (*C function*), 1042
`_fq_zech_poly_mul_classical` (*C function*), 1042
`_fq_zech_poly_mul_reorder` (*C function*), 1042
`_fq_zech_poly_mulhigh` (*C function*), 1044
`_fq_zech_poly_mulhigh_classical` (*C function*), 1043
`_fq_zech_poly_mulmod` (*C function*), 1043
`_fq_zech_poly_mulmod_KS` (*C function*), 1043
`_fq_zech_poly_mulmod_classical` (*C function*), 1043
`_fq_zech_poly_mulmod_preinv` (*C function*), 1044
`_fq_zech_poly_neg` (*C function*), 1041
`_fq_zech_poly_normalise` (*C function*), 1038
`_fq_zech_poly_normalise2` (*C function*), 1038
`_fq_zech_poly_pow` (*C function*), 1045
`_fq_zech_poly_pow_trunc` (*C function*), 1047
`_fq_zech_poly_pow_trunc_binexp` (*C function*), 1047
`_fq_zech_poly_powmod_fmpz_binexp` (*C function*), 1046
`_fq_zech_poly_powmod_fmpz_binexp_preinv` (*C function*), 1046
`_fq_zech_poly_powmod_fmpz_sliding_preinv` (*C function*), 1046
`_fq_zech_poly_powmod_ui_binexp` (*C function*), 1045
`_fq_zech_poly_powmod_ui_binexp_preinv` (*C function*), 1045
`_fq_zech_poly_powmod_x_fmpz_preinv` (*C function*), 1046
`_fq_zech_poly_precompute_matrix` (*C function*), 1054
`_fq_zech_poly_print` (*C function*), 1056
`_fq_zech_poly_print_pretty` (*C function*), 1055
`_fq_zech_poly_reduce_matrix_mod_poly` (*C function*), 1054
`_fq_zech_poly_rem` (*C function*), 1048
`_fq_zech_poly_reverse` (*C function*), 1038
`_fq_zech_poly_scalar_addmul_fq_zech` (*C function*), 1041
`_fq_zech_poly_scalar_div_fq_zech` (*C function*), 1042
`_fq_zech_poly_scalar_mul_fq_zech` (*C function*), 1041

_fq_zech_poly_scalar_submul_fq_zech (C function), 1041
 _fq_zech_poly_set (C function), 1039
 _fq_zech_poly_set_length (C function), 1038
 _fq_zech_poly_shift_left (C function), 1047
 _fq_zech_poly_shift_right (C function), 1047
 _fq_zech_poly_sqr (C function), 1045
 _fq_zech_poly_sqr_KS (C function), 1045
 _fq_zech_poly_sqr_classical (C function), 1044
 _fq_zech_poly_sqrt (C function), 1052
 _fq_zech_poly_sqrt_series (C function), 1052
 _fq_zech_poly_sub (C function), 1041
 _fq_zech_poly_xgcd (C function), 1050
 _fq_zech_poly_xgcd_euclidean_f (C function), 1051
 _fq_zech_poly_zero (C function), 1039
 _fq_zech_pow (C function), 1026
 _fq_zech_reduce (C function), 1025
 _fq_zech_sparse_reduce (C function), 1025
 _fq_zech_vec_add (C function), 1031
 _fq_zech_vec_clear (C function), 1030
 _fq_zech_vec_dot (C function), 1031
 _fq_zech_vec_equal (C function), 1031
 _fq_zech_vec_fprint (C function), 1030
 _fq_zech_vec_init (C function), 1030
 _fq_zech_vec_is_zero (C function), 1031
 _fq_zech_vec_neg (C function), 1030
 _fq_zech_vec_print (C function), 1030
 _fq_zech_vec_randtest (C function), 1030
 _fq_zech_vec_scalar_addmul_fq_zech (C function), 1031
 _fq_zech_vec_scalar_submul_fq_zech (C function), 1031
 _fq_zech_vec_set (C function), 1030
 _fq_zech_vec_sub (C function), 1031
 _fq_zech_vec_swap (C function), 1030
 _fq_zech_vec_zero (C function), 1030
 _gr_ctx_init_mpn_mod (C function), 455
 _gr_ctx_qqbar_set_limits (C function), 55
 _gr_fmpz_poly_evaluate (C function), 61
 _gr_fmpz_poly_evaluate_horner (C function), 61
 _gr_fmpz_poly_evaluate_rectangular (C function), 61
 _gr_mat_charpoly (C function), 88
 _gr_mat_charpoly_berkowitz (C function), 88
 _gr_mat_charpoly_danilevsky (C function), 89
 _gr_mat_charpoly_danilevsky_inplace (C function), 89
 _gr_mat_charpoly_faddeev (C function), 89
 _gr_mat_charpoly_faddeev_bsgs (C function), 89
 _gr_mat_charpoly_from_hessenberg (C function), 89
 _gr_mat_charpoly_gauss (C function), 89
 _gr_mat_charpoly_generic (C function), 88
 _gr_mat_charpoly_householder (C function), 89
 _gr_mat_companion (C function), 90
 _gr_mat_companion_fraction (C function), 90
 _gr_mat_gr_poly_evaluate (C function), 84
 _gr_mat_gr_poly_solve_lode_newton (C function), 95
 _gr_mat_gr_poly_solve_lode_newton_start (C function), 95
 _gr_mat_gr_poly_solve_lode_newton_step (C function), 95
 _gr_mpoly_fit_length (C function), 126
 _gr_mpoly_normalise (C function), 123
 _gr_mpoly_push_exp_fmpz (C function), 127
 _gr_mpoly_push_exp_ui (C function), 126
 _gr_mpoly_set_length (C function), 126
 _gr_other_add_vec (C function), 74
 _gr_other_div_vec (C function), 74
 _gr_other_divexact_vec (C function), 74
 _gr_other_mul_vec (C function), 74
 _gr_other_pow_vec (C function), 74
 _gr_other_sub_vec (C function), 74
 _gr_poly_acos_series (C function), 116
 _gr_poly_acosh_series (C function), 116
 _gr_poly_add (C function), 99
 _gr_poly_asin_series (C function), 116
 _gr_poly_asinh_series (C function), 116
 _gr_poly_atan_series (C function), 116
 _gr_poly_atanh_series (C function), 116
 _gr_poly_compose (C function), 110
 _gr_poly_compose_divconquer (C function), 110
 _gr_poly_compose_horner (C function), 110
 _gr_poly_compose_mod (C function), 118
 _gr_poly_compose_mod_brent_kung (C function), 118
 _gr_poly_compose_mod_brent_kung_precomp_preinv (C function), 119
 _gr_poly_compose_mod_brent_kung_preinv (C function), 119
 _gr_poly_compose_mod_horner (C function), 118
 _gr_poly_compose_mod_horner_preinv (C function), 119
 _gr_poly_compose_mod_preinv (C function), 119
 _gr_poly_compose_series (C function), 111
 _gr_poly_compose_series_brent_kung (C function), 111
 _gr_poly_compose_series_divconquer (C function), 111
 _gr_poly_compose_series_horner (C function), 111
 _gr_poly_derivative (C function), 112
 _gr_poly_div (C function), 104
 _gr_poly_div_basecase (C function), 104
 _gr_poly_div_basecase_noinv (C function), 104
 _gr_poly_div_basecase_preinv1 (C function), 104
 _gr_poly_div_divconquer (C function), 104
 _gr_poly_div_divconquer_noinv (C function), 104

`_gr_poly_div_divconquer_preinv1` (*C function*), 104
`_gr_poly_div_newton` (*C function*), 104
`_gr_poly_div_newton_n_preinv` (*C function*), 104
`_gr_poly_div_series` (*C function*), 105
`_gr_poly_div_series_basecase` (*C function*), 105
`_gr_poly_div_series_basecase_noinv` (*C function*), 105
`_gr_poly_div_series_basecase_preinv1` (*C function*), 105
`_gr_poly_div_series_divconquer` (*C function*), 105
`_gr_poly_div_series_invmul` (*C function*), 105
`_gr_poly_div_series_newton` (*C function*), 105
`_gr_poly_divexact_basecase` (*C function*), 106
`_gr_poly_divexact_basecase_bidirectional` (*C function*), 106
`_gr_poly_divexact_basecase_noinv` (*C function*), 106
`_gr_poly_divexact_bidirectional` (*C function*), 106
`_gr_poly_divexact_series_basecase` (*C function*), 106
`_gr_poly_divexact_series_basecase_noinv` (*C function*), 106
`_gr_poly_divrem` (*C function*), 103
`_gr_poly_divrem_basecase` (*C function*), 103
`_gr_poly_divrem_basecase_noinv` (*C function*), 103
`_gr_poly_divrem_basecase_preinv1` (*C function*), 103
`_gr_poly_divrem_divconquer` (*C function*), 103
`_gr_poly_divrem_divconquer_noinv` (*C function*), 103
`_gr_poly_divrem_divconquer_preinv1` (*C function*), 103
`_gr_poly_divrem_newton` (*C function*), 103
`_gr_poly_divrem_newton_n_preinv` (*C function*), 104
`_gr_poly_equal` (*C function*), 99
`_gr_poly_evaluate` (*C function*), 107
`_gr_poly_evaluate_horner` (*C function*), 107
`_gr_poly_evaluate_modular` (*C function*), 107
`_gr_poly_evaluate_other` (*C function*), 107
`_gr_poly_evaluate_other_horner` (*C function*), 107
`_gr_poly_evaluate_other_rectangular` (*C function*), 107
`_gr_poly_evaluate_rectangular` (*C function*), 107
`_gr_poly_evaluate_vec_fast` (*C function*), 109
`_gr_poly_evaluate_vec_fast_precomp` (*C function*), 109
`_gr_poly_evaluate_vec_iter` (*C function*), 109
`_gr_poly_exp_series` (*C function*), 117
`_gr_poly_exp_series_basecase` (*C function*), 117
`_gr_poly_exp_series_basecase_mul` (*C function*), 117
`_gr_poly_exp_series_generic` (*C function*), 117
`_gr_poly_exp_series_newton` (*C function*), 117
`_gr_poly_gcd` (*C function*), 112
`_gr_poly_gcd_euclidean` (*C function*), 112
`_gr_poly_gcd_generic` (*C function*), 112
`_gr_poly_gcd_hgcd` (*C function*), 112
`_gr_poly_gcd_subresultant` (*C function*), 112
`_gr_poly_get_str` (*C function*), 98
`_gr_poly_hgcd` (*C function*), 112
`_gr_poly_integral` (*C function*), 112
`_gr_poly_interpolate` (*C function*), 109
`_gr_poly_interpolate_exact` (*C function*), 109
`_gr_poly_interpolate_fast` (*C function*), 110
`_gr_poly_interpolate_fast_precomp` (*C function*), 110
`_gr_poly_interpolation_weights` (*C function*), 110
`_gr_poly_inv_series` (*C function*), 105
`_gr_poly_inv_series_basecase` (*C function*), 105
`_gr_poly_inv_series_basecase_preinv1` (*C function*), 105
`_gr_poly_inv_series_newton` (*C function*), 105
`_gr_poly_is_monic` (*C function*), 112
`_gr_poly_log1p_series` (*C function*), 117
`_gr_poly_log_series` (*C function*), 117
`_gr_poly_make_monic` (*C function*), 112
`_gr_poly_mul` (*C function*), 99
`_gr_poly_mul_karatsuba` (*C function*), 100
`_gr_poly_mul_toom33` (*C function*), 100
`_gr_poly_mullov` (*C function*), 99
`_gr_poly_mullov_bivariate_KS` (*C function*), 100
`_gr_poly_mullov_classical` (*C function*), 100
`_gr_poly_mullov_complex_reorder` (*C function*), 100
`_gr_poly_mullov_generic` (*C function*), 99
`_gr_poly_mullov_toom_serial` (*C function*), 101
`_gr_poly_mulmid` (*C function*), 99
`_gr_poly_mulmid_bivariate_KS` (*C function*), 100
`_gr_poly_mulmid_classical` (*C function*), 100
`_gr_poly_mulmid_complex_reorder` (*C function*), 100
`_gr_poly_mulmid_generic` (*C function*), 99
`_gr_poly_mulmod` (*C function*), 118
`_gr_poly_mulmod_preinv` (*C function*), 118
`_gr_poly_newton_basis_evaluate` (*C function*), 108
`_gr_poly_newton_basis_from_monomial` (*C function*), 108
`_gr_poly_newton_basis_interpolate` (*C function*), 108
`_gr_poly_newton_basis_interpolate_exact` (*C function*), 108

`_gr_poly_newton_basis_to_monomial` (*C function*), 108
`_gr_poly_normalise` (*C function*), 98
`_gr_poly_nth_derivative` (*C function*), 112
`_gr_poly_pow_series_fmpq_recurrence` (*C function*), 102
`_gr_poly_pow_series_ui` (*C function*), 102
`_gr_poly_pow_series_ui_binexp` (*C function*), 102
`_gr_poly_pow_ui` (*C function*), 102
`_gr_poly_pow_ui_binexp` (*C function*), 102
`_gr_poly_powmod_fmpz_binexp` (*C function*), 118
`_gr_poly_powmod_fmpz_binexp_preinv` (*C function*), 118
`_gr_poly_powmod_fmpz_sliding_preinv` (*C function*), 118
`_gr_poly_powmod_ui_binexp` (*C function*), 118
`_gr_poly_powmod_ui_binexp_preinv` (*C function*), 118
`_gr_poly_powmod_x_fmpz_preinv` (*C function*), 118
`_gr_poly_precompute_matrix` (*C function*), 119
`_gr_poly_product_roots` (*C function*), 109
`_gr_poly_reduce_matrix_mod_poly` (*C function*), 119
`_gr_poly_refine_roots_aberth` (*C function*), 116
`_gr_poly_refine_roots_wdk` (*C function*), 116
`_gr_poly_rem` (*C function*), 104
`_gr_poly_resultant` (*C function*), 113
`_gr_poly_resultant_euclidean` (*C function*), 113
`_gr_poly_resultant_hgcd` (*C function*), 113
`_gr_poly_resultant_small` (*C function*), 113
`_gr_poly_resultant_sylvester` (*C function*), 113
`_gr_poly_reverse` (*C function*), 98
`_gr_poly_revert_series` (*C function*), 111
`_gr_poly_revert_series_lagrange` (*C function*), 111
`_gr_poly_revert_series_lagrange_fast` (*C function*), 111
`_gr_poly_revert_series_newton` (*C function*), 111
`_gr_poly_rsqrts_series` (*C function*), 107
`_gr_poly_rsqrts_series_basecase` (*C function*), 107
`_gr_poly_rsqrts_series_miller` (*C function*), 107
`_gr_poly_rsqrts_series_newton` (*C function*), 107
`_gr_poly_set_length` (*C function*), 98
`_gr_poly_set_str` (*C function*), 98
`_gr_poly_shift_left` (*C function*), 102
`_gr_poly_shift_right` (*C function*), 102
`_gr_poly_shiftless_decomposition_from_factors` (*C function*), 115
`_gr_poly_sin_cos_series_basecase` (*C function*), 117
`_gr_poly_sin_cos_series_tangent` (*C function*), 117
`_gr_poly_sqrt_series` (*C function*), 107
`_gr_poly_sqrt_series_basecase` (*C function*), 107
`_gr_poly_sqrt_series_miller` (*C function*), 107
`_gr_poly_sqrt_series_newton` (*C function*), 107
`_gr_poly_sub` (*C function*), 99
`_gr_poly_tan_series` (*C function*), 117
`_gr_poly_tan_series_basecase` (*C function*), 117
`_gr_poly_tan_series_newton` (*C function*), 117
`_gr_poly_taylor_shift` (*C function*), 110
`_gr_poly_taylor_shift_convolution` (*C function*), 110
`_gr_poly_taylor_shift_divconquer` (*C function*), 110
`_gr_poly_taylor_shift_horner` (*C function*), 110
`_gr_poly_test_approx_mulmid_pos_entrywise_accurate` (*C function*), 120
`_gr_poly_test_div` (*C function*), 120
`_gr_poly_test_div_series` (*C function*), 120
`_gr_poly_test_divrem` (*C function*), 120
`_gr_poly_test_gcd` (*C function*), 120
`_gr_poly_test_inv_series` (*C function*), 120
`_gr_poly_test_mulmod` (*C function*), 120
`_gr_poly_test_mulmid` (*C function*), 120
`_gr_poly_test_xgcd` (*C function*), 120
`_gr_poly_tree_alloc` (*C function*), 109
`_gr_poly_tree_build` (*C function*), 109
`_gr_poly_tree_free` (*C function*), 109
`_gr_poly_write` (*C function*), 98
`_gr_poly_xgcd` (*C function*), 113
`_gr_poly_xgcd_euclidean` (*C function*), 113
`_gr_poly_xgcd_generic` (*C function*), 113
`_gr_poly_xgcd_hgcd` (*C function*), 113
`_gr_scalar_add_vec` (*C function*), 74
`_gr_scalar_div_vec` (*C function*), 74
`_gr_scalar_divexact_vec` (*C function*), 74
`_gr_scalar_mul_vec` (*C function*), 74
`_gr_scalar_other_add_vec` (*C function*), 75
`_gr_scalar_other_div_vec` (*C function*), 75
`_gr_scalar_other_divexact_vec` (*C function*), 75
`_gr_scalar_other_mul_vec` (*C function*), 75
`_gr_scalar_other_pow_vec` (*C function*), 75
`_gr_scalar_other_sub_vec` (*C function*), 75
`_gr_scalar_pow_vec` (*C function*), 74
`_gr_scalar_sub_vec` (*C function*), 74
`_gr_series_get_error` (*C function*), 130
`_gr_series_is_exact` (*C function*), 130
`_gr_series_make_exact` (*C function*), 130
`_gr_series_set_error` (*C function*), 130
`_gr_vec_add` (*C function*), 74
`_gr_vec_add_other` (*C function*), 74
`_gr_vec_add_scalar` (*C function*), 74

`_gr_vec_add_scalar_fmpq` (*C function*), 75
`_gr_vec_add_scalar_fmpz` (*C function*), 75
`_gr_vec_add_scalar_other` (*C function*), 75
`_gr_vec_add_scalar_si` (*C function*), 75
`_gr_vec_add_scalar_ui` (*C function*), 75
`_gr_vec_addmul_scalar` (*C function*), 76
`_gr_vec_addmul_scalar_fmpz` (*C function*), 76
`_gr_vec_addmul_scalar_si` (*C function*), 76
`_gr_vec_clear` (*C function*), 73
`_gr_vec_contains` (*C function*), 77
`_gr_vec_div` (*C function*), 74
`_gr_vec_div_other` (*C function*), 74
`_gr_vec_div_scalar` (*C function*), 74
`_gr_vec_div_scalar_fmpq` (*C function*), 75
`_gr_vec_div_scalar_fmpz` (*C function*), 75
`_gr_vec_div_scalar_other` (*C function*), 75
`_gr_vec_div_scalar_si` (*C function*), 75
`_gr_vec_div_scalar_ui` (*C function*), 75
`_gr_vec_divexact` (*C function*), 74
`_gr_vec_divexact_other` (*C function*), 74
`_gr_vec_divexact_scalar` (*C function*), 74
`_gr_vec_divexact_scalar_fmpq` (*C function*), 75
`_gr_vec_divexact_scalar_fmpz` (*C function*), 75
`_gr_vec_divexact_scalar_other` (*C function*), 75
`_gr_vec_divexact_scalar_si` (*C function*), 75
`_gr_vec_divexact_scalar_ui` (*C function*), 75
`_gr_vec_dot` (*C function*), 76
`_gr_vec_dot_fmpz` (*C function*), 76
`_gr_vec_dot_rev` (*C function*), 76
`_gr_vec_dot_si` (*C function*), 76
`_gr_vec_dot_ui` (*C function*), 76
`_gr_vec_equal` (*C function*), 74
`_gr_vec_init` (*C function*), 73
`_gr_vec_is_zero` (*C function*), 74
`_gr_vec_mul` (*C function*), 74
`_gr_vec_mul_other` (*C function*), 74
`_gr_vec_mul_scalar` (*C function*), 74
`_gr_vec_mul_scalar_2exp_si` (*C function*), 76
`_gr_vec_mul_scalar_fmpq` (*C function*), 75
`_gr_vec_mul_scalar_fmpz` (*C function*), 75
`_gr_vec_mul_scalar_other` (*C function*), 75
`_gr_vec_mul_scalar_si` (*C function*), 75
`_gr_vec_mul_scalar_ui` (*C function*), 75
`_gr_vec_neg` (*C function*), 74
`_gr_vec_normalise` (*C function*), 74
`_gr_vec_normalise_weak` (*C function*), 74
`_gr_vec_permute` (*C function*), 77
`_gr_vec_pow` (*C function*), 74
`_gr_vec_pow_other` (*C function*), 74
`_gr_vec_pow_scalar` (*C function*), 74
`_gr_vec_pow_scalar_fmpq` (*C function*), 75
`_gr_vec_pow_scalar_fmpz` (*C function*), 75
`_gr_vec_pow_scalar_other` (*C function*), 75
`_gr_vec_pow_scalar_si` (*C function*), 75
`_gr_vec_pow_scalar_ui` (*C function*), 75
`_gr_vec_product` (*C function*), 76
`_gr_vec_randtest` (*C function*), 73
`_gr_vec_reciprocals` (*C function*), 77
`_gr_vec_set` (*C function*), 73
`_gr_vec_set_powers` (*C function*), 77
`_gr_vec_shuffle` (*C function*), 77
`_gr_vec_sort` (*C function*), 77
`_gr_vec_step` (*C function*), 77
`_gr_vec_sub` (*C function*), 74
`_gr_vec_sub_other` (*C function*), 74
`_gr_vec_sub_scalar` (*C function*), 74
`_gr_vec_sub_scalar_fmpq` (*C function*), 75
`_gr_vec_sub_scalar_fmpz` (*C function*), 75
`_gr_vec_sub_scalar_other` (*C function*), 75
`_gr_vec_sub_scalar_si` (*C function*), 75
`_gr_vec_sub_scalar_ui` (*C function*), 75
`_gr_vec_submul_scalar` (*C function*), 76
`_gr_vec_submul_scalar_si` (*C function*), 76
`_gr_vec_sum` (*C function*), 76
`_gr_vec_swap` (*C function*), 73
`_gr_vec_write` (*C function*), 73
`_gr_vec_zero` (*C function*), 74
`_mag_vec_clear` (*C function*), 584
`_mag_vec_init` (*C function*), 584
`_mpn_mod_poly_div` (*C function*), 459
`_mpn_mod_poly_div_series` (*C function*), 459
`_mpn_mod_poly_divrem` (*C function*), 459
`_mpn_mod_poly_divrem_basecase` (*C function*), 459
`_mpn_mod_poly_divrem_basecase_preinv1` (*C function*), 459
`_mpn_mod_poly_divrem_q1_preinv1` (*C function*), 459
`_mpn_mod_poly_divrem_q1_preinv1_fmma` (*C function*), 459
`_mpn_mod_poly_divrem_q1_preinv1_fmma_precond` (*C function*), 459
`_mpn_mod_poly_divrem_q1_preinv1_karatsuba_precond` (*C function*), 459
`_mpn_mod_poly_gcd` (*C function*), 459
`_mpn_mod_poly_inv_series` (*C function*), 459
`_mpn_mod_poly_mullo` (*C function*), 458
`_mpn_mod_poly_mullo_KS` (*C function*), 458
`_mpn_mod_poly_mullo_classical` (*C function*), 458
`_mpn_mod_poly_mullo_fft_small` (*C function*), 458
`_mpn_mod_poly_mullo_karatsuba` (*C function*), 458
`_mpn_mod_poly_mulmid` (*C function*), 458
`_mpn_mod_poly_mulmid_KS` (*C function*), 458
`_mpn_mod_poly_mulmid_classical` (*C function*), 458
`_mpn_mod_poly_mulmid_fft_small` (*C function*), 458
`_mpn_mod_poly_mulmid_karatsuba` (*C function*), 458
`_mpn_mod_poly_xgcd` (*C function*), 459
`_mpn_mod_scalar_mul_vec` (*C function*), 457
`_mpn_mod_vec_add` (*C function*), 457

_mpn_mod_vec_addmul_scalar (*C function*), 457
 _mpn_mod_vec_clear (*C function*), 457
 _mpn_mod_vec_dot (*C function*), 457
 _mpn_mod_vec_dot_rev (*C function*), 457
 _mpn_mod_vec_mul (*C function*), 457
 _mpn_mod_vec_mul_scalar (*C function*), 457
 _mpn_mod_vec_neg (*C function*), 457
 _mpn_mod_vec_set (*C function*), 457
 _mpn_mod_vec_sub (*C function*), 457
 _mpn_mod_vec_submul_scalar (*C function*), 457
 _mpn_mod_vec_swap (*C function*), 457
 _mpn_mod_vec_zero (*C function*), 457
 _mpoly_heap_insert (*C function*), 31
 _mpoly_heap_insert1 (*C function*), 31
 _mpoly_heap_pop (*C function*), 31
 _mpoly_heap_pop1 (*C function*), 31
 _mul_precomp_clear (*C function*), 307
 _mul_precomp_init (*C function*), 307
 _n_randint (*C function*), 136
 _n_randlimb (*C function*), 136
 _nf_elem_add (*C function*), 539
 _nf_elem_div (*C function*), 540
 _nf_elem_equal (*C function*), 539
 _nf_elem_inv (*C function*), 540
 _nf_elem_invertible_check (*C function*), 538
 _nf_elem_mul (*C function*), 540
 _nf_elem_mul_red (*C function*), 540
 _nf_elem_norm (*C function*), 540
 _nf_elem_norm_div (*C function*), 540
 _nf_elem_pow (*C function*), 540
 _nf_elem_reduce (*C function*), 537
 _nf_elem_set_coeff_num_fmpz (*C function*), 539
 _nf_elem_sub (*C function*), 540
 _nf_elem_trace (*C function*), 540
 _nfixed_complex_mat_mul_bound (*C function*), 599
 _nfixed_dot_2 (*C function*), 599
 _nfixed_dot_3 (*C function*), 599
 _nfixed_dot_4 (*C function*), 599
 _nfixed_dot_5 (*C function*), 599
 _nfixed_dot_6 (*C function*), 599
 _nfixed_dot_7 (*C function*), 599
 _nfixed_dot_8 (*C function*), 599
 _nfixed_mat_mul (*C function*), 599
 _nfixed_mat_mul_bound (*C function*), 599
 _nfixed_mat_mul_bound_classical (*C function*), 599
 _nfixed_mat_mul_bound_strassen (*C function*), 599
 _nfixed_mat_mul_bound_waksman (*C function*), 599
 _nfixed_mat_mul_classical (*C function*), 599
 _nfixed_mat_mul_classical_precise (*C function*), 599
 _nfixed_mat_mul_strassen (*C function*), 599
 _nfixed_mat_mul_waksman (*C function*), 599
 _nfixed_print (*C function*), 598
 _nfixed_vec_add (*C function*), 598
 _nfixed_vec_sub (*C function*), 598
 _nfloat_add_1 (*C function*), 596
 _nfloat_add_2 (*C function*), 596
 _nfloat_add_3 (*C function*), 596
 _nfloat_add_4 (*C function*), 596
 _nfloat_add_n (*C function*), 596
 _nfloat_cmp (*C function*), 596
 _nfloat_cmpabs (*C function*), 596
 _nfloat_complex_mul_karatsuba (*C function*), 596
 _nfloat_complex_mul_naive (*C function*), 596
 _nfloat_complex_mul_standard (*C function*), 596
 _nfloat_complex_sqr (*C function*), 596
 _nfloat_complex_sqr_karatsuba (*C function*), 596
 _nfloat_complex_sqr_naive (*C function*), 596
 _nfloat_complex_sqr_standard (*C function*), 596
 _nfloat_complex_vec_add (*C function*), 596
 _nfloat_complex_vec_clear (*C function*), 596
 _nfloat_complex_vec_init (*C function*), 596
 _nfloat_complex_vec_set (*C function*), 596
 _nfloat_complex_vec_sub (*C function*), 596
 _nfloat_complex_vec_zero (*C function*), 596
 _nfloat_overflow (*C function*), 596
 _nfloat_set_mpn_2exp (*C function*), 594
 _nfloat_sub_1 (*C function*), 596
 _nfloat_sub_2 (*C function*), 596
 _nfloat_sub_3 (*C function*), 596
 _nfloat_sub_4 (*C function*), 596
 _nfloat_sub_n (*C function*), 596
 _nfloat_underflow (*C function*), 596
 _nfloat_vec_add (*C function*), 595
 _nfloat_vec_addmul_scalar (*C function*), 595
 _nfloat_vec_clear (*C function*), 595
 _nfloat_vec_dot (*C function*), 595
 _nfloat_vec_dot_rev (*C function*), 595
 _nfloat_vec_init (*C function*), 595
 _nfloat_vec_mul (*C function*), 595
 _nfloat_vec_mul_scalar (*C function*), 595
 _nfloat_vec_set (*C function*), 595
 _nfloat_vec_sub (*C function*), 595
 _nfloat_vec_submul_scalar (*C function*), 595
 _nfloat_vec_zero (*C function*), 595
 _nmod_add (*C function*), 382
 _nmod_mat_mul_classical_op (*C function*), 393
 _nmod_mat_mul_classical_threaded_op (*C function*), 393
 _nmod_mat_mul_classical_threaded_pool_op (*C function*), 393
 _nmod_mat_pow (*C function*), 394
 _nmod_mul_fullword (*C function*), 383
 _nmod_poly_KS2_pack (*C function*), 404
 _nmod_poly_KS2_pack1 (*C function*), 404
 _nmod_poly_KS2_recover_reduce (*C function*), 405

`_nmod_poly_KS2_recover_reduce1` (*C function*), 405
`_nmod_poly_KS2_recover_reduce2` (*C function*), 405
`_nmod_poly_KS2_recover_reduce2b` (*C function*), 405
`_nmod_poly_KS2_recover_reduce3` (*C function*), 405
`_nmod_poly_KS2_reduce` (*C function*), 405
`_nmod_poly_KS2_unpack` (*C function*), 404
`_nmod_poly_KS2_unpack1` (*C function*), 404
`_nmod_poly_KS2_unpack2` (*C function*), 404
`_nmod_poly_KS2_unpack3` (*C function*), 404
`_nmod_poly_add` (*C function*), 403
`_nmod_poly_asin_series` (*C function*), 428
`_nmod_poly_asinh_series` (*C function*), 429
`_nmod_poly_atan_series` (*C function*), 428
`_nmod_poly_atanh_series` (*C function*), 428
`_nmod_poly_bit_pack` (*C function*), 404
`_nmod_poly_bit_unpack` (*C function*), 404
`_nmod_poly_compose` (*C function*), 418
`_nmod_poly_compose_divconquer` (*C function*), 418
`_nmod_poly_compose_horner` (*C function*), 418
`_nmod_poly_compose_mod` (*C function*), 421
`_nmod_poly_compose_mod_brent_kung` (*C function*), 419
`_nmod_poly_compose_mod_brent_kung_precomp_preinv` (*C function*), 420
`_nmod_poly_compose_mod_brent_kung_precomp_preinv_worker` (*C function*), 420
`_nmod_poly_compose_mod_brent_kung_preinv` (*C function*), 419
`_nmod_poly_compose_mod_brent_kung_vec_preinv` (*C function*), 420
`_nmod_poly_compose_mod_brent_kung_vec_preinv_nmodded` (*C function*), 421
`_nmod_poly_compose_mod_horner` (*C function*), 419
`_nmod_poly_compose_series` (*C function*), 426
`_nmod_poly_conway` (*C function*), 430
`_nmod_poly_conway_rand` (*C function*), 430
`_nmod_poly_cos_series` (*C function*), 429
`_nmod_poly_cosh_series` (*C function*), 429
`_nmod_poly_derivative` (*C function*), 414
`_nmod_poly_discriminant` (*C function*), 426
`_nmod_poly_div` (*C function*), 411
`_nmod_poly_div_newton_n_preinv` (*C function*), 413
`_nmod_poly_div_root` (*C function*), 413
`_nmod_poly_div_series` (*C function*), 412
`_nmod_poly_div_series_basecase` (*C function*), 412
`_nmod_poly_divexact` (*C function*), 412
`_nmod_poly_divides` (*C function*), 414
`_nmod_poly_divides_classical` (*C function*), 414
`_nmod_poly_divrem` (*C function*), 411
`_nmod_poly_divrem_basecase` (*C function*), 411
`_nmod_poly_divrem_mpn_ctx` (*C function*), 306
`_nmod_poly_divrem_newton_n_preinv` (*C function*), 413
`_nmod_poly_divrem_precomp` (*C function*), 307
`_nmod_poly_divrem_precomp_clear` (*C function*), 307
`_nmod_poly_divrem_precomp_init` (*C function*), 307
`_nmod_poly_evaluate_geometric_nmod_vec_fast` (*C function*), 416
`_nmod_poly_evaluate_geometric_nmod_vec_fast_precomp` (*C function*), 416
`_nmod_poly_evaluate_geometric_nmod_vec_iter` (*C function*), 416
`_nmod_poly_evaluate_nmod` (*C function*), 415
`_nmod_poly_evaluate_nmod_precomp` (*C function*), 414
`_nmod_poly_evaluate_nmod_precomp_lazy` (*C function*), 414
`_nmod_poly_evaluate_nmod_vec` (*C function*), 416
`_nmod_poly_evaluate_nmod_vec_fast` (*C function*), 415
`_nmod_poly_evaluate_nmod_vec_fast_precomp` (*C function*), 415
`_nmod_poly_evaluate_nmod_vec_iter` (*C function*), 415
`_nmod_poly_exp_expinv_series` (*C function*), 428
`_nmod_poly_exp_series` (*C function*), 428
`_nmod_poly_gcd` (*C function*), 422
`_nmod_poly_gcd_euclidean` (*C function*), 422
`_nmod_poly_gcd_euclidean_redc_half` (*C function*), 422
`_nmod_poly_gcd_reduced` (*C function*), 422
`_nmod_poly_gcd_hgcd` (*C function*), 422
`_nmod_poly_gcdinv` (*C function*), 425
`_nmod_poly_hgcd` (*C function*), 422
`_nmod_poly_integral` (*C function*), 414
`_nmod_poly_interpolate_geometric_nmod_vec_fast_precomp` (*C function*), 417
`_nmod_poly_interpolate_nmod_vec` (*C function*), 417
`_nmod_poly_interpolate_nmod_vec_barycentric` (*C function*), 417
`_nmod_poly_interpolate_nmod_vec_fast` (*C function*), 417
`_nmod_poly_interpolate_nmod_vec_fast_precomp` (*C function*), 417
`_nmod_poly_interpolate_nmod_vec_newton` (*C function*), 417
`_nmod_poly_interpolation_weights` (*C function*), 417
`_nmod_poly_interval_poly_worker` (*C function*), 442
`_nmod_poly_inv_series` (*C function*), 412
`_nmod_poly_inv_series_basecase` (*C function*), 412

`_nmod_poly_inv_series_newton` (*C function*), 412
`_nmod_poly_invmod` (*C function*), 425
`_nmod_poly_invsqrt_series` (*C function*), 426
`_nmod_poly_is_squarefree` (*C function*), 441
`_nmod_poly_log_series` (*C function*), 428
`_nmod_poly_make_monic` (*C function*), 404
`_nmod_poly_mod_matrix_rows_evaluate` (*C function*), 420
`_nmod_poly_mul` (*C function*), 406
`_nmod_poly_mul_KS` (*C function*), 405
`_nmod_poly_mul_KS2` (*C function*), 406
`_nmod_poly_mul_KS4` (*C function*), 406
`_nmod_poly_mul_classical` (*C function*), 405
`_nmod_poly_mul_mid_default_mpn_ctx` (*C function*), 306
`_nmod_poly_mul_mid_mpn_ctx` (*C function*), 306
`_nmod_poly_mul_mid_precomp` (*C function*), 307
`_nmod_poly_mulhigh` (*C function*), 407
`_nmod_poly_mulhigh_classical` (*C function*), 405
`_nmod_poly_mulloy` (*C function*), 406
`_nmod_poly_mulloy_KS` (*C function*), 406
`_nmod_poly_mulloy_classical` (*C function*), 405
`_nmod_poly_mulloy_fft_small` (*C function*), 407
`_nmod_poly_mulloy_want_fft_small` (*C function*), 407
`_nmod_poly_mulmid` (*C function*), 406
`_nmod_poly_mulmid_KS` (*C function*), 406
`_nmod_poly_mulmid_classical` (*C function*), 406
`_nmod_poly_mulmid_fft_small` (*C function*), 407
`_nmod_poly_mulmid_fft_small_repack` (*C function*), 407
`_nmod_poly_mulmod` (*C function*), 407
`_nmod_poly_mulmod_precond` (*C function*), 408
`_nmod_poly_mulmod_precond_init_method` (*C function*), 408
`_nmod_poly_mulmod_precond_init_num` (*C function*), 408
`_nmod_poly_mulmod_preinv` (*C function*), 407
`_nmod_poly_multi crt_local_size` (*C function*), 432
`_nmod_poly_multi crt_run` (*C function*), 432
`_nmod_poly_multi crt_run_p` (*C function*), 432
`_nmod_poly_normalise` (*C function*), 399
`_nmod_poly_pow` (*C function*), 409
`_nmod_poly_pow_binexp` (*C function*), 409
`_nmod_poly_pow_trunc` (*C function*), 409
`_nmod_poly_pow_trunc_binexp` (*C function*), 409
`_nmod_poly_power_sums` (*C function*), 427
`_nmod_poly_power_sums_naive` (*C function*), 427
`_nmod_poly_power_sums_schoenhage` (*C function*), 427
`_nmod_poly_power_sums_to_poly` (*C function*), 427
`_nmod_poly_power_sums_to_poly_naive` (*C function*), 427
`_nmod_poly_power_sums_to_poly_schoenhage` (*C function*), 427
`_nmod_poly_powers_mod_preinv_naive` (*C function*), 410
`_nmod_poly_powers_mod_preinv_threaded` (*C function*), 411
`_nmod_poly_powers_mod_preinv_threaded_pool` (*C function*), 411
`_nmod_poly_powmod_fmpz_binexp` (*C function*), 409
`_nmod_poly_powmod_fmpz_binexp_preinv` (*C function*), 410
`_nmod_poly_powmod_ui_binexp` (*C function*), 409
`_nmod_poly_powmod_ui_binexp_preinv` (*C function*), 410
`_nmod_poly_powmod_x_fmpz_preinv` (*C function*), 410
`_nmod_poly_powmod_x_ui_preinv` (*C function*), 410
`_nmod_poly_precompute_matrix` (*C function*), 420
`_nmod_poly_precompute_matrix_worker` (*C function*), 420
`_nmod_poly_product_roots_nmod_vec` (*C function*), 430
`_nmod_poly_reduce_matrix_mod_poly` (*C function*), 420
`_nmod_poly_rem` (*C function*), 412
`_nmod_poly_rem_q1` (*C function*), 411
`_nmod_poly_resultant` (*C function*), 425
`_nmod_poly_resultant_euclidean` (*C function*), 424
`_nmod_poly_resultant_hgcd` (*C function*), 424
`_nmod_poly_reverse` (*C function*), 400
`_nmod_poly_revert_series` (*C function*), 426
`_nmod_poly_shift_left` (*C function*), 403
`_nmod_poly_shift_right` (*C function*), 403
`_nmod_poly_sin_series` (*C function*), 429
`_nmod_poly_sinh_series` (*C function*), 429
`_nmod_poly_sqrt` (*C function*), 427
`_nmod_poly_sqrt_series` (*C function*), 426
`_nmod_poly_sub` (*C function*), 403
`_nmod_poly_tan_series` (*C function*), 429
`_nmod_poly_tanh_series` (*C function*), 429
`_nmod_poly_taylor_shift` (*C function*), 419
`_nmod_poly_taylor_shift_convolution` (*C function*), 419
`_nmod_poly_taylor_shift_horner` (*C function*), 419
`_nmod_poly_tree_alloc` (*C function*), 430
`_nmod_poly_tree_build` (*C function*), 430
`_nmod_poly_tree_free` (*C function*), 430
`_nmod_poly_xgcd` (*C function*), 423
`_nmod_poly_xgcd_euclidean` (*C function*), 423
`_nmod_poly_xgcd_hgcd` (*C function*), 423
`_nmod_redc_2_pow_ui` (*C function*), 384
`_nmod_redc_fast_2_pow_ui` (*C function*), 384
`_nmod_redc_fast_pow_ui` (*C function*), 384

`_nmod_redc_pow_ui` (*C function*), 384
`_nmod_sub` (*C function*), 382
`_nmod_vec_add` (*C function*), 387
`_nmod_vec_clear` (*C function*), 385
`_nmod_vec_dot` (*C function*), 388
`_nmod_vec_dot_bound_limbs` (*C function*), 388
`_nmod_vec_dot_bound_limbs_from_params` (*C function*), 388
`_nmod_vec_dot_params` (*C function*), 388
`_nmod_vec_dot_ptr` (*C function*), 388
`_nmod_vec_dot_rev` (*C function*), 388
`_nmod_vec_equal` (*C function*), 386
`_nmod_vec_fprint` (*C function*), 386
`_nmod_vec_fprint_pretty` (*C function*), 386
`_nmod_vec_init` (*C function*), 385
`_nmod_vec_invert` (*C function*), 387
`_nmod_vec_max_bits` (*C function*), 386
`_nmod_vec_neg` (*C function*), 387
`_nmod_vec_print` (*C function*), 386
`_nmod_vec_print_pretty` (*C function*), 386
`_nmod_vec_rand` (*C function*), 386
`_nmod_vec_randtest` (*C function*), 386
`_nmod_vec_reduce` (*C function*), 386
`_nmod_vec_scalar_addmul_nmod` (*C function*), 387
`_nmod_vec_scalar_mul_nmod` (*C function*), 387
`_nmod_vec_scalar_mul_nmod_redc` (*C function*), 387
`_nmod_vec_scalar_mul_nmod_shoup` (*C function*), 387
`_nmod_vec_set` (*C function*), 386
`_nmod_vec_sub` (*C function*), 387
`_nmod_vec_swap` (*C function*), 386
`_nmod_vec_zero` (*C function*), 386
`_padic_canonicalise` (*C function*), 1064
`_padic_ctx_pow_ui` (*C function*), 1064
`_padic_exp` (*C function*), 1067
`_padic_exp_balanced` (*C function*), 1067
`_padic_exp_bound` (*C function*), 1067
`_padic_exp_rectangular` (*C function*), 1067
`_padic_fprint` (*C function*), 1070
`_padic_inv` (*C function*), 1067
`_padic_inv_clear` (*C function*), 1066
`_padic_inv_precomp` (*C function*), 1066
`_padic_inv_precompute` (*C function*), 1066
`_padic_lifts_exps` (*C function*), 1066
`_padic_lifts_pows` (*C function*), 1066
`_padic_log` (*C function*), 1068
`_padic_log_balanced` (*C function*), 1068
`_padic_log_bound` (*C function*), 1068
`_padic_log_rectangular` (*C function*), 1068
`_padic_log_satoh` (*C function*), 1068
`_padic_mat_add` (*C function*), 1080
`_padic_mat_canonicalise` (*C function*), 1078
`_padic_mat_mul` (*C function*), 1081
`_padic_mat_neg` (*C function*), 1080
`_padic_mat_reduce` (*C function*), 1078
`_padic_mat_scalar_mul_fmpz` (*C function*), 1080
`_padic_mat_scalar_mul_padic` (*C function*), 1080
`_padic_mat_sub` (*C function*), 1080
`_padic_poly_add` (*C function*), 1073
`_padic_poly_canonicalise` (*C function*), 1071
`_padic_poly_compose` (*C function*), 1076
`_padic_poly_compose_pow` (*C function*), 1076
`_padic_poly_derivative` (*C function*), 1075
`_padic_poly_evaluate_padic` (*C function*), 1075
`_padic_poly_fprint` (*C function*), 1076
`_padic_poly_fprint_pretty` (*C function*), 1076
`_padic_poly_is_canonical` (*C function*), 1077
`_padic_poly_is_reduced` (*C function*), 1077
`_padic_poly_mul` (*C function*), 1074
`_padic_poly_normalise` (*C function*), 1071
`_padic_poly_pow` (*C function*), 1074
`_padic_poly_print` (*C function*), 1076
`_padic_poly_print_pretty` (*C function*), 1076
`_padic_poly_scalar_mul_padic` (*C function*), 1074
`_padic_poly_set_length` (*C function*), 1070
`_padic_poly_sub` (*C function*), 1073
`_padic_print` (*C function*), 1070
`_padic_reduce` (*C function*), 1064
`_padic_teichmuller` (*C function*), 1069
`_perm_clear` (*C function*), 521
`_perm_compose` (*C function*), 521
`_perm_init` (*C function*), 521
`_perm_inv` (*C function*), 521
`_perm_one` (*C function*), 521
`_perm_parity` (*C function*), 521
`_perm_randtest` (*C function*), 522
`_perm_set` (*C function*), 521
`_qadic_char2_sqrt_precomp_clear` (*C function*), 1085
`_qadic_char2_sqrt_precomp_init` (*C function*), 1085
`_qadic_char2_sqrt_with_precomp` (*C function*), 1085
`_qadic_exp` (*C function*), 1085
`_qadic_exp_balanced` (*C function*), 1085
`_qadic_exp_rectangular` (*C function*), 1085
`_qadic_frobenius` (*C function*), 1087
`_qadic_frobenius_a` (*C function*), 1087
`_qadic_inv` (*C function*), 1084
`_qadic_log` (*C function*), 1086
`_qadic_log_balanced` (*C function*), 1086
`_qadic_log_rectangular` (*C function*), 1086
`_qadic_norm` (*C function*), 1087
`_qadic_pow` (*C function*), 1084
`_qadic_teichmuller` (*C function*), 1087
`_qadic_trace` (*C function*), 1087
`_qqbar_acb_lindep` (*C function*), 557
`_qqbar_enclosure_raw` (*C function*), 557
`_qqbar_evaluate_fmpq_poly` (*C function*), 551
`_qqbar_evaluate_fmpz_poly` (*C function*), 551
`_qqbar_get_fmpq` (*C function*), 546

_qqbar_roots_poly_squarefree (*C function*), 552
 _qqbar_validate_existence_uniqueness (*C function*), 557
 _qqbar_validate_uniqueness (*C function*), 557
 _qqbar_vec_clear (*C function*), 545
 _qqbar_vec_init (*C function*), 545
 _radix_size_digits (*C function*), 310
 _radix_size_digits_1 (*C function*), 310
 _radix_valuation_digits_1 (*C function*), 310
 _unity_zp (*C type*), 288
 _unity_zp_pow_select_k (*C function*), 289
 _unity_zp_reduce_cyclotomic (*C function*), 290
 _unity_zp_reduce_cyclotomic_divmod (*C function*), 290
 _unity_zpq (*C type*), 288
 _unity_zpq_mul_unity_p (*C function*), 291
 acb_calc_integrate_opt_t.deg_limit (*C member*), 804
 acb_calc_integrate_opt_t.depth_limit (*C member*), 805
 acb_calc_integrate_opt_t.eval_limit (*C member*), 804
 acb_calc_integrate_opt_t.use_heap (*C member*), 805
 acb_calc_integrate_opt_t.verbose (*C member*), 805
 acb_calc_integrate_taylor (*C function*), 806
 acb_chebyshev_t2_ui (*C function*), 652
 acb_chebyshev_t_ui (*C function*), 652
 acb_chebyshev_u2_ui (*C function*), 652
 acb_chebyshev_u_ui (*C function*), 652
 acb_clear (*C function*), 639
 acb_conj (*C function*), 643
 acb_const_pi (*C function*), 646
 acb_contains (*C function*), 642
 acb_contains_fmpq (*C function*), 642
 acb_contains_fmpz (*C function*), 642
 acb_contains_int (*C function*), 642
 acb_contains_interior (*C function*), 642
 acb_contains_zero (*C function*), 642
 acb_cos (*C function*), 647
 acb_cos_pi (*C function*), 647
 acb_cosh (*C function*), 648
 acb_cot (*C function*), 647
 acb_cot_pi (*C function*), 647
 acb_coth (*C function*), 648
 acb_csc (*C function*), 648
 acb_csc_pi (*C function*), 648
 acb_csch (*C function*), 648
 acb_csgn (*C function*), 643
 acb_cube (*C function*), 644
 acb_dft (*C function*), 690
 acb_dft_bluestein (*C function*), 693
 acb_dft_bluestein_clear (*C function*), 693
 acb_dft_bluestein_init (*C function*), 693
 acb_dft_bluestein_precomp (*C function*), 693
 acb_dft_bluestein_struct (*C type*), 693
 acb_dft_bluestein_t (*C type*), 693
 acb_dft_convolve (*C function*), 691
 acb_dft_convolve_naive (*C function*), 691
 acb_dft_convolve_rad2 (*C function*), 691
 acb_dft_crt (*C function*), 692
 acb_dft_crt_clear (*C function*), 692
 acb_dft_crt_init (*C function*), 692
 acb_dft_crt_precomp (*C function*), 692
 acb_dft_crt_struct (*C type*), 692
 acb_dft_crt_t (*C type*), 692
 acb_dft_cyc (*C function*), 693
 acb_dft_cyc_clear (*C function*), 693
 acb_dft_cyc_init (*C function*), 693
 acb_dft_cyc_precomp (*C function*), 693
 acb_dft_cyc_struct (*C type*), 693
 acb_dft_cyc_t (*C type*), 693
 acb_dft_inverse (*C function*), 690

A

Abs (*C macro*), 906
 acb_abs (*C function*), 643
 acb_acos (*C function*), 648
 acb_acosh (*C function*), 648
 acb_add (*C function*), 643
 acb_add_arb (*C function*), 643
 acb_add_error_arb (*C function*), 640
 acb_add_error_arf (*C function*), 640
 acb_add_error_mag (*C function*), 640
 acb_add_fmpz (*C function*), 643
 acb_add_si (*C function*), 643
 acb_add_ui (*C function*), 643
 acb_addmul (*C function*), 644
 acb_addmul_arb (*C function*), 644
 acb_addmul_fmpz (*C function*), 644
 acb_addmul_si (*C function*), 644
 acb_addmul_ui (*C function*), 644
 acb_agm (*C function*), 651
 acb_agm1 (*C function*), 651
 acb_agm1_cpx (*C function*), 651
 acb_allocated_bytes (*C function*), 639
 acb_approx_dot (*C function*), 645
 acb_arg (*C function*), 643
 acb_asin (*C function*), 648
 acb_asinh (*C function*), 648
 acb_atan (*C function*), 648
 acb_atanh (*C function*), 648
 acb_barnes_g (*C function*), 650
 acb_bernoulli_poly_ui (*C function*), 651
 acb_bits (*C function*), 642
 acb_calc_cauchy_bound (*C function*), 806
 acb_calc_func_t (*C type*), 802
 acb_calc_integrate (*C function*), 803
 acb_calc_integrate_gl_auto_deg (*C function*), 805
 acb_calc_integrate_opt_init (*C function*), 805
 acb_calc_integrate_opt_struct (*C type*), 804
 acb_calc_integrate_opt_t (*C type*), 804

`acb_dft_inverse_precomp` (*C function*), 690
`acb_dft_inverse_rad2` (*C function*), 693
`acb_dft_naive` (*C function*), 692
`acb_dft_naive_clear` (*C function*), 692
`acb_dft_naive_init` (*C function*), 692
`acb_dft_naive_precomp` (*C function*), 692
`acb_dft_naive_struct` (*C type*), 692
`acb_dft_naive_t` (*C type*), 692
`acb_dft_pre_struct` (*C type*), 690
`acb_dft_pre_t` (*C type*), 690
`acb_dft_precomp` (*C function*), 690
`acb_dft_precomp_clear` (*C function*), 690
`acb_dft_precomp_init` (*C function*), 690
`acb_dft_prod_clear` (*C function*), 691
`acb_dft_prod_init` (*C function*), 691
`acb_dft_prod_struct` (*C type*), 691
`acb_dft_prod_t` (*C type*), 691
`acb_dft_rad2` (*C function*), 693
`acb_dft_rad2_clear` (*C function*), 693
`acb_dft_rad2_init` (*C function*), 693
`acb_dft_rad2_precomp` (*C function*), 693
`acb_dft_rad2_struct` (*C type*), 693
`acb_dft_rad2_t` (*C type*), 693
`acb_digamma` (*C function*), 650
`acb_dirichlet_backlund_s` (*C function*), 792
`acb_dirichlet_backlund_s_bound` (*C function*), 792
`acb_dirichlet_backlund_s_gram` (*C function*), 792
`acb_dirichlet_chi` (*C function*), 786
`acb_dirichlet_chi_theta_arb` (*C function*), 787
`acb_dirichlet_chi_vec` (*C function*), 786
`acb_dirichlet_dft` (*C function*), 788
`acb_dirichlet_dft_conrey` (*C function*), 788
`acb_dirichlet_dft_prod` (*C function*), 691
`acb_dirichlet_dft_prod_precomp` (*C function*), 691
`acb_dirichlet_eta` (*C function*), 783
`acb_dirichlet_gauss_sum` (*C function*), 786
`acb_dirichlet_gauss_sum_factor` (*C function*), 786
`acb_dirichlet_gauss_sum_naive` (*C function*), 786
`acb_dirichlet_gauss_sum_order2` (*C function*), 786
`acb_dirichlet_gauss_sum_theta` (*C function*), 786
`acb_dirichlet_gram_point` (*C function*), 791
`acb_dirichlet_hardy_theta` (*C function*), 790
`acb_dirichlet_hardy_theta_series` (*C function*), 791
`acb_dirichlet_hardy_z` (*C function*), 790
`acb_dirichlet_hardy_z_series` (*C function*), 791
`acb_dirichlet_hardy_z_zero` (*C function*), 792
`acb_dirichlet_hardy_z_zeros` (*C function*), 792
`acb_dirichlet_hurwitz` (*C function*), 784
`acb_dirichlet_hurwitz_precomp_bound` (*C function*), 785
`acb_dirichlet_hurwitz_precomp_choose_param` (*C function*), 785
`acb_dirichlet_hurwitz_precomp_clear` (*C function*), 784
`acb_dirichlet_hurwitz_precomp_eval` (*C function*), 785
`acb_dirichlet_hurwitz_precomp_init` (*C function*), 784
`acb_dirichlet_hurwitz_precomp_init_num` (*C function*), 784
`acb_dirichlet_hurwitz_precomp_struct` (*C type*), 784
`acb_dirichlet_hurwitz_precomp_t` (*C type*), 784
`acb_dirichlet_isolate_hardy_z_zero` (*C function*), 791
`acb_dirichlet_jacobi_sum` (*C function*), 787
`acb_dirichlet_jacobi_sum_factor` (*C function*), 786
`acb_dirichlet_jacobi_sum_gauss` (*C function*), 787
`acb_dirichlet_jacobi_sum_naive` (*C function*), 786
`acb_dirichlet_jacobi_sum_ui` (*C function*), 787
`acb_dirichlet_l` (*C function*), 789
`acb_dirichlet_l_euler_product` (*C function*), 789
`acb_dirichlet_l_fmpq` (*C function*), 789
`acb_dirichlet_l_fmpq_afe` (*C function*), 789
`acb_dirichlet_l_hurwitz` (*C function*), 789
`acb_dirichlet_l_jet` (*C function*), 790
`acb_dirichlet_l_series` (*C function*), 790
`acb_dirichlet_l_vec_hurwitz` (*C function*), 790
`acb_dirichlet_lerch_phi` (*C function*), 785
`acb_dirichlet_lerch_phi_direct` (*C function*), 785
`acb_dirichlet_lerch_phi_integral` (*C function*), 785
`acb_dirichlet_pairing` (*C function*), 786
`acb_dirichlet_pairing_char` (*C function*), 786
`acb_dirichlet_platt_hardy_z_zeros` (*C function*), 793
`acb_dirichlet_platt_local_hardy_z_zeros` (*C function*), 793
`acb_dirichlet_platt_multieval` (*C function*), 793
`acb_dirichlet_platt_multieval_threaded` (*C function*), 793
`acb_dirichlet_platt_scaled_lambda` (*C function*), 792
`acb_dirichlet_platt_scaled_lambda_vec` (*C function*), 792
`acb_dirichlet_platt_ws_interpolation` (*C function*), 793
`acb_dirichlet_platt_zeta_zeros` (*C function*), 793
`acb_dirichlet_powsum_sieved` (*C function*), 782

- acb_dirichlet_powsum_smooth (*C function*), 782
 acb_dirichlet_powsum_term (*C function*), 782
 acb_dirichlet_qseries_arb_powers_naive (*C function*), 787
 acb_dirichlet_qseries_arb_powers_smallorder (*C function*), 787
 acb_dirichlet_root (*C function*), 782
 acb_dirichlet_root_number (*C function*), 789
 acb_dirichlet_root_number_theta (*C function*), 789
 acb_dirichlet_roots_clear (*C function*), 782
 acb_dirichlet_roots_init (*C function*), 782
 acb_dirichlet_roots_struct (*C type*), 782
 acb_dirichlet_roots_t (*C type*), 782
 acb_dirichlet_stieltjes (*C function*), 785
 acb_dirichlet_theta_length (*C function*), 787
 acb_dirichlet_turing_method_bound (*C function*), 791
 acb_dirichlet_ui_theta_arb (*C function*), 787
 acb_dirichlet_xi (*C function*), 783
 acb_dirichlet_zeta (*C function*), 783
 acb_dirichlet_zeta_bound (*C function*), 783
 acb_dirichlet_zeta_deriv_bound (*C function*), 783
 acb_dirichlet_zeta_jet (*C function*), 783
 acb_dirichlet_zeta_jet_rs (*C function*), 784
 acb_dirichlet_zeta_nzeros (*C function*), 792
 acb_dirichlet_zeta_nzeros_gram (*C function*), 792
 acb_dirichlet_zeta_rs (*C function*), 784
 acb_dirichlet_zeta_rs_bound (*C function*), 784
 acb_dirichlet_zeta_rs_d_coeffs (*C function*), 783
 acb_dirichlet_zeta_rs_f_coeffs (*C function*), 783
 acb_dirichlet_zeta_rs_r (*C function*), 784
 acb_dirichlet_zeta_zero (*C function*), 792
 acb_dirichlet_zeta_zeros (*C function*), 792
 acb_div (*C function*), 644
 acb_div_arb (*C function*), 644
 acb_div_fmpz (*C function*), 644
 acb_div_onei (*C function*), 643
 acb_div_si (*C function*), 644
 acb_div_ui (*C function*), 644
 acb_dot (*C function*), 645
 acb_dot_fmpz (*C function*), 645
 acb_dot_precise (*C function*), 645
 acb_dot_si (*C function*), 645
 acb_dot_simple (*C function*), 645
 acb_dot_siui (*C function*), 645
 acb_dot_ui (*C function*), 645
 acb_dot_uiui (*C function*), 645
 acb_elliptic_e (*C function*), 742
 acb_elliptic_e_inc (*C function*), 743
 acb_elliptic_f (*C function*), 743
 acb_elliptic_inv_p (*C function*), 746
 acb_elliptic_invariants (*C function*), 746
 acb_elliptic_k (*C function*), 742
 acb_elliptic_k_jet (*C function*), 742
 acb_elliptic_k_series (*C function*), 742
 acb_elliptic_p (*C function*), 745
 acb_elliptic_p_jet (*C function*), 745
 acb_elliptic_p_prime (*C function*), 745
 acb_elliptic_p_series (*C function*), 746
 acb_elliptic_pi (*C function*), 743
 acb_elliptic_pi_inc (*C function*), 743
 acb_elliptic_rc1 (*C function*), 745
 acb_elliptic_rf (*C function*), 744
 acb_elliptic_rg (*C function*), 744
 acb_elliptic_rj (*C function*), 744
 acb_elliptic_rj_carlson (*C function*), 744
 acb_elliptic_rj_integration (*C function*), 744
 acb_elliptic_roots (*C function*), 746
 acb_elliptic_sigma (*C function*), 746
 acb_elliptic_zeta (*C function*), 746
 acb_eq (*C function*), 641
 acb_equal (*C function*), 641
 acb_equal_si (*C function*), 641
 acb_exp (*C function*), 647
 acb_exp_invexp (*C function*), 647
 acb_exp_pi_i (*C function*), 647
 acb_expm1 (*C function*), 647
 acb_fprint (*C function*), 640
 acb_fprintf (*C function*), 640
 acb_fprintfn (*C function*), 640
 acb_gamma (*C function*), 650
 acb_get_abs_lbound_arf (*C function*), 641
 acb_get_abs_ubound_arf (*C function*), 641
 acb_get_imag (*C function*), 643
 acb_get_mag (*C function*), 642
 acb_get_mag_lower (*C function*), 642
 acb_get_mid (*C function*), 640
 acb_get_rad_ubound_arf (*C function*), 641
 acb_get_real (*C function*), 643
 acb_get_unique_fmpz (*C function*), 642
 acb_hurwitz_zeta (*C function*), 651
 acb_hypgeom_0f1 (*C function*), 720
 acb_hypgeom_0f1_asymp (*C function*), 720
 acb_hypgeom_0f1_direct (*C function*), 720
 acb_hypgeom_1f1 (*C function*), 720
 acb_hypgeom_2f1 (*C function*), 729
 acb_hypgeom_2f1_choose (*C function*), 729
 acb_hypgeom_2f1_continuation (*C function*), 728
 acb_hypgeom_2f1_corner (*C function*), 729
 acb_hypgeom_2f1_direct (*C function*), 728
 acb_hypgeom_2f1_series_direct (*C function*), 728
 acb_hypgeom_2f1_transform (*C function*), 729
 acb_hypgeom_2f1_transform_limit (*C function*), 729
 acb_hypgeom_airy (*C function*), 724
 acb_hypgeom_airy_asymp (*C function*), 723
 acb_hypgeom_airy_bound (*C function*), 723
 acb_hypgeom_airy_direct (*C function*), 723
 acb_hypgeom_airy_jet (*C function*), 724

`acb_hypgeom_airy_series` (*C function*), 724
`acb_hypgeom_bessel_i` (*C function*), 722
`acb_hypgeom_bessel_i_0f1` (*C function*), 722
`acb_hypgeom_bessel_i_asymp` (*C function*), 722
`acb_hypgeom_bessel_i_scaled` (*C function*), 722
`acb_hypgeom_bessel_j` (*C function*), 722
`acb_hypgeom_bessel_j_0f1` (*C function*), 721
`acb_hypgeom_bessel_j_asymp` (*C function*), 721
`acb_hypgeom_bessel_j_deriv_bound` (*C function*), 721
`acb_hypgeom_bessel_jy` (*C function*), 722
`acb_hypgeom_bessel_k` (*C function*), 723
`acb_hypgeom_bessel_k_0f1` (*C function*), 723
`acb_hypgeom_bessel_k_0f1_series` (*C function*), 722
`acb_hypgeom_bessel_k_asymp` (*C function*), 722
`acb_hypgeom_bessel_k_scaled` (*C function*), 723
`acb_hypgeom_bessel_y` (*C function*), 722
`acb_hypgeom_beta_lower` (*C function*), 726
`acb_hypgeom_beta_lower_series` (*C function*), 726
`acb_hypgeom_chebyshev_t` (*C function*), 730
`acb_hypgeom_chebyshev_u` (*C function*), 730
`acb_hypgeom_chi` (*C function*), 728
`acb_hypgeom_chi_2f3` (*C function*), 728
`acb_hypgeom_chi_asymp` (*C function*), 728
`acb_hypgeom_chi_series` (*C function*), 728
`acb_hypgeom_ci` (*C function*), 727
`acb_hypgeom_ci_2f3` (*C function*), 727
`acb_hypgeom_ci_asymp` (*C function*), 727
`acb_hypgeom_ci_series` (*C function*), 727
`acb_hypgeom_coulomb` (*C function*), 724
`acb_hypgeom_coulomb_jet` (*C function*), 725
`acb_hypgeom_coulomb_series` (*C function*), 725
`acb_hypgeom_dilog` (*C function*), 732
`acb_hypgeom_dilog_bernoulli` (*C function*), 732
`acb_hypgeom_dilog_bitburst` (*C function*), 732
`acb_hypgeom_dilog_continuation` (*C function*), 732
`acb_hypgeom_dilog_transform` (*C function*), 732
`acb_hypgeom_dilog_zero` (*C function*), 732
`acb_hypgeom_dilog_zero_taylor` (*C function*), 732
`acb_hypgeom_ei` (*C function*), 727
`acb_hypgeom_ei_2f2` (*C function*), 727
`acb_hypgeom_ei_asymp` (*C function*), 727
`acb_hypgeom_ei_series` (*C function*), 727
`acb_hypgeom_erf` (*C function*), 720
`acb_hypgeom_erf_1f1a` (*C function*), 720
`acb_hypgeom_erf_1f1b` (*C function*), 720
`acb_hypgeom_erf_asymp` (*C function*), 720
`acb_hypgeom_erf_propagated_error` (*C function*), 720
`acb_hypgeom_erf_series` (*C function*), 721
`acb_hypgeom_erfc` (*C function*), 721
`acb_hypgeom_erfc_series` (*C function*), 721
`acb_hypgeom_erfi` (*C function*), 721
`acb_hypgeom_erfi_series` (*C function*), 721
`acb_hypgeom_expint` (*C function*), 727
`acb_hypgeom_fresnel` (*C function*), 721
`acb_hypgeom_fresnel_series` (*C function*), 721
`acb_hypgeom_gamma` (*C function*), 717
`acb_hypgeom_gamma_lower` (*C function*), 726
`acb_hypgeom_gamma_lower_series` (*C function*), 726
`acb_hypgeom_gamma_stirling` (*C function*), 716
`acb_hypgeom_gamma_stirling_sum_horner` (*C function*), 716
`acb_hypgeom_gamma_stirling_sum_improved` (*C function*), 716
`acb_hypgeom_gamma_taylor` (*C function*), 716
`acb_hypgeom_gamma_upper` (*C function*), 725
`acb_hypgeom_gamma_upper_1f1a` (*C function*), 725
`acb_hypgeom_gamma_upper_1f1b` (*C function*), 725
`acb_hypgeom_gamma_upper_asymp` (*C function*), 725
`acb_hypgeom_gamma_upper_series` (*C function*), 726
`acb_hypgeom_gamma_upper_singular` (*C function*), 725
`acb_hypgeom_gegenbauer_c` (*C function*), 730
`acb_hypgeom_hermite_h` (*C function*), 730
`acb_hypgeom_jacobi_p` (*C function*), 730
`acb_hypgeom_laguerre_l` (*C function*), 730
`acb_hypgeom_legendre_p` (*C function*), 730
`acb_hypgeom_legendre_p_uiui_rec` (*C function*), 731
`acb_hypgeom_legendre_q` (*C function*), 731
`acb_hypgeom_lgamma` (*C function*), 717
`acb_hypgeom_li` (*C function*), 728
`acb_hypgeom_li_series` (*C function*), 728
`acb_hypgeom_log_rising_ui` (*C function*), 716
`acb_hypgeom_log_rising_ui_jet` (*C function*), 716
`acb_hypgeom_m` (*C function*), 720
`acb_hypgeom_m_1f1` (*C function*), 720
`acb_hypgeom_m_asymp` (*C function*), 719
`acb_hypgeom_pfq` (*C function*), 719
`acb_hypgeom_pfq_bound_factor` (*C function*), 717
`acb_hypgeom_pfq_choose_n` (*C function*), 717
`acb_hypgeom_pfq_direct` (*C function*), 718
`acb_hypgeom_pfq_series_direct` (*C function*), 718
`acb_hypgeom_pfq_series_sum` (*C function*), 718
`acb_hypgeom_pfq_series_sum_bs` (*C function*), 718
`acb_hypgeom_pfq_series_sum_forward` (*C function*), 718
`acb_hypgeom_pfq_series_sum_rs` (*C function*), 718
`acb_hypgeom_pfq_sum` (*C function*), 717
`acb_hypgeom_pfq_sum_bs` (*C function*), 717
`acb_hypgeom_pfq_sum_bs_invz` (*C function*), 718

`acb_hypgeom_pfq_sum_fme` (*C function*), 717
`acb_hypgeom_pfq_sum_forward` (*C function*), 717
`acb_hypgeom_pfq_sum_invz` (*C function*), 718
`acb_hypgeom_pfq_sum_rs` (*C function*), 717
`acb_hypgeom_rgamma` (*C function*), 717
`acb_hypgeom_rising` (*C function*), 716
`acb_hypgeom_rising_ui` (*C function*), 716
`acb_hypgeom_rising_ui_bs` (*C function*), 716
`acb_hypgeom_rising_ui_forward` (*C function*), 716
`acb_hypgeom_rising_ui_jet` (*C function*), 716
`acb_hypgeom_rising_ui_jet_bs` (*C function*), 716
`acb_hypgeom_rising_ui_jet_powsum` (*C function*), 716
`acb_hypgeom_rising_ui_jet_rs` (*C function*), 716
`acb_hypgeom_rising_ui_rec` (*C function*), 716
`acb_hypgeom_rising_ui_rs` (*C function*), 716
`acb_hypgeom_shi` (*C function*), 727
`acb_hypgeom_shi_series` (*C function*), 728
`acb_hypgeom_si` (*C function*), 727
`acb_hypgeom_si_1f2` (*C function*), 727
`acb_hypgeom_si_asymp` (*C function*), 727
`acb_hypgeom_si_series` (*C function*), 727
`acb_hypgeom_spherical_y` (*C function*), 731
`acb_hypgeom_u` (*C function*), 719
`acb_hypgeom_u_1f1` (*C function*), 719
`acb_hypgeom_u_1f1_series` (*C function*), 719
`acb_hypgeom_u_asymp` (*C function*), 719
`acb_hypgeom_u_use_asymp` (*C function*), 719
`acb_imagref` (*C macro*), 638
`acb_indeterminate` (*C function*), 642
`acb_init` (*C function*), 639
`acb_inv` (*C function*), 644
`acb_is_exact` (*C function*), 641
`acb_is_finite` (*C function*), 641
`acb_is_int` (*C function*), 641
`acb_is_int_2exp_si` (*C function*), 641
`acb_is_one` (*C function*), 641
`acb_is_real` (*C function*), 642
`acb_is_zero` (*C function*), 641
`acb_lambertw` (*C function*), 649
`acb_lambertw_asymp` (*C function*), 649
`acb_lambertw_bound_deriv` (*C function*), 649
`acb_lambertw_check_branch` (*C function*), 649
`acb_lgamma` (*C function*), 650
`acb_log` (*C function*), 647
`acb_log1p` (*C function*), 647
`acb_log_analytic` (*C function*), 647
`acb_log_barnes_g` (*C function*), 650
`acb_log_sin_pi` (*C function*), 650
`acb_mat_add` (*C function*), 708
`acb_mat_add_error_mag` (*C function*), 713
`acb_mat_allocated_bytes` (*C function*), 705
`acb_mat_approx_eig_qr` (*C function*), 713
`acb_mat_approx_inv` (*C function*), 711
`acb_mat_approx_lu` (*C function*), 711
`acb_mat_approx_mul` (*C function*), 708
`acb_mat_approx_solve` (*C function*), 711
`acb_mat_approx_solve_lu_precomp` (*C function*), 711
`acb_mat_approx_solve_tril` (*C function*), 711
`acb_mat_approx_solve_triu` (*C function*), 711
`acb_mat_bound_frobenius_norm` (*C function*), 708
`acb_mat_bound_inf_norm` (*C function*), 708
`acb_mat_charpoly` (*C function*), 712
`acb_mat_clear` (*C function*), 705
`acb_mat_companion` (*C function*), 712
`acb_mat_conjugate` (*C function*), 707
`acb_mat_conjugate_transpose` (*C function*), 707
`acb_mat_contains` (*C function*), 706
`acb_mat_contains_fmpq_mat` (*C function*), 706
`acb_mat_contains_fmpz_mat` (*C function*), 706
`acb_mat_det` (*C function*), 711
`acb_mat_det_lu` (*C function*), 711
`acb_mat_det_precond` (*C function*), 711
`acb_mat_dft` (*C function*), 707
`acb_mat_diag_prod` (*C function*), 712
`acb_mat_eig_enclosure_rump` (*C function*), 714
`acb_mat_eig_global_enclosure` (*C function*), 713
`acb_mat_eig_multiple` (*C function*), 715
`acb_mat_eig_multiple_rump` (*C function*), 715
`acb_mat_eig_simple` (*C function*), 714
`acb_mat_eig_simple_rump` (*C function*), 714
`acb_mat_eig_simple_vdhoeven_mourrain` (*C function*), 714
`acb_mat_entry` (*C macro*), 704
`acb_mat_eq` (*C function*), 706
`acb_mat_equal` (*C function*), 706
`acb_mat_exp` (*C function*), 712
`acb_mat_exp_taylor_sum` (*C function*), 712
`acb_mat_fprintfd` (*C function*), 706
`acb_mat_frobenius_norm` (*C function*), 708
`acb_mat_get_imag` (*C function*), 705
`acb_mat_get_mid` (*C function*), 713
`acb_mat_get_real` (*C function*), 705
`acb_mat_indeterminate` (*C function*), 707
`acb_mat_init` (*C function*), 705
`acb_mat_inv` (*C function*), 711
`acb_mat_is_diag` (*C function*), 707
`acb_mat_is_empty` (*C function*), 706
`acb_mat_is_exact` (*C function*), 706
`acb_mat_is_finite` (*C function*), 706
`acb_mat_is_real` (*C function*), 706
`acb_mat_is_square` (*C function*), 706
`acb_mat_is_tril` (*C function*), 707
`acb_mat_is_triu` (*C function*), 707
`acb_mat_is_zero` (*C function*), 706
`acb_mat_lu` (*C function*), 710
`acb_mat_lu_classical` (*C function*), 710
`acb_mat_lu_recursive` (*C function*), 710
`acb_mat_mul` (*C function*), 708
`acb_mat_mul_classical` (*C function*), 708

`acb_mat_mul_entrywise` (*C function*), 708
`acb_mat_mul_reorder` (*C function*), 708
`acb_mat_mul_threaded` (*C function*), 708
`acb_mat_ncols` (*C macro*), 705
`acb_mat_ne` (*C function*), 706
`acb_mat_neg` (*C function*), 708
`acb_mat_nrows` (*C macro*), 704
`acb_mat_one` (*C function*), 707
`acb_mat_onei` (*C function*), 707
`acb_mat_ones` (*C function*), 707
`acb_mat_overlaps` (*C function*), 706
`acb_mat_pow_ui` (*C function*), 708
`acb_mat_printd` (*C function*), 706
`acb_mat_randtest` (*C function*), 706
`acb_mat_randtest_eig` (*C function*), 706
`acb_mat_scalar_addmul_acb` (*C function*), 709
`acb_mat_scalar_addmul_arb` (*C function*), 709
`acb_mat_scalar_addmul_fmpz` (*C function*), 709
`acb_mat_scalar_addmul_si` (*C function*), 709
`acb_mat_scalar_div_acb` (*C function*), 709
`acb_mat_scalar_div_arb` (*C function*), 709
`acb_mat_scalar_div_fmpz` (*C function*), 709
`acb_mat_scalar_div_si` (*C function*), 709
`acb_mat_scalar_mul_2exp_si` (*C function*), 709
`acb_mat_scalar_mul_acb` (*C function*), 709
`acb_mat_scalar_mul_arb` (*C function*), 709
`acb_mat_scalar_mul_fmpz` (*C function*), 709
`acb_mat_scalar_mul_si` (*C function*), 709
`acb_mat_set` (*C function*), 705
`acb_mat_set_arb_mat` (*C function*), 705
`acb_mat_set_fmpq_mat` (*C function*), 705
`acb_mat_set_fmpz_mat` (*C function*), 705
`acb_mat_set_real_imag` (*C function*), 705
`acb_mat_set_round_arb_mat` (*C function*), 705
`acb_mat_set_round_fmpz_mat` (*C function*), 705
`acb_mat_solve` (*C function*), 710
`acb_mat_solve_lu` (*C function*), 710
`acb_mat_solve_lu_precomp` (*C function*), 710
`acb_mat_solve_precond` (*C function*), 710
`acb_mat_solve_tril` (*C function*), 710
`acb_mat_solve_tril_classical` (*C function*), 710
`acb_mat_solve_tril_recursive` (*C function*), 710
`acb_mat_solve_triu` (*C function*), 710
`acb_mat_solve_triu_classical` (*C function*), 710
`acb_mat_solve_triu_recursive` (*C function*), 710
`acb_mat_sqr` (*C function*), 708
`acb_mat_sqr_classical` (*C function*), 708
`acb_mat_struct` (*C type*), 704
`acb_mat_sub` (*C function*), 708
`acb_mat_t` (*C type*), 704
`acb_mat_trace` (*C function*), 712
`acb_mat_transpose` (*C function*), 707
`acb_mat_vector_mul_col` (*C function*), 709
`acb_mat_vector_mul_row` (*C function*), 709
`acb_mat_window_clear` (*C function*), 705
`acb_mat_window_init` (*C function*), 705
`acb_mat_zero` (*C function*), 707
`acb_modular_addseq_eta` (*C function*), 752
`acb_modular_addseq_theta` (*C function*), 750
`acb_modular_delta` (*C function*), 752
`acb_modular_eisenstein` (*C function*), 752
`acb_modular_elliptic_e` (*C function*), 753
`acb_modular_elliptic_k` (*C function*), 753
`acb_modular_elliptic_k_cpx` (*C function*), 753
`acb_modular_elliptic_p` (*C function*), 753
`acb_modular_elliptic_p_zpx` (*C function*), 753
`acb_modular_epsilon_arg` (*C function*), 752
`acb_modular_eta` (*C function*), 752
`acb_modular_eta_sum` (*C function*), 752
`acb_modular_fill_addseq` (*C function*), 748
`acb_modular_fundamental_domain_approx` (*C function*), 748
`acb_modular_fundamental_domain_approx_arf` (*C function*), 748
`acb_modular_fundamental_domain_approx_d` (*C function*), 748
`acb_modular_hilbert_class_poly` (*C function*), 753
`acb_modular_is_in_fundamental_domain` (*C function*), 748
`acb_modular_j` (*C function*), 752
`acb_modular_lambda` (*C function*), 752
`acb_modular_theta` (*C function*), 751
`acb_modular_theta_const_sum` (*C function*), 751
`acb_modular_theta_const_sum_basecase` (*C function*), 751
`acb_modular_theta_const_sum_rs` (*C function*), 751
`acb_modular_theta_jet` (*C function*), 751
`acb_modular_theta_jet_notransform` (*C function*), 751
`acb_modular_theta_notransform` (*C function*), 751
`acb_modular_theta_series` (*C function*), 751
`acb_modular_theta_sum` (*C function*), 750
`acb_modular_theta_transform` (*C function*), 749
`acb_modular_transform` (*C function*), 748
`acb_mul` (*C function*), 644
`acb_mul_2exp_fmpz` (*C function*), 644
`acb_mul_2exp_si` (*C function*), 644
`acb_mul_arb` (*C function*), 644
`acb_mul_fmpz` (*C function*), 644
`acb_mul_i_pow_si` (*C function*), 644
`acb_mul_onei` (*C function*), 643
`acb_mul_si` (*C function*), 644
`acb_mul_ui` (*C function*), 644
`acb_ne` (*C function*), 641
`acb_neg` (*C function*), 643
`acb_neg_round` (*C function*), 643
`acb_one` (*C function*), 639
`acb_onei` (*C function*), 639
`acb_overlaps` (*C function*), 641

`acb_poly_add` (*C function*), 674
`acb_poly_add_series` (*C function*), 674
`acb_poly_add_si` (*C function*), 674
`acb_poly_agm1_series` (*C function*), 686
`acb_poly_allocated_bytes` (*C function*), 671
`acb_poly_atan_series` (*C function*), 681
`acb_poly_binomial_transform` (*C function*), 679
`acb_poly_binomial_transform_basecase` (*C function*), 679
`acb_poly_binomial_transform_convolution` (*C function*), 679
`acb_poly_borel_transform` (*C function*), 679
`acb_poly_clear` (*C function*), 671
`acb_poly_compose` (*C function*), 676
`acb_poly_compose_series` (*C function*), 676
`acb_poly_contains` (*C function*), 673
`acb_poly_contains_fmpq_poly` (*C function*), 673
`acb_poly_contains_fmpz_poly` (*C function*), 673
`acb_poly_cos_pi_series` (*C function*), 682
`acb_poly_cos_series` (*C function*), 682
`acb_poly_cosh_series` (*C function*), 683
`acb_poly_cot_pi_series` (*C function*), 682
`acb_poly_degree` (*C function*), 671
`acb_poly_derivative` (*C function*), 679
`acb_poly_digamma_series` (*C function*), 683
`acb_poly_div_series` (*C function*), 676
`acb_poly_divrem` (*C function*), 676
`acb_poly_elliptic_k_series` (*C function*), 686
`acb_poly_elliptic_p_series` (*C function*), 686
`acb_poly_equal` (*C function*), 673
`acb_poly_erf_series` (*C function*), 685
`acb_poly_evaluate` (*C function*), 677
`acb_poly_evaluate2` (*C function*), 677
`acb_poly_evaluate2_horner` (*C function*), 677
`acb_poly_evaluate2_rectangular` (*C function*), 677
`acb_poly_evaluate_horner` (*C function*), 677
`acb_poly_evaluate_rectangular` (*C function*), 677
`acb_poly_evaluate_vec_fast` (*C function*), 678
`acb_poly_evaluate_vec_iter` (*C function*), 678
`acb_poly_exp_pi_i_series` (*C function*), 681
`acb_poly_exp_series` (*C function*), 681
`acb_poly_exp_series_basecase` (*C function*), 681
`acb_poly_find_roots` (*C function*), 686
`acb_poly_fit_length` (*C function*), 671
`acb_poly_fprintf` (*C function*), 673
`acb_poly_gamma_series` (*C function*), 683
`acb_poly_get_coeff_acb` (*C function*), 672
`acb_poly_get_coeff_ptr` (*C macro*), 672
`acb_poly_get_unique_fmpz_poly` (*C function*), 673
`acb_poly_graeffe_transform` (*C function*), 679
`acb_poly_init` (*C function*), 671
`acb_poly_integral` (*C function*), 679
`acb_poly_interpolate_barycentric` (*C function*), 678
`acb_poly_interpolate_fast` (*C function*), 678
`acb_poly_interpolate_newton` (*C function*), 678
`acb_poly_inv_borel_transform` (*C function*), 679
`acb_poly_inv_series` (*C function*), 676
`acb_poly_is_one` (*C function*), 672
`acb_poly_is_real` (*C function*), 673
`acb_poly_is_x` (*C function*), 672
`acb_poly_is_zero` (*C function*), 671
`acb_poly_lambertw_series` (*C function*), 683
`acb_poly_length` (*C function*), 671
`acb_poly_lgamma_series` (*C function*), 683
`acb_poly_log1p_series` (*C function*), 681
`acb_poly_log_series` (*C function*), 681
`acb_poly_majorant` (*C function*), 674
`acb_poly_mul` (*C function*), 675
`acb_poly_mullow` (*C function*), 675
`acb_poly_mullow_classical` (*C function*), 675
`acb_poly_mullow_transpose` (*C function*), 675
`acb_poly_mullow_transpose_gauss` (*C function*), 675
`acb_poly_mulmid` (*C function*), 675
`acb_poly_mulmid_classical` (*C function*), 675
`acb_poly_mulmid_transpose` (*C function*), 675
`acb_poly_neg` (*C function*), 674
`acb_poly_nth_derivative` (*C function*), 679
`acb_poly_one` (*C function*), 672
`acb_poly_overlaps` (*C function*), 673
`acb_poly_polylog_series` (*C function*), 685
`acb_poly_pow_acb_series` (*C function*), 680
`acb_poly_pow_series` (*C function*), 680
`acb_poly_pow_ui` (*C function*), 680
`acb_poly_pow_ui_trunc_binexp` (*C function*), 680
`acb_poly_printf` (*C function*), 673
`acb_poly_product_roots` (*C function*), 677
`acb_poly_randtest` (*C function*), 673
`acb_poly_revert_series` (*C function*), 676
`acb_poly_rgamma_series` (*C function*), 683
`acb_poly_rising_ui_series` (*C function*), 683
`acb_poly_root_bound_fujiwara` (*C function*), 686
`acb_poly_rsqrtd_series` (*C function*), 681
`acb_poly_scalar_div` (*C function*), 674
`acb_poly_scalar_mul` (*C function*), 674
`acb_poly_scalar_mul_2exp_si` (*C function*), 674
`acb_poly_set` (*C function*), 672
`acb_poly_set2_arb_poly` (*C function*), 673
`acb_poly_set2_fmpq_poly` (*C function*), 673
`acb_poly_set2_fmpz_poly` (*C function*), 673
`acb_poly_set_acb` (*C function*), 673
`acb_poly_set_arb_poly` (*C function*), 673
`acb_poly_set_coeff_acb` (*C function*), 672
`acb_poly_set_coeff_si` (*C function*), 672
`acb_poly_set_fmpq_poly` (*C function*), 673
`acb_poly_set_fmpz_poly` (*C function*), 673
`acb_poly_set_round` (*C function*), 672
`acb_poly_set_si` (*C function*), 673

`acb_poly_set_trunc` (*C function*), 672
`acb_poly_set_trunc_round` (*C function*), 672
`acb_poly_shift_left` (*C function*), 672
`acb_poly_shift_right` (*C function*), 672
`acb_poly_sin_cos_pi_series` (*C function*), 682
`acb_poly_sin_cos_series` (*C function*), 681
`acb_poly_sin_pi_series` (*C function*), 682
`acb_poly_sin_series` (*C function*), 682
`acb_poly_sinc_pi_series` (*C function*), 683
`acb_poly_sinc_series` (*C function*), 683
`acb_poly_sinh_cosh_series` (*C function*), 682
`acb_poly_sinh_cosh_series_basecase` (*C function*), 682
`acb_poly_sinh_cosh_series_exponential` (*C function*), 682
`acb_poly_sinh_series` (*C function*), 682
`acb_poly_sqrt_series` (*C function*), 680
`acb_poly_struct` (*C type*), 671
`acb_poly_sub` (*C function*), 674
`acb_poly_sub_series` (*C function*), 674
`acb_poly_swap` (*C function*), 671
`acb_poly_t` (*C type*), 671
`acb_poly_tan_series` (*C function*), 682
`acb_poly_taylor_shift` (*C function*), 676
`acb_poly_truncate` (*C function*), 672
`acb_poly_validate_real_roots` (*C function*), 687
`acb_poly_valuation` (*C function*), 672
`acb_poly_zero` (*C function*), 672
`acb_poly_zeta_series` (*C function*), 685
`acb_polygamma` (*C function*), 650
`acb_polylog` (*C function*), 651
`acb_polylog_si` (*C function*), 651
`acb_pow` (*C function*), 646
`acb_pow_analytic` (*C function*), 646
`acb_pow_arb` (*C function*), 646
`acb_pow_fmpz` (*C function*), 646
`acb_pow_si` (*C function*), 646
`acb_pow_ui` (*C function*), 646
`acb_print` (*C function*), 640
`acb_printd` (*C function*), 640
`acb_printn` (*C function*), 640
`acb_ptr` (*C type*), 638
`acb_quadratic_roots_fmpz` (*C function*), 646
`acb_randtest` (*C function*), 640
`acb_randtest_param` (*C function*), 640
`acb_randtest_precise` (*C function*), 640
`acb_randtest_special` (*C function*), 640
`acb_real_abs` (*C function*), 652
`acb_real_ceil` (*C function*), 652
`acb_real_floor` (*C function*), 652
`acb_real_heaviside` (*C function*), 652
`acb_real_max` (*C function*), 652
`acb_real_min` (*C function*), 652
`acb_real_sgn` (*C function*), 652
`acb_real_sqrtpos` (*C function*), 652
`acb_realref` (*C macro*), 638
`acb_rel_accuracy_bits` (*C function*), 642
`acb_rel_error_bits` (*C function*), 642
`acb_rel_one_accuracy_bits` (*C function*), 642
`acb_rgamma` (*C function*), 650
`acb_rising` (*C function*), 649
`acb_rising2_ui` (*C function*), 649
`acb_rising_ui` (*C function*), 649
`acb_rising_ui_get_mag` (*C function*), 649
`acb_root_ui` (*C function*), 646
`acb_rsqr` (*C function*), 646
`acb_rsqr_analytic` (*C function*), 646
`acb_sec` (*C function*), 647
`acb_sech` (*C function*), 648
`acb_set` (*C function*), 639
`acb_set_arb` (*C function*), 639
`acb_set_arb_arb` (*C function*), 639
`acb_set_d` (*C function*), 639
`acb_set_d_d` (*C function*), 639
`acb_set_fmpq` (*C function*), 639
`acb_set_fmpz` (*C function*), 639
`acb_set_fmpz_fmpz` (*C function*), 639
`acb_set_round` (*C function*), 639
`acb_set_round_arb` (*C function*), 640
`acb_set_round_fmpz` (*C function*), 639
`acb_set_si` (*C function*), 639
`acb_set_si_si` (*C function*), 639
`acb_set_ui` (*C function*), 639
`acb_sgn` (*C function*), 643
`acb_siegel_cho_yinv` (*C function*), 758
`acb_siegel_cocycle` (*C function*), 758
`acb_siegel_is_reduced` (*C function*), 758
`acb_siegel_kappa` (*C function*), 758
`acb_siegel_kappa2` (*C function*), 759
`acb_siegel_randtest` (*C function*), 759
`acb_siegel_randtest_compact` (*C function*), 759
`acb_siegel_randtest_reduced` (*C function*), 759
`acb_siegel_randtest_vec` (*C function*), 759
`acb_siegel_randtest_vec_reduced` (*C function*), 759
`acb_siegel_reduce` (*C function*), 758
`acb_siegel_transform` (*C function*), 758
`acb_siegel_transform_cocycle_inv` (*C function*), 758
`acb_sin` (*C function*), 647
`acb_sin_cos` (*C function*), 647
`acb_sin_cos_pi` (*C function*), 647
`acb_sin_pi` (*C function*), 647
`acb_sinc` (*C function*), 648
`acb_sinc_pi` (*C function*), 648
`acb_sinh` (*C function*), 648
`acb_sinh_cosh` (*C function*), 648
`acb_sqr` (*C function*), 644
`acb_sqrt` (*C function*), 646
`acb_sqrt_analytic` (*C function*), 646
`acb_sqrts` (*C function*), 646
`acb_srcptr` (*C type*), 638
`acb_struct` (*C type*), 638
`acb_sub` (*C function*), 643
`acb_sub_arb` (*C function*), 643

`acb_sub_fmpz` (*C function*), 643
`acb_sub_si` (*C function*), 643
`acb_sub_ui` (*C function*), 643
`acb_submul` (*C function*), 644
`acb_submul_arb` (*C function*), 644
`acb_submul_fmpz` (*C function*), 644
`acb_submul_si` (*C function*), 644
`acb_submul_ui` (*C function*), 644
`acb_swap` (*C function*), 640
`acb_t` (*C type*), 638
`acb_tan` (*C function*), 647
`acb_tan_pi` (*C function*), 647
`acb_tanh` (*C function*), 648
`acb_theta_agm_mul` (*C function*), 767
`acb_theta_agm_mul_tight` (*C function*), 767
`acb_theta_agm_sqrt` (*C function*), 767
`acb_theta_all` (*C function*), 754
`acb_theta_char_bit` (*C function*), 759
`acb_theta_char_dot` (*C function*), 759
`acb_theta_char_dot_slong` (*C function*), 759
`acb_theta_char_get_acb` (*C function*), 759
`acb_theta_char_get_arb` (*C function*), 759
`acb_theta_char_is_even` (*C function*), 759
`acb_theta_char_set_slong_vec` (*C function*), 754
`acb_theta_char_shuffle` (*C function*), 759
`acb_theta_char_table` (*C function*), 759
`acb_theta_ctx_exp_inv` (*C function*), 764
`acb_theta_ctx_sqr_inv` (*C function*), 764
`acb_theta_ctx_tau_clear` (*C function*), 763
`acb_theta_ctx_tau_dupl` (*C function*), 764
`acb_theta_ctx_tau_init` (*C function*), 763
`acb_theta_ctx_tau_overlaps` (*C function*), 764
`acb_theta_ctx_tau_set` (*C function*), 764
`acb_theta_ctx_tau_struct` (*C type*), 763
`acb_theta_ctx_tau_t` (*C type*), 763
`acb_theta_ctx_z_add_real` (*C function*), 764
`acb_theta_ctx_z_clear` (*C function*), 764
`acb_theta_ctx_z_common_v` (*C function*), 764
`acb_theta_ctx_z_dupl` (*C function*), 764
`acb_theta_ctx_z_init` (*C function*), 764
`acb_theta_ctx_z_overlaps` (*C function*), 764
`acb_theta_ctx_z_set` (*C function*), 764
`acb_theta_ctx_z_struct` (*C type*), 763
`acb_theta_ctx_z_t` (*C type*), 763
`acb_theta_ctx_z_vec_clear` (*C function*), 764
`acb_theta_ctx_z_vec_init` (*C function*), 764
`acb_theta_eld_border` (*C function*), 762
`acb_theta_eld_box` (*C function*), 761
`acb_theta_eld_clear` (*C function*), 761
`acb_theta_eld_contains` (*C function*), 762
`acb_theta_eld_distances` (*C function*), 762
`acb_theta_eld_init` (*C function*), 761
`acb_theta_eld_nb_border` (*C function*), 762
`acb_theta_eld_nb_pts` (*C function*), 761
`acb_theta_eld_points` (*C function*), 761
`acb_theta_eld_print` (*C function*), 762
`acb_theta_eld_set` (*C function*), 761
`acb_theta_eld_struct` (*C type*), 761
`acb_theta_eld_t` (*C type*), 761
`acb_theta_g2_character` (*C function*), 775
`acb_theta_g2_chi35` (*C function*), 775
`acb_theta_g2_chi3_6` (*C function*), 775
`acb_theta_g2_chi5` (*C function*), 775
`acb_theta_g2_covariants` (*C function*), 775
`acb_theta_g2_detk_symj` (*C function*), 774
`acb_theta_g2_even_weight` (*C function*), 775
`acb_theta_g2_sextic_chi5` (*C function*), 775
`acb_theta_g2_transvectant` (*C function*), 774
`acb_theta_jet` (*C function*), 754
`acb_theta_jet_compose` (*C function*), 760
`acb_theta_jet_exp_pi_i` (*C function*), 760
`acb_theta_jet_exp_qf` (*C function*), 760
`acb_theta_jet_index` (*C function*), 760
`acb_theta_jet_mul` (*C function*), 760
`acb_theta_jet_nb` (*C function*), 760
`acb_theta_jet_notransform` (*C function*), 773
`acb_theta_jet_total_order` (*C function*), 760
`acb_theta_jet_tuples` (*C function*), 760
`acb_theta_one` (*C function*), 754
`acb_theta_ql_exact` (*C function*), 771
`acb_theta_ql_jet` (*C function*), 772
`acb_theta_ql_jet_error` (*C function*), 771
`acb_theta_ql_jet_fd` (*C function*), 772
`acb_theta_ql_local_bound` (*C function*), 771
`acb_theta_ql_lower_dim` (*C function*), 769
`acb_theta_ql_nb_steps` (*C function*), 769
`acb_theta_ql_recombine` (*C function*), 769
`acb_theta_ql_setup` (*C function*), 770
`acb_theta_reduce_tau` (*C function*), 773
`acb_theta_reduce_z` (*C function*), 773
`acb_theta_sum` (*C function*), 766
`acb_theta_sum_addprec` (*C function*), 763
`acb_theta_sum_jet` (*C function*), 766
`acb_theta_sum_jet_radius` (*C function*), 762
`acb_theta_sum_radius` (*C function*), 762
`acb_theta_sum_sqr_pow` (*C function*), 766
`acb_theta_sum_term` (*C function*), 763
`acb_theta_sum_work` (*C function*), 766
`acb_theta_sum_worker_t` (*C type*), 765
`acb_theta_sum_worker_t.worker` (*C function*), 765
`acb_trim` (*C function*), 642
`acb_union` (*C function*), 641
`acb_unit_root` (*C function*), 646
`acb_urandom` (*C function*), 641
`acb_zero` (*C function*), 639
`acb_zeta` (*C function*), 651
`acf_add` (*C function*), 612
`acf_allocated_bytes` (*C function*), 611
`acf_approx_div` (*C function*), 612
`acf_approx_dot` (*C function*), 612
`acf_approx_inv` (*C function*), 612
`acf_approx_sqrt` (*C function*), 612
`acf_clear` (*C function*), 611
`acf_equal` (*C function*), 612

`acf_imag_ptr` (*C function*), 612
`acf_imagref` (*C macro*), 611
`acf_init` (*C function*), 611
`acf_mul` (*C function*), 612
`acf_ptr` (*C type*), 611
`acf_real_ptr` (*C function*), 612
`acf_realref` (*C macro*), 611
`acf_set` (*C function*), 612
`acf_srcptr` (*C type*), 611
`acf_struct` (*C type*), 611
`acf_sub` (*C function*), 612
`acf_swap` (*C function*), 611
`acf_t` (*C type*), 611
`Acos` (*C macro*), 907
`Acosh` (*C macro*), 908
`Acot` (*C macro*), 908
`Acoth` (*C macro*), 908
`Acsc` (*C macro*), 908
`Acsch` (*C macro*), 908
`Add` (*C macro*), 900
`add_ssaaaa` (*C macro*), 278
`add_sssaaaaaa` (*C macro*), 278
`AGM` (*C macro*), 912
`AGMSequence` (*C macro*), 912
`AiryAi` (*C macro*), 910
`AiryAiZero` (*C macro*), 910
`AiryBi` (*C macro*), 910
`AiryBiZero` (*C macro*), 910
`AlgebraicNumbers` (*C macro*), 901
`AlgebraicNumberSerialized` (*C macro*), 900
`All` (*C macro*), 898
`AnalyticContinuation` (*C macro*), 904
`And` (*C macro*), 897
`AngleBrackets` (*C macro*), 913
`Approximation` (*C macro*), 900
`aprcl_config` (*C type*), 287
`aprcl_config_gauss_clear` (*C function*), 287
`aprcl_config_gauss_init` (*C function*), 287
`aprcl_config_gauss_init_min_R` (*C function*), 287
`aprcl_config_jacobi_clear` (*C function*), 288
`aprcl_config_jacobi_init` (*C function*), 287
`aprcl_is_prime` (*C function*), 287
`aprcl_is_prime_final_division` (*C function*), 287
`aprcl_is_prime_gauss` (*C function*), 287
`aprcl_is_prime_gauss_min_R` (*C function*), 287
`aprcl_is_prime_jacobi` (*C function*), 287
`aprcl_R_value` (*C function*), 287
`arb_abs` (*C function*), 622
`arb_acos` (*C function*), 628
`arb_acosh` (*C function*), 629
`arb_add` (*C function*), 622
`arb_add_arf` (*C function*), 622
`arb_add_error` (*C function*), 617
`arb_add_error_2exp_fmpz` (*C function*), 618
`arb_add_error_2exp_si` (*C function*), 618
`arb_add_error_arf` (*C function*), 617
`arb_add_error_mag` (*C function*), 617
`arb_add_fmpz` (*C function*), 622
`arb_add_fmpz_2exp` (*C function*), 622
`arb_add_si` (*C function*), 622
`arb_add_ui` (*C function*), 622
`arb_addmul` (*C function*), 623
`arb_addmul_arf` (*C function*), 623
`arb_addmul_fmpz` (*C function*), 623
`arb_addmul_si` (*C function*), 623
`arb_addmul_ui` (*C function*), 623
`arb_agm` (*C function*), 633
`arb_allocated_bytes` (*C function*), 614
`arb_approx_dot` (*C function*), 624
`arb_asin` (*C function*), 628
`arb_asinh` (*C function*), 629
`arb_atan` (*C function*), 628
`arb_atan2` (*C function*), 628
`arb_atan_arf` (*C function*), 628
`arb_atan_arf_bb` (*C function*), 635
`arb_atan_arf_newton` (*C function*), 637
`arb_atan_frac_bsplitt` (*C function*), 635
`ARB_ATAN_GAUSS_PRIME_CACHE_NUM` (*C macro*), 636
`arb_atan_gauss_primes_vec_bsplitt` (*C function*), 636
`arb_atan_newton` (*C function*), 637
`arb_atanh` (*C function*), 629
`arb_bell_fmpz` (*C function*), 633
`arb_bell_sum_bsplitt` (*C function*), 633
`arb_bell_sum_taylor` (*C function*), 633
`arb_bell_ui` (*C function*), 633
`arb_bernoulli_fmpz` (*C function*), 632
`arb_bernoulli_poly_ui` (*C function*), 632
`arb_bernoulli_ui` (*C function*), 632
`arb_bernoulli_ui_zeta` (*C function*), 632
`arb_bin_ui` (*C function*), 630
`arb_bin_uiui` (*C function*), 630
`arb_bits` (*C function*), 619
`arb_calc_func_t` (*C type*), 799
`ARB_CALC_IMPRECISE_INPUT` (*C macro*), 799
`arb_calc_isolate_roots` (*C function*), 800
`arb_calc_newton_conv_factor` (*C function*), 801
`arb_calc_newton_step` (*C function*), 801
`ARB_CALC_NO_CONVERGENCE` (*C macro*), 799
`arb_calc_refine_root_bisect` (*C function*), 801
`arb_calc_refine_root_newton` (*C function*), 801
`ARB_CALC_SUCCESS` (*C macro*), 799
`arb_calc_verbose` (*C var*), 800
`arb_can_round_arf` (*C function*), 619
`arb_can_round_mpfr` (*C function*), 620
`arb_ceil` (*C function*), 619
`arb_chebyshev_t2_ui` (*C function*), 633
`arb_chebyshev_t_ui` (*C function*), 633
`arb_chebyshev_u2_ui` (*C function*), 633
`arb_chebyshev_u_ui` (*C function*), 633
`arb_clear` (*C function*), 614
`arb_const_apery` (*C function*), 629
`arb_const_catalan` (*C function*), 629

`arb_const_e` (*C function*), 629
`arb_const_euler` (*C function*), 629
`arb_const_glaisher` (*C function*), 629
`arb_const_khinchin` (*C function*), 629
`arb_const_log10` (*C function*), 629
`arb_const_log2` (*C function*), 629
`arb_const_log_sqrt2pi` (*C function*), 629
`arb_const_pi` (*C function*), 629
`arb_const_reciprocal_fibonacci` (*C function*), 629
`arb_const_sqrt_pi` (*C function*), 629
`arb_contains` (*C function*), 621
`arb_contains_arf` (*C function*), 621
`arb_contains_fmpq` (*C function*), 621
`arb_contains_fmpz` (*C function*), 621
`arb_contains_int` (*C function*), 621
`arb_contains_interior` (*C function*), 621
`arb_contains_mpf` (*C function*), 621
`arb_contains_negative` (*C function*), 621
`arb_contains_nonnegative` (*C function*), 621
`arb_contains_nonpositive` (*C function*), 621
`arb_contains_positive` (*C function*), 621
`arb_contains_si` (*C function*), 621
`arb_contains_zero` (*C function*), 621
`arb_cos` (*C function*), 627
`arb_cos_pi` (*C function*), 627
`arb_cos_pi_fmpq` (*C function*), 627
`arb_cosh` (*C function*), 628
`arb_cot` (*C function*), 627
`arb_cot_pi` (*C function*), 627
`arb_coth` (*C function*), 628
`arb_csc` (*C function*), 627
`arb_csc_pi` (*C function*), 627
`arb_csch` (*C function*), 628
`arb_digamma` (*C function*), 630
`arb_div` (*C function*), 623
`arb_div_2expm1_ui` (*C function*), 624
`arb_div_arf` (*C function*), 623
`arb_div_fmpz` (*C function*), 623
`arb_div_si` (*C function*), 623
`arb_div_ui` (*C function*), 623
`arb_dot` (*C function*), 624
`arb_dot_fmpz` (*C function*), 624
`arb_dot_precise` (*C function*), 624
`arb_dot_si` (*C function*), 624
`arb_dot_simple` (*C function*), 624
`arb_dot_siui` (*C function*), 624
`arb_dot_ui` (*C function*), 624
`arb_dot_uiui` (*C function*), 624
`arb_doublefac_ui` (*C function*), 630
`arb_dump_file` (*C function*), 616
`arb_dump_str` (*C function*), 616
`arb_eq` (*C function*), 621
`arb_equal` (*C function*), 620
`arb_equal_si` (*C function*), 620
`arb_euler_number_fmpz` (*C function*), 633
`arb_euler_number_ui` (*C function*), 633
`arb_exp` (*C function*), 626
`arb_exp_arf` (*C function*), 636
`arb_exp_arf_bb` (*C function*), 634
`arb_exp_arf_generic` (*C function*), 636
`arb_exp_arf_log_reduction` (*C function*), 636
`arb_exp_arf_rs_generic` (*C function*), 635
`arb_exp_invexp` (*C function*), 626
`arb_expm1` (*C function*), 626
`arb_fac_ui` (*C function*), 630
`arb_fib_fmpz` (*C function*), 633
`arb_fib_ui` (*C function*), 633
`arb_floor` (*C function*), 619
`arb_fma` (*C function*), 623
`arb_fma_arf` (*C function*), 623
`arb_fma_fmpz` (*C function*), 623
`arb_fma_si` (*C function*), 623
`arb_fma_ui` (*C function*), 623
`arb_fmpz_div_fmpz` (*C function*), 623
`arb_fmpz_euler_number_ui` (*C function*), 633
`arb_fmpz_euler_number_ui_multi_mod` (*C function*), 633
`arb_fmpz_poly_complex_roots` (*C function*), 689
`arb_fmpz_poly_deflate` (*C function*), 689
`arb_fmpz_poly_deflation` (*C function*), 689
`arb_fmpz_poly_evaluate_acb` (*C function*), 688
`arb_fmpz_poly_evaluate_acb_horner` (*C function*), 688
`arb_fmpz_poly_evaluate_acb_rectangular` (*C function*), 688
`arb_fmpz_poly_evaluate_arb` (*C function*), 688
`arb_fmpz_poly_evaluate_arb_horner` (*C function*), 688
`arb_fmpz_poly_evaluate_arb_rectangular` (*C function*), 688
`arb_fmpz_poly_gauss_period_minpoly` (*C function*), 689
`arb_fprint` (*C function*), 616
`arb_fprintfd` (*C function*), 616
`arb_fprintfn` (*C function*), 616
`arb_fpwrap_cdouble_acos` (*C function*), 809
`arb_fpwrap_cdouble_acosh` (*C function*), 809
`arb_fpwrap_cdouble_agm` (*C function*), 814
`arb_fpwrap_cdouble_airy_ai` (*C function*), 812
`arb_fpwrap_cdouble_airy_ai_prime` (*C function*), 812
`arb_fpwrap_cdouble_airy_bi` (*C function*), 812
`arb_fpwrap_cdouble_airy_bi_prime` (*C function*), 812
`arb_fpwrap_cdouble_asin` (*C function*), 809
`arb_fpwrap_cdouble_asinh` (*C function*), 809
`arb_fpwrap_cdouble_atan` (*C function*), 809
`arb_fpwrap_cdouble_atanh` (*C function*), 809
`arb_fpwrap_cdouble_barnes_g` (*C function*), 810
`arb_fpwrap_cdouble_bessel_i` (*C function*), 812
`arb_fpwrap_cdouble_bessel_j` (*C function*), 812
`arb_fpwrap_cdouble_bessel_k` (*C function*), 812
`arb_fpwrap_cdouble_bessel_k_scaled` (*C function*), 812
`arb_fpwrap_cdouble_bessel_y` (*C function*), 812

`arb_fpwrap_cdouble_beta_lower` (*C function*), 811

`arb_fpwrap_cdouble_cbrt` (*C function*), 808

`arb_fpwrap_cdouble_chebyshev_t` (*C function*), 813

`arb_fpwrap_cdouble_chebyshev_u` (*C function*), 813

`arb_fpwrap_cdouble_cos` (*C function*), 809

`arb_fpwrap_cdouble_cos_integral` (*C function*), 811

`arb_fpwrap_cdouble_cos_pi` (*C function*), 809

`arb_fpwrap_cdouble_cosh_integral` (*C function*), 812

`arb_fpwrap_cdouble_cot` (*C function*), 809

`arb_fpwrap_cdouble_cot_pi` (*C function*), 809

`arb_fpwrap_cdouble_coulomb_f` (*C function*), 812

`arb_fpwrap_cdouble_coulomb_g` (*C function*), 812

`arb_fpwrap_cdouble_coulomb_hneg` (*C function*), 813

`arb_fpwrap_cdouble_coulomb_hpos` (*C function*), 813

`arb_fpwrap_cdouble_csc` (*C function*), 809

`arb_fpwrap_cdouble_dedekind_eta` (*C function*), 815

`arb_fpwrap_cdouble_digamma` (*C function*), 810

`arb_fpwrap_cdouble_dilog` (*C function*), 812

`arb_fpwrap_cdouble_dirichlet_eta` (*C function*), 810

`arb_fpwrap_cdouble_elliptic_e` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_e_inc` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_f` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_inv_p` (*C function*), 815

`arb_fpwrap_cdouble_elliptic_k` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_p` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_p_prime` (*C function*), 815

`arb_fpwrap_cdouble_elliptic_pi` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_pi_inc` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_rf` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_rg` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_rj` (*C function*), 814

`arb_fpwrap_cdouble_elliptic_sigma` (*C function*), 815

`arb_fpwrap_cdouble_elliptic_zeta` (*C function*), 815

`arb_fpwrap_cdouble_erf` (*C function*), 811

`arb_fpwrap_cdouble_erfc` (*C function*), 811

`arb_fpwrap_cdouble_erfi` (*C function*), 811

`arb_fpwrap_cdouble_exp` (*C function*), 808

`arb_fpwrap_cdouble_exp_integral_e` (*C function*), 811

`arb_fpwrap_cdouble_exp_integral_ei` (*C function*), 811

`arb_fpwrap_cdouble_expm1` (*C function*), 808

`arb_fpwrap_cdouble_fresnel_c` (*C function*), 811

`arb_fpwrap_cdouble_fresnel_s` (*C function*), 811

`arb_fpwrap_cdouble_gamma` (*C function*), 810

`arb_fpwrap_cdouble_gamma_lower` (*C function*), 811

`arb_fpwrap_cdouble_gamma_upper` (*C function*), 811

`arb_fpwrap_cdouble_gegenbauer_c` (*C function*), 813

`arb_fpwrap_cdouble_hardy_theta` (*C function*), 811

`arb_fpwrap_cdouble_hardy_z` (*C function*), 811

`arb_fpwrap_cdouble_hermite_h` (*C function*), 813

`arb_fpwrap_cdouble_hurwitz_zeta` (*C function*), 810

`arb_fpwrap_cdouble_hypgeom_0f1` (*C function*), 814

`arb_fpwrap_cdouble_hypgeom_1f1` (*C function*), 814

`arb_fpwrap_cdouble_hypgeom_2f1` (*C function*), 814

`arb_fpwrap_cdouble_hypgeom_pfq` (*C function*), 814

`arb_fpwrap_cdouble_hypgeom_u` (*C function*), 814

`arb_fpwrap_cdouble_jacobi_p` (*C function*), 813

`arb_fpwrap_cdouble_jacobi_theta_1` (*C function*), 815

`arb_fpwrap_cdouble_jacobi_theta_2` (*C function*), 815

`arb_fpwrap_cdouble_jacobi_theta_3` (*C function*), 815

`arb_fpwrap_cdouble_jacobi_theta_4` (*C function*), 815

`arb_fpwrap_cdouble_laguerre_l` (*C function*), 813

`arb_fpwrap_cdouble_lambertw` (*C function*), 809

`arb_fpwrap_cdouble_legendre_p` (*C function*), 813

`arb_fpwrap_cdouble_legendre_q` (*C function*), 813

`arb_fpwrap_cdouble_lerch_phi` (*C function*), 810

`arb_fpwrap_cdouble_lgamma` (*C function*), 810

`arb_fpwrap_cdouble_log` (*C function*), 808

`arb_fpwrap_cdouble_log1p` (*C function*), 808

`arb_fpwrap_cdouble_log_barnes_g` (*C function*), 810
`arb_fpwrap_cdouble_log_integral` (*C function*), 812
`arb_fpwrap_cdouble_modular_delta` (*C function*), 815
`arb_fpwrap_cdouble_modular_j` (*C function*), 815
`arb_fpwrap_cdouble_modular_lambda` (*C function*), 815
`arb_fpwrap_cdouble_polygamma` (*C function*), 810
`arb_fpwrap_cdouble_polylog` (*C function*), 810
`arb_fpwrap_cdouble_pow` (*C function*), 808
`arb_fpwrap_cdouble_rgamma` (*C function*), 810
`arb_fpwrap_cdouble_riemann_xi` (*C function*), 810
`arb_fpwrap_cdouble_rising` (*C function*), 810
`arb_fpwrap_cdouble_rsqr` (*C function*), 808
`arb_fpwrap_cdouble_sec` (*C function*), 809
`arb_fpwrap_cdouble_sin` (*C function*), 808
`arb_fpwrap_cdouble_sin_integral` (*C function*), 811
`arb_fpwrap_cdouble_sin_pi` (*C function*), 809
`arb_fpwrap_cdouble_sinc` (*C function*), 809
`arb_fpwrap_cdouble_sinc_pi` (*C function*), 809
`arb_fpwrap_cdouble_sinh_integral` (*C function*), 811
`arb_fpwrap_cdouble_spherical_y` (*C function*), 813
`arb_fpwrap_cdouble_sqrt` (*C function*), 808
`arb_fpwrap_cdouble_tan` (*C function*), 809
`arb_fpwrap_cdouble_tan_pi` (*C function*), 809
`arb_fpwrap_cdouble_zeta` (*C function*), 810
`arb_fpwrap_cdouble_zeta_zero` (*C function*), 811
`arb_fpwrap_double_acos` (*C function*), 809
`arb_fpwrap_double_acosh` (*C function*), 809
`arb_fpwrap_double_agm` (*C function*), 814
`arb_fpwrap_double_airy_ai` (*C function*), 812
`arb_fpwrap_double_airy_ai_prime` (*C function*), 812
`arb_fpwrap_double_airy_ai_prime_zero` (*C function*), 812
`arb_fpwrap_double_airy_ai_zero` (*C function*), 812
`arb_fpwrap_double_airy_bi` (*C function*), 812
`arb_fpwrap_double_airy_bi_prime` (*C function*), 812
`arb_fpwrap_double_airy_bi_prime_zero` (*C function*), 812
`arb_fpwrap_double_airy_bi_zero` (*C function*), 812
`arb_fpwrap_double_asin` (*C function*), 809
`arb_fpwrap_double_asinh` (*C function*), 809
`arb_fpwrap_double_atan` (*C function*), 809
`arb_fpwrap_double_atan2` (*C function*), 809
`arb_fpwrap_double_atanh` (*C function*), 809
`arb_fpwrap_double_barnes_g` (*C function*), 810
`arb_fpwrap_double_bessel_i` (*C function*), 812
`arb_fpwrap_double_bessel_j` (*C function*), 812
`arb_fpwrap_double_bessel_k` (*C function*), 812
`arb_fpwrap_double_bessel_k_scaled` (*C function*), 812
`arb_fpwrap_double_bessel_y` (*C function*), 812
`arb_fpwrap_double_beta_lower` (*C function*), 811
`arb_fpwrap_double_cbrt` (*C function*), 808
`arb_fpwrap_double_chebyshev_t` (*C function*), 813
`arb_fpwrap_double_chebyshev_u` (*C function*), 813
`arb_fpwrap_double_cos` (*C function*), 809
`arb_fpwrap_double_cos_integral` (*C function*), 811
`arb_fpwrap_double_cos_pi` (*C function*), 809
`arb_fpwrap_double_cosh_integral` (*C function*), 812
`arb_fpwrap_double_cot` (*C function*), 809
`arb_fpwrap_double_cot_pi` (*C function*), 809
`arb_fpwrap_double_coulomb_f` (*C function*), 812
`arb_fpwrap_double_coulomb_g` (*C function*), 812
`arb_fpwrap_double_csc` (*C function*), 809
`arb_fpwrap_double_digamma` (*C function*), 810
`arb_fpwrap_double_dilog` (*C function*), 812
`arb_fpwrap_double_erf` (*C function*), 811
`arb_fpwrap_double_erfc` (*C function*), 811
`arb_fpwrap_double_erfcinv` (*C function*), 811
`arb_fpwrap_double_erfi` (*C function*), 811
`arb_fpwrap_double_erfinv` (*C function*), 811
`arb_fpwrap_double_exp` (*C function*), 808
`arb_fpwrap_double_exp_integral_e` (*C function*), 811
`arb_fpwrap_double_exp_integral_ei` (*C function*), 811
`arb_fpwrap_double_expm1` (*C function*), 808
`arb_fpwrap_double_fresnel_c` (*C function*), 811
`arb_fpwrap_double_fresnel_s` (*C function*), 811
`arb_fpwrap_double_gamma` (*C function*), 810
`arb_fpwrap_double_gamma_lower` (*C function*), 811
`arb_fpwrap_double_gamma_upper` (*C function*), 811
`arb_fpwrap_double_gegenbauer_c` (*C function*), 813
`arb_fpwrap_double_hermite_h` (*C function*), 813
`arb_fpwrap_double_hurwitz_zeta` (*C function*), 810
`arb_fpwrap_double_hypgeom_0f1` (*C function*), 814
`arb_fpwrap_double_hypgeom_1f1` (*C function*), 814
`arb_fpwrap_double_hypgeom_2f1` (*C function*), 814
`arb_fpwrap_double_hypgeom_pfq` (*C function*), 814

`arb_fpwrap_double_hypgeom_u` (*C function*), 814
`arb_fpwrap_double_jacobi_p` (*C function*), 813
`arb_fpwrap_double_laguerre_l` (*C function*), 813
`arb_fpwrap_double_lambertw` (*C function*), 809
`arb_fpwrap_double_legendre_p` (*C function*), 813
`arb_fpwrap_double_legendre_q` (*C function*), 813
`arb_fpwrap_double_legendre_root` (*C function*), 813
`arb_fpwrap_double_lerch_phi` (*C function*), 810
`arb_fpwrap_double_lgamma` (*C function*), 810
`arb_fpwrap_double_log` (*C function*), 808
`arb_fpwrap_double_loglp` (*C function*), 808
`arb_fpwrap_double_log_barnes_g` (*C function*), 810
`arb_fpwrap_double_log_integral` (*C function*), 812
`arb_fpwrap_double_polygamma` (*C function*), 810
`arb_fpwrap_double_polylog` (*C function*), 810
`arb_fpwrap_double_pow` (*C function*), 808
`arb_fpwrap_double_rgamma` (*C function*), 810
`arb_fpwrap_double_rising` (*C function*), 810
`arb_fpwrap_double_rsqr` (*C function*), 808
`arb_fpwrap_double_sec` (*C function*), 809
`arb_fpwrap_double_sin` (*C function*), 808
`arb_fpwrap_double_sin_integral` (*C function*), 811
`arb_fpwrap_double_sin_pi` (*C function*), 809
`arb_fpwrap_double_sinc` (*C function*), 809
`arb_fpwrap_double_sinc_pi` (*C function*), 809
`arb_fpwrap_double_sinh_integral` (*C function*), 811
`arb_fpwrap_double_sqrt` (*C function*), 808
`arb_fpwrap_double_tan` (*C function*), 809
`arb_fpwrap_double_tan_pi` (*C function*), 809
`arb_fpwrap_double_zeta` (*C function*), 810
`arb_gamma` (*C function*), 630
`arb_gamma_fmpq` (*C function*), 630
`arb_gamma_fmpz` (*C function*), 630
`arb_ge` (*C function*), 621
`arb_get_abs_lbound_arf` (*C function*), 618
`arb_get_abs_ubound_arf` (*C function*), 618
`arb_get_fmpz_mid_rad_10exp` (*C function*), 619
`arb_get_interval_arf` (*C function*), 619
`arb_get_interval_fmpz_2exp` (*C function*), 618
`arb_get_interval_mpfr` (*C function*), 619
`arb_get_lbound_arf` (*C function*), 618
`arb_get_mag` (*C function*), 618
`arb_get_mag_lower` (*C function*), 618
`arb_get_mag_lower_nonnegative` (*C function*), 618
`arb_get_mid_arb` (*C function*), 617
`arb_get_rad_arb` (*C function*), 617
`arb_get_rand_fmpq` (*C function*), 617
`arb_get_str` (*C function*), 615
`arb_get_ubound_arf` (*C function*), 618
`arb_get_unique_fmpz` (*C function*), 619
`arb_gt` (*C function*), 621
`arb_hurwitz_zeta` (*C function*), 631
`arb_hypgeom_0f1` (*C function*), 734
`arb_hypgeom_1f1` (*C function*), 734
`arb_hypgeom_1f1_integration` (*C function*), 734
`arb_hypgeom_2f1` (*C function*), 735
`arb_hypgeom_2f1_integration` (*C function*), 735
`arb_hypgeom_airy` (*C function*), 739
`arb_hypgeom_airy_jet` (*C function*), 739
`arb_hypgeom_airy_series` (*C function*), 739
`arb_hypgeom_airy_zero` (*C function*), 739
`arb_hypgeom_bessel_i` (*C function*), 739
`arb_hypgeom_bessel_i_integration` (*C function*), 739
`arb_hypgeom_bessel_i_scaled` (*C function*), 739
`arb_hypgeom_bessel_j` (*C function*), 739
`arb_hypgeom_bessel_jy` (*C function*), 739
`arb_hypgeom_bessel_k` (*C function*), 739
`arb_hypgeom_bessel_k_integration` (*C function*), 739
`arb_hypgeom_bessel_k_scaled` (*C function*), 739
`arb_hypgeom_bessel_y` (*C function*), 739
`arb_hypgeom_beta_lower` (*C function*), 736
`arb_hypgeom_beta_lower_series` (*C function*), 737
`arb_hypgeom_central_bin_ui` (*C function*), 734
`arb_hypgeom_chebyshev_t` (*C function*), 740
`arb_hypgeom_chebyshev_u` (*C function*), 740
`arb_hypgeom_chi` (*C function*), 738
`arb_hypgeom_chi_series` (*C function*), 738
`arb_hypgeom_ci` (*C function*), 738
`arb_hypgeom_ci_series` (*C function*), 738
`arb_hypgeom_coulomb` (*C function*), 740
`arb_hypgeom_coulomb_jet` (*C function*), 740
`arb_hypgeom_coulomb_series` (*C function*), 740
`arb_hypgeom_dilog` (*C function*), 741
`arb_hypgeom_ei` (*C function*), 738
`arb_hypgeom_ei_series` (*C function*), 738
`arb_hypgeom_erf` (*C function*), 735
`arb_hypgeom_erf_series` (*C function*), 735
`arb_hypgeom_erfc` (*C function*), 735
`arb_hypgeom_erfc_series` (*C function*), 735
`arb_hypgeom_erfcinv` (*C function*), 735
`arb_hypgeom_erfi` (*C function*), 735
`arb_hypgeom_erfi_series` (*C function*), 735
`arb_hypgeom_erfinv` (*C function*), 735
`arb_hypgeom_expint` (*C function*), 738
`arb_hypgeom_fresnel` (*C function*), 736
`arb_hypgeom_fresnel_series` (*C function*), 736
`arb_hypgeom_gamma` (*C function*), 734
`arb_hypgeom_gamma_fmpq` (*C function*), 734
`arb_hypgeom_gamma_fmpz` (*C function*), 734
`arb_hypgeom_gamma_lower` (*C function*), 736
`arb_hypgeom_gamma_lower_series` (*C function*), 736
`arb_hypgeom_gamma_stirling` (*C function*), 733

`arb_hypgeom_gamma_stirling_sum_horner` (*C function*), 733
`arb_hypgeom_gamma_stirling_sum_improved` (*C function*), 733
`arb_hypgeom_gamma_taylor` (*C function*), 734
`arb_hypgeom_gamma_upper` (*C function*), 736
`arb_hypgeom_gamma_upper_integration` (*C function*), 736
`arb_hypgeom_gamma_upper_series` (*C function*), 736
`arb_hypgeom_gegenbauer_c` (*C function*), 740
`arb_hypgeom_hermite_h` (*C function*), 740
`arb_hypgeom_infsum` (*C function*), 798
`arb_hypgeom_jacobi_p` (*C function*), 740
`arb_hypgeom_laguerre_l` (*C function*), 740
`arb_hypgeom_legendre_p` (*C function*), 740
`arb_hypgeom_legendre_p_ui` (*C function*), 740
`arb_hypgeom_legendre_p_ui_asymp` (*C function*), 740
`arb_hypgeom_legendre_p_ui_deriv_bound` (*C function*), 740
`arb_hypgeom_legendre_p_ui_one` (*C function*), 740
`arb_hypgeom_legendre_p_ui_rec` (*C function*), 740
`arb_hypgeom_legendre_p_ui_root` (*C function*), 741
`arb_hypgeom_legendre_p_ui_zero` (*C function*), 740
`arb_hypgeom_legendre_q` (*C function*), 740
`arb_hypgeom_lgamma` (*C function*), 734
`arb_hypgeom_li` (*C function*), 738
`arb_hypgeom_li_series` (*C function*), 739
`arb_hypgeom_m` (*C function*), 734
`arb_hypgeom_pfq` (*C function*), 734
`arb_hypgeom_rgamma` (*C function*), 734
`arb_hypgeom_rising` (*C function*), 733
`arb_hypgeom_rising_ui` (*C function*), 733
`arb_hypgeom_rising_ui_bs` (*C function*), 733
`arb_hypgeom_rising_ui_forward` (*C function*), 733
`arb_hypgeom_rising_ui_jet` (*C function*), 733
`arb_hypgeom_rising_ui_jet_bs` (*C function*), 733
`arb_hypgeom_rising_ui_jet_powsum` (*C function*), 733
`arb_hypgeom_rising_ui_jet_rs` (*C function*), 733
`arb_hypgeom_rising_ui_rec` (*C function*), 733
`arb_hypgeom_rising_ui_rs` (*C function*), 733
`arb_hypgeom_shi` (*C function*), 738
`arb_hypgeom_shi_series` (*C function*), 738
`arb_hypgeom_si` (*C function*), 738
`arb_hypgeom_si_series` (*C function*), 738
`arb_hypgeom_sum` (*C function*), 798
`arb_hypgeom_sum_fmpq_arb` (*C function*), 741
`arb_hypgeom_sum_fmpq_arb_forward` (*C function*), 741
`arb_hypgeom_sum_fmpq_arb_rs` (*C function*), 741
`arb_hypgeom_sum_fmpq_imag_arb` (*C function*), 741
`arb_hypgeom_sum_fmpq_imag_arb_bs` (*C function*), 741
`arb_hypgeom_sum_fmpq_imag_arb_forward` (*C function*), 741
`arb_hypgeom_sum_fmpq_imag_arb_rs` (*C function*), 741
`arb_hypgeom_u` (*C function*), 734
`arb_hypgeom_u_integration` (*C function*), 735
`arb_hypot` (*C function*), 625
`arb_indeterminate` (*C function*), 615
`arb_init` (*C function*), 614
`arb_intersection` (*C function*), 618
`arb_inv` (*C function*), 623
`arb_is_exact` (*C function*), 620
`arb_is_finite` (*C function*), 620
`arb_is_int` (*C function*), 620
`arb_is_int_2exp_si` (*C function*), 620
`arb_is_negative` (*C function*), 620
`arb_is_nonnegative` (*C function*), 620
`arb_is_nonpositive` (*C function*), 620
`arb_is_nonzero` (*C function*), 620
`arb_is_one` (*C function*), 620
`arb_is_positive` (*C function*), 620
`arb_is_zero` (*C function*), 620
`arb_lambertw` (*C function*), 630
`arb_le` (*C function*), 621
`arb_lgamma` (*C function*), 630
`arb_load_file` (*C function*), 616
`arb_load_str` (*C function*), 616
`arb_log` (*C function*), 626
`arb_log1p` (*C function*), 626
`arb_log_arf` (*C function*), 626
`arb_log_arf_newton` (*C function*), 636
`arb_log_base_ui` (*C function*), 626
`arb_log_fmpz` (*C function*), 626
`arb_log_hypot` (*C function*), 626
`arb_log_newton` (*C function*), 636
`ARB_LOG_PRIME_CACHE_NUM` (*C macro*), 636
`arb_log_primes_vec_bsplitt` (*C function*), 636
`ARB_LOG_REDUCTION_DEFAULT_MAX_PREC` (*C macro*), 636
`arb_log_ui` (*C function*), 626
`arb_log_ui_from_prev` (*C function*), 626
`arb_lt` (*C function*), 621
`arb_mat_add` (*C function*), 697
`arb_mat_add_error_mag` (*C function*), 703
`arb_mat_allocated_bytes` (*C function*), 694
`arb_mat_approx_inv` (*C function*), 701
`arb_mat_approx_lu` (*C function*), 701
`arb_mat_approx_mul` (*C function*), 698
`arb_mat_approx_solve` (*C function*), 701
`arb_mat_approx_solve_lu_precomp` (*C function*), 701
`arb_mat_approx_solve_tril` (*C function*), 701
`arb_mat_approx_solve_triu` (*C function*), 701

`arb_mat_bound_frobenius_norm` (*C function*), 697

`arb_mat_bound_inf_norm` (*C function*), 697

`arb_mat_charpoly` (*C function*), 702

`arb_mat_cho` (*C function*), 701

`arb_mat_clear` (*C function*), 694

`arb_mat_companion` (*C function*), 702

`arb_mat_contains` (*C function*), 695

`arb_mat_contains_fmpq_mat` (*C function*), 695

`arb_mat_contains_fmpz_mat` (*C function*), 695

`arb_mat_count_is_zero` (*C function*), 703

`arb_mat_count_not_is_zero` (*C function*), 703

`arb_mat_dct` (*C function*), 697

`arb_mat_det` (*C function*), 700

`arb_mat_det_lu` (*C function*), 700

`arb_mat_det_precond` (*C function*), 700

`arb_mat_diag_prod` (*C function*), 703

`arb_mat_entry` (*C macro*), 694

`arb_mat_eq` (*C function*), 695

`arb_mat_equal` (*C function*), 695

`arb_mat_exp` (*C function*), 703

`arb_mat_exp_taylor_sum` (*C function*), 703

`arb_mat_fprintf` (*C function*), 695

`arb_mat_frobenius_norm` (*C function*), 697

`arb_mat_get_mid` (*C function*), 703

`arb_mat_hilbert` (*C function*), 696

`arb_mat_indeterminate` (*C function*), 696

`arb_mat_init` (*C function*), 694

`arb_mat_inv` (*C function*), 700

`arb_mat_inv_cho_precomp` (*C function*), 701

`arb_mat_inv_ldl_precomp` (*C function*), 702

`arb_mat_is_diag` (*C function*), 696

`arb_mat_is_empty` (*C function*), 695

`arb_mat_is_exact` (*C function*), 696

`arb_mat_is_finite` (*C function*), 696

`arb_mat_is_square` (*C function*), 696

`arb_mat_is_tril` (*C function*), 696

`arb_mat_is_triu` (*C function*), 696

`arb_mat_is_zero` (*C function*), 696

`arb_mat_ldl` (*C function*), 702

`arb_mat_lu` (*C function*), 699

`arb_mat_lu_classical` (*C function*), 699

`arb_mat_lu_recursive` (*C function*), 699

`arb_mat_mul` (*C function*), 697

`arb_mat_mul_block` (*C function*), 697

`arb_mat_mul_classical` (*C function*), 697

`arb_mat_mul_entrywise` (*C function*), 698

`arb_mat_mul_threaded` (*C function*), 697

`arb_mat_ncols` (*C macro*), 694

`arb_mat_ne` (*C function*), 695

`arb_mat_neg` (*C function*), 697

`arb_mat_nrows` (*C macro*), 694

`arb_mat_one` (*C function*), 696

`arb_mat_ones` (*C function*), 696

`arb_mat_overlaps` (*C function*), 695

`arb_mat_pascal` (*C function*), 696

`arb_mat_pow_ui` (*C function*), 698

`arb_mat_printd` (*C function*), 695

`arb_mat_randtest` (*C function*), 695

`arb_mat_randtest_cho` (*C function*), 695

`arb_mat_randtest_spd` (*C function*), 695

`arb_mat_scalar_addmul_arb` (*C function*), 698

`arb_mat_scalar_addmul_fmpz` (*C function*), 698

`arb_mat_scalar_addmul_si` (*C function*), 698

`arb_mat_scalar_div_arb` (*C function*), 698

`arb_mat_scalar_div_fmpz` (*C function*), 698

`arb_mat_scalar_div_si` (*C function*), 698

`arb_mat_scalar_mul_2exp_si` (*C function*), 698

`arb_mat_scalar_mul_arb` (*C function*), 698

`arb_mat_scalar_mul_fmpz` (*C function*), 698

`arb_mat_scalar_mul_si` (*C function*), 698

`arb_mat_set` (*C function*), 695

`arb_mat_set_fmpq_mat` (*C function*), 695

`arb_mat_set_fmpz_mat` (*C function*), 695

`arb_mat_set_round_fmpz_mat` (*C function*), 695

`arb_mat_solve` (*C function*), 700

`arb_mat_solve_cho_precomp` (*C function*), 701

`arb_mat_solve_ldl_precomp` (*C function*), 702

`arb_mat_solve_lu` (*C function*), 700

`arb_mat_solve_lu_precomp` (*C function*), 700

`arb_mat_solve_preapprox` (*C function*), 700

`arb_mat_solve_precond` (*C function*), 700

`arb_mat_solve_tril` (*C function*), 699

`arb_mat_solve_tril_classical` (*C function*), 699

`arb_mat_solve_tril_recursive` (*C function*), 699

`arb_mat_solve_triu` (*C function*), 699

`arb_mat_solve_triu_classical` (*C function*), 699

`arb_mat_solve_triu_recursive` (*C function*), 699

`arb_mat_spd_get_fmpz_mat` (*C function*), 704

`arb_mat_spd_inv` (*C function*), 702

`arb_mat_spd_is_lll_reduced` (*C function*), 704

`arb_mat_spd_lll_reduce` (*C function*), 704

`arb_mat_spd_solve` (*C function*), 701

`arb_mat_sqr` (*C function*), 698

`arb_mat_sqr_classical` (*C function*), 698

`arb_mat_stirling` (*C function*), 696

`arb_mat_struct` (*C type*), 694

`arb_mat_sub` (*C function*), 697

`arb_mat_t` (*C type*), 694

`arb_mat_trace` (*C function*), 703

`arb_mat_transpose` (*C function*), 697

`arb_mat_vector_mul_col` (*C function*), 699

`arb_mat_vector_mul_row` (*C function*), 699

`arb_mat_window_clear` (*C function*), 694

`arb_mat_window_init` (*C function*), 694

`arb_mat_zero` (*C function*), 696

`arb_max` (*C function*), 622

`arb_midref` (*C macro*), 613

`arb_min` (*C function*), 622

`arb_minmax` (*C function*), 622

`arb_mul` (*C function*), 622

`arb_mul_2exp_fmpz` (*C function*), 623

`arb_mul_2exp_si` (*C function*), 623
`arb_mul_arf` (*C function*), 622
`arb_mul_fmpz` (*C function*), 623
`arb_mul_si` (*C function*), 622
`arb_mul_ui` (*C function*), 622
`arb_ne` (*C function*), 621
`arb_neg` (*C function*), 622
`arb_neg_inf` (*C function*), 615
`arb_neg_round` (*C function*), 622
`arb_nint` (*C function*), 619
`arb_nonnegative_abs` (*C function*), 622
`arb_nonnegative_part` (*C function*), 618
`arb_one` (*C function*), 615
`arb_overlaps` (*C function*), 621
`arb_partitions_fmpz` (*C function*), 633
`arb_partitions_ui` (*C function*), 633
`arb_poly_acos_series` (*C function*), 666
`arb_poly_add` (*C function*), 657
`arb_poly_add_series` (*C function*), 658
`arb_poly_add_si` (*C function*), 657
`arb_poly_allocated_bytes` (*C function*), 655
`arb_poly_asin_series` (*C function*), 666
`arb_poly_atan_series` (*C function*), 666
`arb_poly_binomial_transform` (*C function*), 664
`arb_poly_binomial_transform_basecase` (*C function*), 664
`arb_poly_binomial_transform_convolution` (*C function*), 664
`arb_poly_borel_transform` (*C function*), 664
`arb_poly_clear` (*C function*), 655
`arb_poly_compose` (*C function*), 660
`arb_poly_compose_series` (*C function*), 660
`arb_poly_contains` (*C function*), 657
`arb_poly_contains_fmpz_poly` (*C function*), 657
`arb_poly_contains_fmpz_poly` (*C function*), 657
`arb_poly_cos_pi_series` (*C function*), 667
`arb_poly_cos_series` (*C function*), 667
`arb_poly_cosh_series` (*C function*), 668
`arb_poly_cot_pi_series` (*C function*), 667
`arb_poly_degree` (*C function*), 655
`arb_poly_derivative` (*C function*), 663
`arb_poly_digamma_series` (*C function*), 668
`arb_poly_div_series` (*C function*), 659
`arb_poly_divrem` (*C function*), 659
`arb_poly_equal` (*C function*), 657
`arb_poly_evaluate` (*C function*), 660
`arb_poly_evaluate2` (*C function*), 661
`arb_poly_evaluate2_acb` (*C function*), 661
`arb_poly_evaluate2_acb_horner` (*C function*), 661
`arb_poly_evaluate2_acb_rectangular` (*C function*), 661
`arb_poly_evaluate2_horner` (*C function*), 661
`arb_poly_evaluate2_rectangular` (*C function*), 661
`arb_poly_evaluate_acb` (*C function*), 661
`arb_poly_evaluate_acb_horner` (*C function*), 661
`arb_poly_evaluate_acb_rectangular` (*C function*), 661
`arb_poly_evaluate_horner` (*C function*), 660
`arb_poly_evaluate_rectangular` (*C function*), 660
`arb_poly_evaluate_vec_fast` (*C function*), 662
`arb_poly_evaluate_vec_iter` (*C function*), 662
`arb_poly_exp_series` (*C function*), 666
`arb_poly_exp_series_basecase` (*C function*), 666
`arb_poly_fit_length` (*C function*), 655
`arb_poly_fprintf` (*C function*), 657
`arb_poly_gamma_series` (*C function*), 668
`arb_poly_get_coeff_arb` (*C function*), 656
`arb_poly_get_coeff_ptr` (*C macro*), 656
`arb_poly_get_unique_fmpz_poly` (*C function*), 657
`arb_poly_graeffe_transform` (*C function*), 664
`arb_poly_init` (*C function*), 655
`arb_poly_integral` (*C function*), 663
`arb_poly_interpolate_barycentric` (*C function*), 663
`arb_poly_interpolate_fast` (*C function*), 663
`arb_poly_interpolate_newton` (*C function*), 663
`arb_poly_inv_borel_transform` (*C function*), 664
`arb_poly_inv_series` (*C function*), 659
`arb_poly_is_one` (*C function*), 655
`arb_poly_is_x` (*C function*), 656
`arb_poly_is_zero` (*C function*), 655
`arb_poly_lambertw_series` (*C function*), 668
`arb_poly_length` (*C function*), 655
`arb_poly_lgamma_series` (*C function*), 668
`arb_poly_log1p_series` (*C function*), 666
`arb_poly_log_series` (*C function*), 666
`arb_poly_majorant` (*C function*), 657
`arb_poly_mul` (*C function*), 659
`arb_poly_mullo` (*C function*), 659
`arb_poly_mullo_block` (*C function*), 659
`arb_poly_mullo_classical` (*C function*), 658
`arb_poly_mulmid` (*C function*), 659
`arb_poly_mulmid_block` (*C function*), 659
`arb_poly_mulmid_classical` (*C function*), 659
`arb_poly_neg` (*C function*), 658
`arb_poly_nth_derivative` (*C function*), 663
`arb_poly_one` (*C function*), 656
`arb_poly_overlaps` (*C function*), 657
`arb_poly_pow_arb_series` (*C function*), 665
`arb_poly_pow_series` (*C function*), 665
`arb_poly_pow_ui` (*C function*), 665
`arb_poly_pow_ui_trunc_binexp` (*C function*), 665
`arb_poly_printf` (*C function*), 657
`arb_poly_product_roots` (*C function*), 662
`arb_poly_product_roots_complex` (*C function*), 662
`arb_poly_randtest` (*C function*), 657
`arb_poly_revert_series` (*C function*), 660

`arb_poly_rgamma_series` (*C function*), 668
`arb_poly_riemann_siegel_theta_series` (*C function*), 669
`arb_poly_riemann_siegel_z_series` (*C function*), 669
`arb_poly_rising_ui_series` (*C function*), 668
`arb_poly_root_bound_fujiwara` (*C function*), 670
`arb_poly_rsqrtd_series` (*C function*), 665
`arb_poly_scalar_div` (*C function*), 658
`arb_poly_scalar_mul` (*C function*), 658
`arb_poly_scalar_mul_2exp_si` (*C function*), 658
`arb_poly_set` (*C function*), 656
`arb_poly_set_coeff_arb` (*C function*), 656
`arb_poly_set_coeff_si` (*C function*), 656
`arb_poly_set_fmpq_poly` (*C function*), 656
`arb_poly_set_fmpz_poly` (*C function*), 656
`arb_poly_set_round` (*C function*), 656
`arb_poly_set_si` (*C function*), 656
`arb_poly_set_trunc` (*C function*), 656
`arb_poly_set_trunc_round` (*C function*), 656
`arb_poly_shift_left` (*C function*), 656
`arb_poly_shift_right` (*C function*), 656
`arb_poly_sin_cos_pi_series` (*C function*), 667
`arb_poly_sin_cos_series` (*C function*), 666
`arb_poly_sin_pi_series` (*C function*), 667
`arb_poly_sin_series` (*C function*), 667
`arb_poly_sinc_pi_series` (*C function*), 668
`arb_poly_sinc_series` (*C function*), 668
`arb_poly_sinh_cosh_series` (*C function*), 667
`arb_poly_sinh_cosh_series_basecase` (*C function*), 667
`arb_poly_sinh_cosh_series_exponential` (*C function*), 667
`arb_poly_sinh_series` (*C function*), 667
`arb_poly_sqrt_series` (*C function*), 665
`arb_poly_struct` (*C type*), 655
`arb_poly_sub` (*C function*), 658
`arb_poly_sub_series` (*C function*), 658
`arb_poly_swinnerton_dyer_ui` (*C function*), 670
`arb_poly_t` (*C type*), 655
`arb_poly_tan_series` (*C function*), 667
`arb_poly_taylor_shift` (*C function*), 660
`arb_poly_truncate` (*C function*), 656
`arb_poly_valuation` (*C function*), 656
`arb_poly_zero` (*C function*), 656
`arb_poly_zeta_series` (*C function*), 669
`arb_polylog` (*C function*), 632
`arb_polylog_si` (*C function*), 632
`arb_pos_inf` (*C function*), 615
`arb_pow` (*C function*), 626
`arb_pow_fmpq` (*C function*), 626
`arb_pow_fmpz` (*C function*), 625
`arb_pow_fmpz_binexp` (*C function*), 625
`arb_pow_ui` (*C function*), 625
`arb_power_sum_vec` (*C function*), 632
`arb_primorial_nth_ui` (*C function*), 633
`arb_primorial_ui` (*C function*), 633
`arb_print` (*C function*), 616
`arb_printd` (*C function*), 616
`arb_printn` (*C function*), 616
`arb_ptr` (*C type*), 613
`arb_radref` (*C macro*), 613
`arb_randtest` (*C function*), 617
`arb_randtest_exact` (*C function*), 617
`arb_randtest_positive` (*C function*), 617
`arb_randtest_precise` (*C function*), 617
`arb_randtest_special` (*C function*), 617
`arb_randtest_wide` (*C function*), 617
`arb_rel_accuracy_bits` (*C function*), 619
`arb_rel_error_bits` (*C function*), 619
`arb_rel_one_accuracy_bits` (*C function*), 619
`arb_rgamma` (*C function*), 630
`arb_rising` (*C function*), 630
`arb_rising2_ui` (*C function*), 630
`arb_rising_fmpq_ui` (*C function*), 630
`arb_rising_ui` (*C function*), 630
`arb_root` (*C function*), 625
`arb_root_ui` (*C function*), 625
`arb_rsqrtd` (*C function*), 625
`arb_rsqrtd_ui` (*C function*), 625
`arb_sec` (*C function*), 627
`arb_sech` (*C function*), 628
`arb_set` (*C function*), 614
`arb_set_arf` (*C function*), 614
`arb_set_d` (*C function*), 614
`arb_set_fmpq` (*C function*), 615
`arb_set_fmpz` (*C function*), 614
`arb_set_fmpz_2exp` (*C function*), 614
`arb_set_interval_arf` (*C function*), 618
`arb_set_interval_mag` (*C function*), 618
`arb_set_interval_mpfr` (*C function*), 619
`arb_set_interval_neg_pos_mag` (*C function*), 619
`arb_set_round` (*C function*), 614
`arb_set_round_fmpz` (*C function*), 615
`arb_set_round_fmpz_2exp` (*C function*), 615
`arb_set_si` (*C function*), 614
`arb_set_str` (*C function*), 615
`arb_set_ui` (*C function*), 614
`arb_sgn` (*C function*), 622
`arb_sgn_nonzero` (*C function*), 622
`arb_si_pow_ui` (*C function*), 625
`arb_sin` (*C function*), 627
`arb_sin_cos` (*C function*), 627
`arb_sin_cos_arf_atan_reduction` (*C function*), 637
`arb_sin_cos_arf_bb` (*C function*), 636
`arb_sin_cos_arf_generic` (*C function*), 635
`arb_sin_cos_generic` (*C function*), 636
`arb_sin_cos_pi` (*C function*), 627
`arb_sin_cos_pi_fmpq` (*C function*), 627
`arb_sin_cos_wide` (*C function*), 636
`arb_sin_pi` (*C function*), 627
`arb_sin_pi_fmpq` (*C function*), 627
`arb_sinc` (*C function*), 627

arb_sinc_pi (*C function*), 627
 arb_sinh (*C function*), 628
 arb_sinh_cosh (*C function*), 628
 arb_sqr (*C function*), 625
 arb_sqrt (*C function*), 625
 arb_sqrt1pm1 (*C function*), 625
 arb_sqrt_arf (*C function*), 625
 arb_sqrt_fmpz (*C function*), 625
 arb_sqrt_ui (*C function*), 625
 arb_sqrtpos (*C function*), 625
 arb_srcptr (*C type*), 613
 arb_struct (*C type*), 613
 arb_sub (*C function*), 622
 arb_sub_arf (*C function*), 622
 arb_sub_fmpz (*C function*), 622
 arb_sub_si (*C function*), 622
 arb_sub_ui (*C function*), 622
 arb_submul (*C function*), 623
 arb_submul_arf (*C function*), 623
 arb_submul_fmpz (*C function*), 623
 arb_submul_si (*C function*), 623
 arb_submul_ui (*C function*), 623
 arb_swap (*C function*), 614
 arb_t (*C type*), 613
 arb_tan (*C function*), 627
 arb_tan_pi (*C function*), 627
 arb_tanh (*C function*), 628
 arb_trim (*C function*), 619
 arb_trunc (*C function*), 619
 arb_ui_div (*C function*), 623
 arb_ui_pow_ui (*C function*), 625
 arb_union (*C function*), 618
 arb_unit_interval (*C function*), 616
 arb_urandom (*C function*), 617
 arb_zero (*C function*), 615
 arb_zero_pm_inf (*C function*), 615
 arb_zero_pm_one (*C function*), 616
 arb_zeta (*C function*), 631
 arb_zeta_ui (*C function*), 631
 arb_zeta_ui_asymp (*C function*), 631
 arb_zeta_ui_bernoulli (*C function*), 631
 arb_zeta_ui_borwein_bsplitt (*C function*), 631
 arb_zeta_ui_euler_product (*C function*), 631
 arb_zeta_ui_vec (*C function*), 631
 arb_zeta_ui_vec_borwein (*C function*), 631
 arb_zeta_ui_vec_even (*C function*), 631
 arb_zeta_ui_vec_odd (*C function*), 631
 arf_abs (*C function*), 608
 arf_abs_bound_le_2exp_fmpz (*C function*), 605
 arf_abs_bound_lt_2exp_fmpz (*C function*), 605
 arf_abs_bound_lt_2exp_si (*C function*), 605
 arf_add (*C function*), 608
 arf_add_fmpz (*C function*), 608
 arf_add_fmpz_2exp (*C function*), 608
 arf_add_si (*C function*), 608
 arf_add_ui (*C function*), 608
 arf_addmul (*C function*), 608
 arf_addmul_fmpz (*C function*), 608
 arf_addmul_mps (*C function*), 608
 arf_addmul_si (*C function*), 608
 arf_addmul_ui (*C function*), 608
 arf_allocated_bytes (*C function*), 602
 arf_approx_dot (*C function*), 609
 arf_bits (*C function*), 605
 arf_ceil (*C function*), 604
 arf_clear (*C function*), 602
 arf_cmp (*C function*), 605
 arf_cmp_2exp_si (*C function*), 605
 arf_cmp_d (*C function*), 605
 arf_cmp_si (*C function*), 605
 arf_cmp_ui (*C function*), 605
 arf_cmpabs (*C function*), 605
 arf_cmpabs_2exp_si (*C function*), 605
 arf_cmpabs_d (*C function*), 605
 arf_cmpabs_mag (*C function*), 605
 arf_cmpabs_ui (*C function*), 605
 arf_complex_mul (*C function*), 610
 arf_complex_mul_fallback (*C function*), 610
 arf_complex_sqr (*C function*), 610
 arf_debug (*C function*), 607
 arf_div (*C function*), 609
 arf_div_fmpz (*C function*), 609
 arf_div_si (*C function*), 609
 arf_div_ui (*C function*), 609
 arf_dot (*C function*), 609
 arf_dump_file (*C function*), 607
 arf_dump_str (*C function*), 607
 arf_equal (*C function*), 605
 arf_equal_d (*C function*), 605
 arf_equal_si (*C function*), 605
 arf_equal_ui (*C function*), 605
 arf_floor (*C function*), 604
 arf_fma (*C function*), 609
 arf_fmpz_div (*C function*), 609
 arf_fmpz_div_fmpz (*C function*), 609
 arf_fprint (*C function*), 607
 arf_fprintf (*C function*), 607
 arf_frexp (*C function*), 603
 arf_get_d (*C function*), 604
 arf_get_fmpz (*C function*), 604
 arf_get_fmpz_2exp (*C function*), 603
 arf_get_fmpz_fixed_fmpz (*C function*), 604
 arf_get_fmpz_fixed_si (*C function*), 604
 arf_get_mag (*C function*), 606
 arf_get_mag_lower (*C function*), 606
 arf_get_mpf (*C function*), 604
 arf_get_si (*C function*), 604
 arf_get_str (*C function*), 607
 arf_init (*C function*), 602
 arf_init_neg_mag_shallow (*C function*), 606
 arf_init_neg_shallow (*C function*), 606
 arf_init_set_mag_shallow (*C function*), 606
 arf_init_set_shallow (*C function*), 606
 arf_init_set_si (*C function*), 603
 arf_init_set_ui (*C function*), 603

`arf_interval_clear` (*C function*), 800
`arf_interval_fprintf` (*C function*), 800
`arf_interval_get_arb` (*C function*), 800
`arf_interval_init` (*C function*), 800
`arf_interval_printd` (*C function*), 800
`arf_interval_ptr` (*C type*), 800
`arf_interval_set` (*C function*), 800
`arf_interval_srcptr` (*C type*), 800
`arf_interval_struct` (*C type*), 800
`arf_interval_swap` (*C function*), 800
`arf_interval_t` (*C type*), 800
`arf_is_finite` (*C function*), 603
`arf_is_inf` (*C function*), 602
`arf_is_int` (*C function*), 605
`arf_is_int_2exp_si` (*C function*), 605
`arf_is_nan` (*C function*), 602
`arf_is_neg_inf` (*C function*), 602
`arf_is_normal` (*C function*), 602
`arf_is_one` (*C function*), 602
`arf_is_pos_inf` (*C function*), 602
`arf_is_special` (*C function*), 603
`arf_is_zero` (*C function*), 602
`arf_load_file` (*C function*), 607
`arf_load_str` (*C function*), 607
`arf_mag_add_ulp` (*C function*), 606
`arf_mag_fast_add_ulp` (*C function*), 606
`arf_mag_set_ulp` (*C function*), 606
`arf_max` (*C function*), 605
`arf_min` (*C function*), 605
`arf_mul` (*C function*), 608
`arf_mul_2exp_fmpz` (*C function*), 608
`arf_mul_2exp_si` (*C function*), 608
`arf_mul_fmpz` (*C function*), 608
`arf_mul_mpz` (*C function*), 608
`arf_mul_si` (*C function*), 608
`arf_mul_ui` (*C function*), 608
`arf_nan` (*C function*), 602
`arf_neg` (*C function*), 608
`arf_neg_inf` (*C function*), 602
`arf_neg_round` (*C function*), 608
`arf_nint` (*C function*), 604
`arf_one` (*C function*), 602
`arf_pos_inf` (*C function*), 602
`ARF_PREC_EXACT` (*C macro*), 602
`arf_print` (*C function*), 607
`arf_printd` (*C function*), 607
`arf_randtest` (*C function*), 607
`arf_randtest_not_zero` (*C function*), 607
`arf_randtest_special` (*C function*), 607
`ARF_RND_CEIL` (*C macro*), 601
`ARF_RND_DOWN` (*C macro*), 601
`ARF_RND_FLOOR` (*C macro*), 601
`ARF_RND_NEAR` (*C macro*), 601
`arf_rnd_t` (*C type*), 601
`ARF_RND_UP` (*C macro*), 601
`arf_root` (*C function*), 610
`arf_rsqr` (*C function*), 610
`arf_set` (*C function*), 603
`arf_set_d` (*C function*), 603
`arf_set_fmpz` (*C function*), 603
`arf_set_fmpz_2exp` (*C function*), 603
`arf_set_mag` (*C function*), 606
`arf_set_mpfr` (*C function*), 603
`arf_set_mpz` (*C function*), 603
`arf_set_round` (*C function*), 603
`arf_set_round_fmpz` (*C function*), 603
`arf_set_round_fmpz_2exp` (*C function*), 603
`arf_set_round_mpz` (*C function*), 603
`arf_set_round_si` (*C function*), 603
`arf_set_round_ui` (*C function*), 603
`arf_set_si` (*C function*), 603
`arf_set_si_2exp_si` (*C function*), 603
`arf_set_ui` (*C function*), 603
`arf_set_ui_2exp_si` (*C function*), 603
`arf_sgn` (*C function*), 605
`arf_si_div` (*C function*), 609
`arf_sosq` (*C function*), 609
`arf_sqrt` (*C function*), 610
`arf_sqrt_fmpz` (*C function*), 610
`arf_sqrt_ui` (*C function*), 610
`arf_struct` (*C type*), 601
`arf_sub` (*C function*), 608
`arf_sub_fmpz` (*C function*), 608
`arf_sub_si` (*C function*), 608
`arf_sub_ui` (*C function*), 608
`arf_submul` (*C function*), 608
`arf_submul_fmpz` (*C function*), 609
`arf_submul_mpz` (*C function*), 609
`arf_submul_si` (*C function*), 608
`arf_submul_ui` (*C function*), 608
`arf_sum` (*C function*), 609
`arf_swap` (*C function*), 603
`arf_t` (*C type*), 601
`arf_ui_div` (*C function*), 609
`arf_urandom` (*C function*), 607
`arf_zero` (*C function*), 602
`Arg` (*C macro*), 906
`ArgMax` (*C macro*), 903
`ArgMaxUnique` (*C macro*), 903
`ArgMin` (*C macro*), 903
`ArgMinUnique` (*C macro*), 903
`arith_bell_number` (*C function*), 293
`arith_bell_number_dobinski` (*C function*), 293
`arith_bell_number_multi_mod` (*C function*), 293
`arith_bell_number_nmod` (*C function*), 293
`arith_bell_number_nmod_vec` (*C function*), 293
`arith_bell_number_nmod_vec_ogf` (*C function*), 293
`arith_bell_number_nmod_vec_recursive` (*C function*), 293
`arith_bell_number_nmod_vec_series` (*C function*), 293
`arith_bell_number_size` (*C function*), 293
`arith_bell_number_vec` (*C function*), 293
`arith_bell_number_vec_multi_mod` (*C function*), 293

`arith_bell_number_vec_recursive` (*C function*), 293
`arith_bernoulli_number` (*C function*), 294
`arith_bernoulli_number_denom` (*C function*), 294
`arith_bernoulli_number_size` (*C function*), 294
`arith_bernoulli_number_vec` (*C function*), 294
`arith_bernoulli_polynomial` (*C function*), 294
`arith_divisors` (*C function*), 295
`arith_euler_number` (*C function*), 295
`arith_euler_number_size` (*C function*), 295
`arith_euler_number_vec` (*C function*), 295
`arith_euler_polynomial` (*C function*), 295
`arith_harmonic_number` (*C function*), 291
`arith_hrr_expsum_factored` (*C function*), 296
`arith_landau_function_vec` (*C function*), 296
`arith_number_of_partitions` (*C function*), 297
`arith_number_of_partitions_mpfr` (*C function*), 296
`arith_number_of_partitions_nmod_vec` (*C function*), 296
`arith_number_of_partitions_vec` (*C function*), 296
`arith_ramanujan_tau` (*C function*), 295
`arith_ramanujan_tau_series` (*C function*), 295
`arith_stirling_matrix_1` (*C function*), 292
`arith_stirling_matrix_1u` (*C function*), 292
`arith_stirling_matrix_2` (*C function*), 292
`arith_stirling_number_1` (*C function*), 292
`arith_stirling_number_1_vec` (*C function*), 292
`arith_stirling_number_1u` (*C function*), 292
`arith_stirling_number_1u_vec` (*C function*), 292
`arith_stirling_number_2` (*C function*), 292
`arith_stirling_number_2_vec` (*C function*), 292
`arith_sum_of_squares` (*C function*), 297
`arith_sum_of_squares_vec` (*C function*), 297
`Asec` (*C macro*), 908
`Asech` (*C macro*), 908
`Asin` (*C macro*), 907
`Asinh` (*C macro*), 908
`AsymptoticTo` (*C macro*), 904
`Atan` (*C macro*), 908
`Atan2` (*C macro*), 908
`Atanh` (*C macro*), 908
B
`BarnesG` (*C macro*), 909
`BellNumber` (*C macro*), 908
`bernoulli_bound_2exp_si` (*C function*), 795
`bernoulli_cache` (*C var*), 795
`bernoulli_cache_compute` (*C function*), 795
`bernoulli_cache_num` (*C var*), 795
`bernoulli_fmpq_ui` (*C function*), 795
`bernoulli_fmpq_vec_no_cache` (*C function*), 794
`bernoulli_mod_p_harvey` (*C function*), 795
`bernoulli_rev_clear` (*C function*), 794
`bernoulli_rev_init` (*C function*), 794
`bernoulli_rev_next` (*C function*), 794
`bernoulli_rev_t` (*C type*), 794
`BernoulliB` (*C macro*), 908
`BernoulliPolynomial` (*C macro*), 908
`BernsteinEllipse` (*C macro*), 902
`BesselI` (*C macro*), 910
`BesselJ` (*C macro*), 910
`BesselJZero` (*C macro*), 910
`BesselK` (*C macro*), 910
`BesselY` (*C macro*), 910
`BesselYZero` (*C macro*), 910
`BetaFunction` (*C macro*), 909
`Binomial` (*C macro*), 909
`bool_mat_add` (*C function*), 535
`bool_mat_all` (*C function*), 534
`bool_mat_all_pairs_longest_walk` (*C function*), 535
`bool_mat_any` (*C function*), 534
`bool_mat_clear` (*C function*), 533
`bool_mat_complement` (*C function*), 535
`bool_mat_directed_cycle` (*C function*), 534
`bool_mat_directed_path` (*C function*), 534
`bool_mat_equal` (*C function*), 534
`bool_mat_fprint` (*C function*), 534
`bool_mat_get_entry` (*C function*), 533
`bool_mat_get_strongly_connected_components` (*C function*), 535
`bool_mat_init` (*C function*), 533
`bool_mat_is_diagonal` (*C function*), 534
`bool_mat_is_empty` (*C function*), 533
`bool_mat_is_lower_triangular` (*C function*), 534
`bool_mat_is_nilpotent` (*C function*), 534
`bool_mat_is_square` (*C function*), 533
`bool_mat_is_transitive` (*C function*), 534
`bool_mat_mul` (*C function*), 535
`bool_mat_mul_entrywise` (*C function*), 535
`bool_mat_ncols` (*C macro*), 533
`bool_mat_nilpotency_degree` (*C function*), 535
`bool_mat_nrows` (*C macro*), 533
`bool_mat_one` (*C function*), 534
`bool_mat_pow_ui` (*C function*), 535
`bool_mat_print` (*C function*), 534
`bool_mat_randtest` (*C function*), 534
`bool_mat_randtest_diagonal` (*C function*), 534
`bool_mat_randtest_nilpotent` (*C function*), 534
`bool_mat_set` (*C function*), 533
`bool_mat_set_entry` (*C function*), 533
`bool_mat_sqr` (*C function*), 535
`bool_mat_struct` (*C type*), 533
`bool_mat_t` (*C type*), 533
`bool_mat_trace` (*C function*), 535
`bool_mat_transitive_closure` (*C function*), 535
`bool_mat_transpose` (*C function*), 535
`bool_mat_zero` (*C function*), 534
`Braces` (*C macro*), 913
`Brackets` (*C macro*), 913
`bsplit_basecase_func_t` (*C type*), 14

bsplit_clear_func_t (*C type*), 14
bsplit_init_func_t (*C type*), 14
bsplit_merge_func_t (*C type*), 14
butterfly_lshB (*C function*), 299
butterfly_rshB (*C function*), 299

C

ca_abs (*C function*), 855
ca_acos (*C function*), 859
ca_acos_direct (*C function*), 859
ca_acos_logarithm (*C function*), 859
ca_add (*C function*), 852
ca_add_fmpq (*C function*), 852
ca_add_fmpz (*C function*), 852
ca_add_si (*C function*), 852
ca_add_ui (*C function*), 852
ca_arg (*C function*), 856
ca_asin (*C function*), 859
ca_asin_direct (*C function*), 859
ca_asin_logarithm (*C function*), 859
ca_atan (*C function*), 858
ca_atan_direct (*C function*), 858
ca_atan_logarithm (*C function*), 858
ca_can_evaluate_qqbar (*C function*), 849
ca_ceil (*C function*), 856
ca_check_equal (*C function*), 852
ca_check_ge (*C function*), 852
ca_check_gt (*C function*), 852
ca_check_is_algebraic (*C function*), 851
ca_check_is_i (*C function*), 851
ca_check_is_imaginary (*C function*), 851
ca_check_is_infinity (*C function*), 851
ca_check_is_integer (*C function*), 851
ca_check_is_neg_i (*C function*), 851
ca_check_is_neg_i_inf (*C function*), 851
ca_check_is_neg_inf (*C function*), 851
ca_check_is_neg_one (*C function*), 851
ca_check_is_negative_real (*C function*), 851
ca_check_is_number (*C function*), 851
ca_check_is_one (*C function*), 851
ca_check_is_pos_i_inf (*C function*), 851
ca_check_is_pos_inf (*C function*), 851
ca_check_is_rational (*C function*), 851
ca_check_is_real (*C function*), 851
ca_check_is_signed_inf (*C function*), 851
ca_check_is_uinf (*C function*), 851
ca_check_is_undefined (*C function*), 851
ca_check_is_zero (*C function*), 851
ca_check_le (*C function*), 852
ca_check_lt (*C function*), 852
ca_clear (*C function*), 846
ca_cmp_repr (*C function*), 850
ca_condense_field (*C function*), 852
ca_conj (*C function*), 856
ca_conj_deep (*C function*), 856
ca_conj_shallow (*C function*), 856
ca_cos (*C function*), 858
ca_cot (*C function*), 858
ca_csgn (*C function*), 856
ca_ctx_clear (*C function*), 845
ca_ctx_init (*C function*), 845
ca_ctx_print (*C function*), 845
ca_ctx_struct (*C type*), 845
ca_ctx_t (*C type*), 845
ca_div (*C function*), 853
ca_div_fmpq (*C function*), 853
ca_div_fmpz (*C function*), 853
ca_div_si (*C function*), 853
ca_div_ui (*C function*), 853
ca_dot (*C function*), 854
ca_equal_repr (*C function*), 850
ca_erf (*C function*), 859
ca_erfc (*C function*), 859
ca_erfi (*C function*), 859
ca_euler (*C function*), 848
ca_exp (*C function*), 857
ca_ext_cache_clear (*C function*), 884
ca_ext_cache_init (*C function*), 884
ca_ext_cache_insert (*C function*), 884
ca_ext_cache_struct (*C type*), 884
ca_ext_cache_t (*C type*), 884
ca_ext_clear (*C function*), 883
ca_ext_cmp_repr (*C function*), 883
ca_ext_equal_repr (*C function*), 883
CA_EXT_FUNC_ARGS (*C macro*), 882
CA_EXT_FUNC_ENCLOSURE (*C macro*), 882
CA_EXT_FUNC_NARGS (*C macro*), 882
CA_EXT_FUNC_PREC (*C macro*), 882
ca_ext_get_acb_raw (*C function*), 884
ca_ext_get_arg (*C function*), 883
ca_ext_hash (*C function*), 883
CA_EXT_HASH (*C macro*), 882
CA_EXT_HEAD (*C macro*), 882
ca_ext_init_const (*C function*), 883
ca_ext_init_fx (*C function*), 883
ca_ext_init_fxn (*C function*), 883
ca_ext_init_fxy (*C function*), 883
ca_ext_init_qqbar (*C function*), 883
ca_ext_init_set (*C function*), 883
ca_ext_nargs (*C function*), 883
ca_ext_print (*C function*), 883
ca_ext_ptr (*C type*), 882
CA_EXT_QQBAR (*C macro*), 882
CA_EXT_QQBAR_NF (*C macro*), 882
ca_ext_srcptr (*C type*), 882
ca_ext_struct (*C type*), 882
ca_ext_t (*C type*), 882
ca_factor (*C function*), 861
ca_factor_clear (*C function*), 860
ca_factor_get_ca (*C function*), 860
ca_factor_init (*C function*), 860
ca_factor_insert (*C function*), 860
ca_factor_one (*C function*), 860
CA_FACTOR_POLY_CONTENT (*C macro*), 861
CA_FACTOR_POLY_FULL (*C macro*), 861
CA_FACTOR_POLY_NONE (*C macro*), 861

CA_FACTOR_POLY_SQF (*C macro*), 861
 ca_factor_print (*C function*), 860
 ca_factor_struct (*C type*), 860
 ca_factor_t (*C type*), 860
 CA_FACTOR_ZZ_FULL (*C macro*), 861
 CA_FACTOR_ZZ_NONE (*C macro*), 861
 CA_FACTOR_ZZ_SMOOTH (*C macro*), 861
 ca_field_build_ideal (*C function*), 887
 ca_field_build_ideal_erf (*C function*), 887
 ca_field_cache_clear (*C function*), 887
 ca_field_cache_init (*C function*), 887
 ca_field_cache_insert_ext (*C function*), 887
 ca_field_cache_struct (*C type*), 887
 ca_field_cache_t (*C type*), 887
 ca_field_clear (*C function*), 886
 ca_field_cmp (*C function*), 887
 CA_FIELD_EXT (*C macro*), 885
 CA_FIELD_EXT_ELEM (*C macro*), 885
 CA_FIELD_HASH (*C macro*), 885
 CA_FIELD_IDEAL (*C macro*), 886
 CA_FIELD_IDEAL_ELEM (*C macro*), 886
 CA_FIELD_IDEAL_LENGTH (*C macro*), 886
 ca_field_init_const (*C function*), 886
 ca_field_init_fx (*C function*), 886
 ca_field_init_fxy (*C function*), 886
 ca_field_init_multi (*C function*), 886
 ca_field_init_nf (*C function*), 886
 ca_field_init_qq (*C function*), 886
 CA_FIELD_IS_GENERIC (*C macro*), 885
 CA_FIELD_IS_NF (*C macro*), 885
 CA_FIELD_IS_QQ (*C macro*), 885
 CA_FIELD_LENGTH (*C macro*), 885
 CA_FIELD_MCTX (*C macro*), 886
 CA_FIELD_NF (*C macro*), 885
 CA_FIELD_NF_QQBAR (*C macro*), 885
 ca_field_print (*C function*), 887
 ca_field_ptr (*C type*), 885
 ca_field_set_ext (*C function*), 886
 ca_field_srcptr (*C type*), 885
 ca_field_struct (*C type*), 885
 ca_field_t (*C type*), 885
 ca_floor (*C function*), 856
 CA_FMPQ (*C macro*), 863
 CA_FMPQ_DENREF (*C macro*), 863
 ca_fmpq_div (*C function*), 853
 CA_FMPQ_NUMREF (*C macro*), 863
 ca_fmpq_poly_evaluate (*C function*), 854
 ca_fmpq_sub (*C function*), 853
 ca_fmpz_div (*C function*), 853
 ca_fmpz_mpoly_evaluate (*C function*), 854
 ca_fmpz_mpoly_evaluate_horner (*C function*), 854
 ca_fmpz_mpoly_evaluate_iter (*C function*), 854
 ca_fmpz_mpoly_q_evaluate (*C function*), 854
 ca_fmpz_mpoly_q_evaluate_no_division_by_zero (*C function*), 854
 ca_fmpz_poly_evaluate (*C function*), 854
 ca_fmpz_sub (*C function*), 853
 ca_fprint (*C function*), 847
 ca_gamma (*C function*), 859
 ca_get_acb (*C function*), 859
 ca_get_acb_accurate_parts (*C function*), 859
 ca_get_acb_raw (*C function*), 859
 ca_get_decimal_str (*C function*), 860
 ca_get_fexpr (*C function*), 846
 ca_get_fmpq (*C function*), 849
 ca_get_fmpz (*C function*), 849
 ca_get_qqbar (*C function*), 849
 ca_get_str (*C function*), 847
 ca_hash_repr (*C function*), 850
 ca_i (*C function*), 848
 ca_im (*C function*), 856
 ca_init (*C function*), 846
 ca_inv (*C function*), 853
 ca_inv_no_division_by_zero (*C function*), 854
 ca_is_cyclotomic_nf_elem (*C function*), 850
 ca_is_gen_as_ext (*C function*), 852
 ca_is_generic_elem (*C function*), 850
 ca_is_nf_elem (*C function*), 850
 ca_is_qq_elem (*C function*), 850
 ca_is_qq_elem_integer (*C function*), 850
 ca_is_qq_elem_one (*C function*), 850
 ca_is_qq_elem_zero (*C function*), 850
 ca_is_special (*C function*), 850
 ca_is_unknown (*C function*), 850
 ca_log (*C function*), 857
 ca_mat_add (*C function*), 875
 ca_mat_add_ca (*C function*), 876
 ca_mat_addmul_ca (*C function*), 876
 ca_mat_adjugate (*C function*), 879
 ca_mat_adjugate_charpoly (*C function*), 879
 ca_mat_adjugate_cofactor (*C function*), 879
 ca_mat_ca_poly_evaluate (*C function*), 876
 ca_mat_charpoly (*C function*), 880
 ca_mat_charpoly_berkowitz (*C function*), 880
 ca_mat_charpoly_danilevsky (*C function*), 880
 ca_mat_check_equal (*C function*), 875
 ca_mat_check_is_one (*C function*), 875
 ca_mat_check_is_zero (*C function*), 875
 ca_mat_clear (*C function*), 873
 ca_mat_companion (*C function*), 880
 ca_mat_conj (*C function*), 875
 ca_mat_conj_transpose (*C function*), 875
 ca_mat_det (*C function*), 879
 ca_mat_det_bareiss (*C function*), 879
 ca_mat_det_berkowitz (*C function*), 879
 ca_mat_det_cofactor (*C function*), 879
 ca_mat_det_lu (*C function*), 879
 ca_mat_dft (*C function*), 875
 ca_mat_diagonalization (*C function*), 880
 ca_mat_div_ca (*C function*), 876
 ca_mat_div_fmpq (*C function*), 876
 ca_mat_div_fmpz (*C function*), 876
 ca_mat_div_si (*C function*), 876
 ca_mat_eigenvalues (*C function*), 880
 ca_mat_entry (*C macro*), 873

`ca_mat_entry_ptr` (*C function*), 873
`ca_mat_exp` (*C function*), 881
`ca_mat_fflu` (*C function*), 877
`ca_mat_find_pivot` (*C function*), 877
`ca_mat_hilbert` (*C function*), 875
`ca_mat_init` (*C function*), 873
`ca_mat_inv` (*C function*), 878
`ca_mat_jordan_blocks` (*C function*), 880
`ca_mat_jordan_form` (*C function*), 881
`ca_mat_jordan_transformation` (*C function*), 880
`ca_mat_log` (*C function*), 881
`ca_mat_lu` (*C function*), 877
`ca_mat_lu_classical` (*C function*), 877
`ca_mat_lu_recursive` (*C function*), 877
`ca_mat_mul` (*C function*), 875
`ca_mat_mul_ca` (*C function*), 876
`ca_mat_mul_classical` (*C function*), 875
`ca_mat_mul_fmpq` (*C function*), 876
`ca_mat_mul_fmpz` (*C function*), 876
`ca_mat_mul_same_nf` (*C function*), 875
`ca_mat_mul_si` (*C function*), 876
`ca_mat_ncols` (*C macro*), 873
`ca_mat_neg` (*C function*), 875
`ca_mat_nonsingular_fflu` (*C function*), 877
`ca_mat_nonsingular_lu` (*C function*), 877
`ca_mat_nonsingular_solve` (*C function*), 878
`ca_mat_nonsingular_solve_adjugate` (*C function*), 878
`ca_mat_nonsingular_solve_fflu` (*C function*), 878
`ca_mat_nonsingular_solve_lu` (*C function*), 878
`ca_mat_nrows` (*C macro*), 873
`ca_mat_one` (*C function*), 874
`ca_mat_ones` (*C function*), 874
`ca_mat_pascal` (*C function*), 874
`ca_mat_pow_ui_binexp` (*C function*), 876
`ca_mat_print` (*C function*), 874
`ca_mat_printn` (*C function*), 874
`ca_mat_randops` (*C function*), 874
`ca_mat_randtest` (*C function*), 874
`ca_mat_randtest_rational` (*C function*), 874
`ca_mat_rank` (*C function*), 878
`ca_mat_right_kernel` (*C function*), 879
`ca_mat_rref` (*C function*), 878
`ca_mat_rref_fflu` (*C function*), 878
`ca_mat_rref_lu` (*C function*), 878
`ca_mat_set` (*C function*), 874
`ca_mat_set_ca` (*C function*), 874
`ca_mat_set_fmpq_mat` (*C function*), 874
`ca_mat_set_fmpz_mat` (*C function*), 874
`ca_mat_set_jordan_blocks` (*C function*), 880
`ca_mat_solve_fflu_precomp` (*C function*), 878
`ca_mat_solve_lu_precomp` (*C function*), 878
`ca_mat_solve_tril` (*C function*), 878
`ca_mat_solve_tril_classical` (*C function*), 878
`ca_mat_solve_tril_recursive` (*C function*), 878
`ca_mat_solve_triu` (*C function*), 878
`ca_mat_solve_triu_classical` (*C function*), 878
`ca_mat_solve_triu_recursive` (*C function*), 878
`ca_mat_sqr` (*C function*), 876
`ca_mat_stirling` (*C function*), 874
`ca_mat_struct` (*C type*), 873
`ca_mat_sub` (*C function*), 875
`ca_mat_sub_ca` (*C function*), 876
`ca_mat_submul_ca` (*C function*), 876
`ca_mat_swap` (*C function*), 873
`ca_mat_t` (*C type*), 873
`ca_mat_trace` (*C function*), 879
`ca_mat_transfer` (*C function*), 874
`ca_mat_transpose` (*C function*), 875
`ca_mat_window_clear` (*C function*), 873
`ca_mat_window_init` (*C function*), 873
`ca_mat_zero` (*C function*), 874
`ca_merge_fields` (*C function*), 852
`CA_MPOLY_Q` (*C macro*), 863
`ca_mul` (*C function*), 853
`ca_mul_fmpq` (*C function*), 853
`ca_mul_fmpz` (*C function*), 853
`ca_mul_si` (*C function*), 853
`ca_mul_ui` (*C function*), 853
`ca_neg` (*C function*), 852
`ca_neg_i` (*C function*), 848
`ca_neg_i_inf` (*C function*), 848
`ca_neg_inf` (*C function*), 848
`ca_neg_one` (*C function*), 848
`CA_NF_ELEM` (*C macro*), 863
`ca_one` (*C function*), 848
`CA_OPT_GROEBNER_LENGTH_LIMIT` (*C macro*), 862
`CA_OPT_GROEBNER_POLY_BITS_LIMIT` (*C macro*), 862
`CA_OPT_GROEBNER_POLY_LENGTH_LIMIT` (*C macro*), 862
`CA_OPT_LLL_PREC` (*C macro*), 862
`CA_OPT_LOW_PREC` (*C macro*), 862
`CA_OPT_MPOLY_ORD` (*C macro*), 861
`CA_OPT_POW_LIMIT` (*C macro*), 862
`CA_OPT_PREC_LIMIT` (*C macro*), 862
`CA_OPT_PRINT_FLAGS` (*C macro*), 861
`CA_OPT_QQBAR_DEG_LIMIT` (*C macro*), 862
`CA_OPT_SMOOTH_LIMIT` (*C macro*), 862
`CA_OPT_TRIG_FORM` (*C macro*), 862
`CA_OPT_TRIG_FORM.CA_TRIG_DIRECT` (*C macro*), 862
`CA_OPT_TRIG_FORM.CA_TRIG_EXPONENTIAL` (*C macro*), 862
`CA_OPT_TRIG_FORM.CA_TRIG_SINE_COSINE` (*C macro*), 862
`CA_OPT_TRIG_FORM.CA_TRIG_TANGENT` (*C macro*), 862
`CA_OPT_USE_GROEBNER` (*C macro*), 862
`CA_OPT_VERBOSE` (*C macro*), 861
`CA_OPT_VIETA_LIMIT` (*C macro*), 862
`ca_pi` (*C function*), 848
`ca_pi_i` (*C function*), 848
`ca_poly_add` (*C function*), 869

`ca_poly_check_equal` (*C function*), 869
`ca_poly_check_is_one` (*C function*), 869
`ca_poly_check_is_zero` (*C function*), 869
`ca_poly_clear` (*C function*), 867
`ca_poly_compose` (*C function*), 870
`ca_poly_derivative` (*C function*), 871
`ca_poly_div` (*C function*), 870
`ca_poly_div_ca` (*C function*), 870
`ca_poly_div_series` (*C function*), 871
`ca_poly_divrem` (*C function*), 870
`ca_poly_divrem_basecase` (*C function*), 870
`ca_poly_evaluate` (*C function*), 870
`ca_poly_evaluate_horner` (*C function*), 870
`ca_poly_exp_series` (*C function*), 871
`ca_poly_factor_squarefree` (*C function*), 872
`ca_poly_fit_length` (*C function*), 867
`ca_poly_gcd` (*C function*), 871
`ca_poly_gcd_euclidean` (*C function*), 871
`ca_poly_init` (*C function*), 867
`ca_poly_integral` (*C function*), 871
`ca_poly_inv_series` (*C function*), 871
`ca_poly_is_proper` (*C function*), 869
`ca_poly_log_series` (*C function*), 871
`ca_poly_make_monic` (*C function*), 869
`ca_poly_mul` (*C function*), 869
`ca_poly_mul_ca` (*C function*), 870
`ca_poly_mullo` (*C function*), 870
`ca_poly_neg` (*C function*), 869
`ca_poly_one` (*C function*), 868
`ca_poly_pow_ui` (*C function*), 870
`ca_poly_pow_ui_trunc` (*C function*), 870
`ca_poly_print` (*C function*), 868
`ca_poly_printn` (*C function*), 868
`ca_poly_randtest` (*C function*), 868
`ca_poly_randtest_rational` (*C function*), 868
`ca_poly_rem` (*C function*), 870
`ca_poly_reverse` (*C function*), 869
`ca_poly_roots` (*C function*), 872
`ca_poly_set` (*C function*), 868
`ca_poly_set_ca` (*C function*), 868
`ca_poly_set_coeff_ca` (*C function*), 868
`ca_poly_set_fmpq_poly` (*C function*), 868
`ca_poly_set_fmpz_poly` (*C function*), 868
`ca_poly_set_roots` (*C function*), 872
`ca_poly_set_si` (*C function*), 868
`ca_poly_shift_left` (*C function*), 869
`ca_poly_shift_right` (*C function*), 869
`ca_poly_squarefree_part` (*C function*), 872
`ca_poly_struct` (*C type*), 867
`ca_poly_sub` (*C function*), 869
`ca_poly_t` (*C type*), 867
`ca_poly_transfer` (*C function*), 868
`ca_poly_vec_append` (*C function*), 872
`ca_poly_vec_clear` (*C function*), 872
`ca_poly_vec_init` (*C function*), 872
`ca_poly_vec_set_length` (*C function*), 872
`ca_poly_vec_struct` (*C type*), 872
`ca_poly_vec_t` (*C type*), 872
`ca_poly_x` (*C function*), 868
`ca_poly_zero` (*C function*), 868
`ca_pos_i_inf` (*C function*), 848
`ca_pos_inf` (*C function*), 848
`ca_pow` (*C function*), 855
`ca_pow_fmpq` (*C function*), 855
`ca_pow_fmpz` (*C function*), 855
`ca_pow_si` (*C function*), 855
`ca_pow_si_arithmetic` (*C function*), 855
`ca_pow_ui` (*C function*), 855
`ca_print` (*C function*), 847
`CA_PRINT_DEBUG` (*C macro*), 846
`CA_PRINT_DEFAULT` (*C macro*), 846
`CA_PRINT_DIGITS` (*C macro*), 846
`CA_PRINT_FIELD` (*C macro*), 846
`CA_PRINT_N` (*C macro*), 846
`CA_PRINT_REPR` (*C macro*), 846
`ca_printn` (*C function*), 847
`ca_ptr` (*C type*), 845
`ca_randtest` (*C function*), 849
`ca_randtest_rational` (*C function*), 849
`ca_randtest_same_nf` (*C function*), 849
`ca_randtest_special` (*C function*), 849
`ca_re` (*C function*), 856
`ca_rewrite_complex_normal_form` (*C function*), 860
`ca_set` (*C function*), 848
`ca_set_d` (*C function*), 848
`ca_set_d_d` (*C function*), 848
`ca_set_fexpr` (*C function*), 846
`ca_set_fmpq` (*C function*), 848
`ca_set_fmpz` (*C function*), 848
`ca_set_qqbar` (*C function*), 849
`ca_set_si` (*C function*), 848
`ca_set_ui` (*C function*), 848
`ca_sgn` (*C function*), 855
`ca_si_div` (*C function*), 853
`ca_si_sub` (*C function*), 853
`ca_sin` (*C function*), 858
`ca_sin_cos` (*C function*), 857
`ca_sin_cos_direct` (*C function*), 857
`ca_sin_cos_exponential` (*C function*), 857
`ca_sin_cos_tangent` (*C function*), 857
`ca_sqr` (*C function*), 855
`ca_sqrt` (*C function*), 855
`ca_sqrt_factor` (*C function*), 855
`ca_sqrt_inert` (*C function*), 855
`ca_sqrt_nofactor` (*C function*), 855
`ca_sqrt_ui` (*C function*), 855
`ca_srcptr` (*C type*), 845
`ca_struct` (*C type*), 845
`ca_sub` (*C function*), 853
`ca_sub_fmpq` (*C function*), 853
`ca_sub_fmpz` (*C function*), 853
`ca_sub_si` (*C function*), 853
`ca_sub_ui` (*C function*), 853
`ca_swap` (*C function*), 846
`ca_t` (*C type*), 845

`ca_tan` (*C function*), 858
`ca_tan_direct` (*C function*), 858
`ca_tan_exponential` (*C function*), 858
`ca_tan_sine_cosine` (*C function*), 858
`ca_transfer` (*C function*), 848
`ca_ui_div` (*C function*), 853
`ca_ui_sub` (*C function*), 853
`ca_uinf` (*C function*), 848
`ca_undefined` (*C function*), 848
`ca_unknown` (*C function*), 848
`ca_vec_append` (*C function*), 865
`ca_vec_clear` (*C function*), 864
`ca_vec_entry` (*C macro*), 864
`ca_vec_init` (*C function*), 864
`ca_vec_length` (*C function*), 864
`ca_vec_neg` (*C function*), 865
`ca_vec_print` (*C function*), 865
`ca_vec_printn` (*C function*), 865
`ca_vec_set` (*C function*), 865
`ca_vec_set_length` (*C function*), 864
`ca_vec_struct` (*C type*), 864
`ca_vec_swap` (*C function*), 864
`ca_vec_t` (*C type*), 864
`ca_vec_zero` (*C function*), 865
`ca_zero` (*C function*), 848
`calcium_fmpz_hash` (*C function*), 843
`calcium_stream_init_file` (*C function*), 843
`calcium_stream_init_str` (*C function*), 843
`calcium_stream_struct` (*C type*), 843
`calcium_stream_t` (*C type*), 843
`calcium_version` (*C function*), 843
`calcium_write` (*C function*), 843
`calcium_write_acb` (*C function*), 843
`calcium_write_arb` (*C function*), 843
`calcium_write_fmpz` (*C function*), 843
`calcium_write_free` (*C function*), 843
`calcium_write_si` (*C function*), 843
`Call` (*C macro*), 906
`CallIndeterminate` (*C macro*), 906
`Cardinality` (*C macro*), 899
`CarlsonHypergeometricR` (*C macro*), 912
`CarlsonHypergeometricT` (*C macro*), 912
`CarlsonRC` (*C macro*), 912
`CarlsonRD` (*C macro*), 912
`CarlsonRF` (*C macro*), 912
`CarlsonRG` (*C macro*), 912
`CarlsonRJ` (*C macro*), 912
`CartesianPower` (*C macro*), 899
`CartesianProduct` (*C macro*), 899
`Case` (*C macro*), 898
`Cases` (*C macro*), 898
`CatalanConstant` (*C macro*), 899
`CC` (*C macro*), 901
`cd_poly_find_roots` (*C function*), 687
`cd_poly_refine_roots` (*C function*), 687
`cd_poly_refine_roots_with_pivot` (*C function*), 687
`Ceil` (*C macro*), 906
`Characteristic` (*C macro*), 905
`ChebyshevT` (*C macro*), 909
`ChebyshevU` (*C macro*), 909
`ClosedComplexDisk` (*C macro*), 902
`ClosedOpenInterval` (*C macro*), 901
`COEFF_IS_MPZ` (*C macro*), 156
`COEFF_MAX` (*C macro*), 156
`COEFF_MIN` (*C macro*), 156
`COEFF_TO_PTR` (*C function*), 156
`Coefficient` (*C macro*), 905
`Column` (*C macro*), 905
`ColumnMatrix` (*C macro*), 905
`CommutativeRings` (*C macro*), 906
`complex_double` (*C type*), 808
`ComplexBranchDerivative` (*C macro*), 904
`ComplexDerivative` (*C macro*), 904
`ComplexInfinites` (*C macro*), 902
`ComplexLimit` (*C macro*), 904
`ComplexSignedInfinites` (*C macro*), 902
`ComplexSingularityClosure` (*C macro*), 903
`ComplexZeroMultiplicity` (*C macro*), 904
`Concatenation` (*C macro*), 899
`CongruentMod` (*C macro*), 906
`Conjugate` (*C macro*), 906
`ConreyGenerator` (*C macro*), 912
`Cos` (*C macro*), 907
`Cosh` (*C macro*), 907
`CoshIntegral` (*C macro*), 910
`CosIntegral` (*C macro*), 910
`Cot` (*C macro*), 907
`Coth` (*C macro*), 907
`CoulombC` (*C macro*), 911
`CoulombF` (*C macro*), 910
`CoulombG` (*C macro*), 910
`CoulombH` (*C macro*), 910
`CoulombSigma` (*C macro*), 911
`Csc` (*C macro*), 907
`Csch` (*C macro*), 907
`Csgn` (*C macro*), 906
`CurvePath` (*C macro*), 904
`Cyclotomic` (*C macro*), 908

D

`d_is_nan` (*C function*), 1089
`d_lambertw` (*C function*), 1089
`d_log2` (*C function*), 1090
`d_mat_approx_equal` (*C function*), 1092
`d_mat_clear` (*C function*), 1091
`d_mat_entry` (*C function*), 1091
`d_mat_entry_ptr` (*C function*), 1092
`d_mat_equal` (*C function*), 1092
`d_mat_get_entry` (*C function*), 1091
`d_mat_init` (*C function*), 1091
`d_mat_is_square` (*C function*), 1092
`d_mat_mul_classical` (*C function*), 1092
`d_mat_print` (*C function*), 1092
`d_mat_randtest` (*C function*), 1092
`d_mat_set` (*C function*), 1091

d_mat_swap_entrywise (*C function*), 1091
 d_mat_transpose (*C function*), 1092
 d_mat_zero (*C function*), 1092
 d_mul_2exp (*C function*), 1089
 d_mul_2exp_inrange (*C function*), 1089
 d_mul_2exp_inrange2 (*C function*), 1089
 d_polyval (*C function*), 1089
 d_randtest (*C function*), 1089
 d_randtest_signed (*C function*), 1089
 d_randtest_special (*C function*), 1089
 Decimal (*C macro*), 900
 DedekindEta (*C macro*), 912
 DedekindEtaEpsilon (*C macro*), 913
 DedekindSum (*C macro*), 913
 Def (*C macro*), 897
 Derivative (*C macro*), 904
 Det (*C macro*), 905
 DiagonalMatrix (*C macro*), 905
 DigammaFunction (*C macro*), 909
 DigammaFunctionZero (*C macro*), 909
 dirichlet_char_clear (*C function*), 526
 dirichlet_char_eq (*C function*), 527
 dirichlet_char_eq_deep (*C function*), 527
 dirichlet_char_exp (*C function*), 526
 dirichlet_char_first_primitive (*C function*), 526
 dirichlet_char_index (*C function*), 527
 dirichlet_char_init (*C function*), 526
 dirichlet_char_is_primitive (*C function*), 527
 dirichlet_char_is_principal (*C function*), 527
 dirichlet_char_is_real (*C function*), 527
 dirichlet_char_lift (*C function*), 528
 dirichlet_char_log (*C function*), 526
 dirichlet_char_lower (*C function*), 528
 dirichlet_char_mul (*C function*), 528
 dirichlet_char_next (*C function*), 526
 dirichlet_char_next_primitive (*C function*), 527
 dirichlet_char_one (*C function*), 526
 dirichlet_char_pow (*C function*), 528
 dirichlet_char_print (*C function*), 526
 dirichlet_char_set (*C function*), 526
 dirichlet_char_struct (*C type*), 526
 dirichlet_char_t (*C type*), 526
 dirichlet_chi (*C function*), 528
 dirichlet_chi_vec (*C function*), 528
 dirichlet_chi_vec_order (*C function*), 528
 dirichlet_conductor_char (*C function*), 527
 dirichlet_conductor_ui (*C function*), 527
 dirichlet_group_clear (*C function*), 525
 dirichlet_group_dlog_clear (*C function*), 526
 dirichlet_group_dlog_precompute (*C function*), 526
 dirichlet_group_init (*C function*), 525
 dirichlet_group_num_primitive (*C function*), 526
 dirichlet_group_size (*C function*), 525
 dirichlet_group_struct (*C type*), 525
 dirichlet_group_t (*C type*), 525
 dirichlet_index_char (*C function*), 527
 dirichlet_order_char (*C function*), 527
 dirichlet_order_ui (*C function*), 527
 dirichlet_pairing (*C function*), 528
 dirichlet_pairing_char (*C function*), 528
 dirichlet_parity_char (*C function*), 527
 dirichlet_parity_ui (*C function*), 527
 dirichlet_subgroup_init (*C function*), 525
 DirichletCharacter (*C macro*), 912
 DirichletGroup (*C macro*), 912
 DirichletL (*C macro*), 911
 DirichletLambda (*C macro*), 911
 DirichletLZero (*C macro*), 911
 DiscreteLog (*C macro*), 907
 Div (*C macro*), 900
 Divides (*C macro*), 906
 DivisorProduct (*C macro*), 903
 DivisorSigma (*C macro*), 907
 DivisorSum (*C macro*), 903
 dlog_bsgs (*C function*), 531
 dlog_bsgs_clear (*C function*), 531
 dlog_bsgs_init (*C function*), 531
 dlog_bsgs_struct (*C type*), 531
 dlog_bsgs_t (*C type*), 531
 dlog_crt (*C function*), 532
 dlog_crt_clear (*C function*), 532
 dlog_crt_init (*C function*), 532
 dlog_crt_struct (*C type*), 532
 dlog_crt_t (*C type*), 532
 dlog_modpe (*C function*), 531
 dlog_modpe_clear (*C function*), 531
 dlog_modpe_init (*C function*), 531
 dlog_modpe_struct (*C type*), 531
 dlog_modpe_t (*C type*), 531
 DLOG_NONE (*C macro*), 529
 dlog_once (*C function*), 529
 dlog_power (*C function*), 532
 dlog_power_clear (*C function*), 532
 dlog_power_init (*C function*), 532
 dlog_power_struct (*C type*), 532
 dlog_power_t (*C type*), 532
 dlog_precomp (*C function*), 529
 dlog_precomp_clear (*C function*), 529
 dlog_precomp_modpe_init (*C function*), 529
 dlog_precomp_n_init (*C function*), 529
 dlog_precomp_p_init (*C function*), 529
 dlog_precomp_pe_init (*C function*), 529
 dlog_precomp_small_init (*C function*), 529
 dlog_precomp_struct (*C type*), 529
 dlog_precomp_t (*C type*), 529
 dlog_rho (*C function*), 532
 dlog_rho_clear (*C function*), 532
 dlog_rho_init (*C function*), 532
 dlog_rho_struct (*C type*), 532
 dlog_rho_t (*C type*), 532
 dlog_table (*C function*), 531
 dlog_table_clear (*C function*), 531

dlog_table_init (*C function*), 531
dlog_table_struct (*C type*), 531
dlog_table_t (*C type*), 531
dlog_vec (*C function*), 530
dlog_vec_add (*C function*), 530
dlog_vec_eratos (*C function*), 530
dlog_vec_eratos_add (*C function*), 530
dlog_vec_fill (*C function*), 530
dlog_vec_loop (*C function*), 530
dlog_vec_loop_add (*C function*), 530
dlog_vec_set_not_found (*C function*), 530
dlog_vec_sieve (*C function*), 530
dlog_vec_sieve_add (*C function*), 530
do_func_t (*C type*), 13
dot_params_t (*C type*), 387
DoubleFactorial (*C macro*), 909

E

EisensteinE (*C macro*), 913
EisensteinG (*C macro*), 913
Element (*C macro*), 898
Ellipsis (*C macro*), 913
EllipticE (*C macro*), 912
EllipticK (*C macro*), 912
EllipticPi (*C macro*), 912
EllipticRootE (*C macro*), 913
Enclosure (*C macro*), 900
Equal (*C macro*), 897
EqualAndElement (*C macro*), 898
EqualNearestDecimal (*C macro*), 901
EqualQSeriesEllipsis (*C macro*), 906
Equivalent (*C macro*), 898
Erf (*C macro*), 910
Erfc (*C macro*), 910
Erfi (*C macro*), 910
Euler (*C macro*), 899
EulerE (*C macro*), 908
EulerPhi (*C macro*), 907
EulerPolynomial (*C macro*), 908
EulerQSeries (*C macro*), 913
Exists (*C macro*), 898
Exp (*C macro*), 907
ExpIntegrale (*C macro*), 910
ExpIntegralei (*C macro*), 910
ExtendedRealNumbers (*C macro*), 902

F

Factorial (*C macro*), 909
FallingFactorial (*C macro*), 909
False (*C macro*), 897
fermat_to_mpz (*C function*), 298
fexpr_add (*C function*), 894
fexpr_allocated_bytes (*C function*), 890
fexpr_arg (*C function*), 892
fexpr_arithmetic_nodes (*C function*), 894
fexpr_builtin_length (*C function*), 896
fexpr_builtin_lookup (*C function*), 896
fexpr_builtin_name (*C function*), 896

fexpr_call0 (*C function*), 893
fexpr_call1 (*C function*), 893
fexpr_call2 (*C function*), 893
fexpr_call3 (*C function*), 893
fexpr_call4 (*C function*), 893
fexpr_call_builtin1 (*C function*), 893
fexpr_call_builtin2 (*C function*), 893
fexpr_call_vec (*C function*), 893
fexpr_clear (*C function*), 890
fexpr_cmp_fast (*C function*), 890
fexpr_contains (*C function*), 893
fexpr_depth (*C function*), 890
fexpr_div (*C function*), 894
fexpr_equal (*C function*), 890
fexpr_equal_si (*C function*), 890
fexpr_equal_ui (*C function*), 890
fexpr_expanded_normal_form (*C function*), 895
fexpr_fit_size (*C function*), 890
fexpr_func (*C function*), 892
fexpr_get_fmpz (*C function*), 891
fexpr_get_fmpz_mpoly_q (*C function*), 894
fexpr_get_str (*C function*), 892
fexpr_get_str_latex (*C function*), 892
fexpr_get_string (*C function*), 891
fexpr_get_symbol_str (*C function*), 891
fexpr_hash (*C function*), 890
fexpr_init (*C function*), 890
fexpr_is_any_builtin_call (*C function*), 893
fexpr_is_any_builtin_symbol (*C function*), 891
fexpr_is_arithmetic_operation (*C function*), 894
fexpr_is_atom (*C function*), 891
fexpr_is_builtin_call (*C function*), 893
fexpr_is_builtin_symbol (*C function*), 891
fexpr_is_integer (*C function*), 891
fexpr_is_neg_integer (*C function*), 891
fexpr_is_string (*C function*), 891
fexpr_is_symbol (*C function*), 891
fexpr_is_zero (*C function*), 891
FEXPR_LATEX_LOGIC (*C macro*), 892
FEXPR_LATEX_SMALL (*C macro*), 892
fexpr_mul (*C function*), 894
fexpr_nargs (*C function*), 892
fexpr_neg (*C function*), 894
fexpr_num_leaves (*C function*), 890
fexpr_pow (*C function*), 894
fexpr_print (*C function*), 892
fexpr_print_latex (*C function*), 892
fexpr_ptr (*C type*), 889
fexpr_replace (*C function*), 893
fexpr_replace2 (*C function*), 893
fexpr_replace_vec (*C function*), 893
fexpr_set (*C function*), 890
fexpr_set_arf (*C function*), 894
fexpr_set_d (*C function*), 894
fexpr_set_fmpq (*C function*), 894
fexpr_set_fmpz (*C function*), 891
fexpr_set_fmpz_mpoly (*C function*), 894

- `fexpr_set_fmpz_mpoly_q` (*C function*), 894
- `fexpr_set_re_im_d` (*C function*), 894
- `fexpr_set_si` (*C function*), 891
- `fexpr_set_string` (*C function*), 891
- `fexpr_set_symbol_builtin` (*C function*), 891
- `fexpr_set_symbol_str` (*C function*), 891
- `fexpr_set_ui` (*C function*), 891
- `fexpr_size` (*C function*), 890
- `fexpr_size_bytes` (*C function*), 890
- `fexpr_srcptr` (*C type*), 889
- `fexpr_struct` (*C type*), 889
- `fexpr_sub` (*C function*), 894
- `fexpr_swap` (*C function*), 890
- `fexpr_t` (*C type*), 889
- `fexpr_vec_append` (*C function*), 895
- `fexpr_vec_clear` (*C function*), 895
- `fexpr_vec_entry` (*C macro*), 889
- `fexpr_vec_fit_length` (*C function*), 895
- `fexpr_vec_init` (*C function*), 895
- `fexpr_vec_insert_unique` (*C function*), 895
- `fexpr_vec_print` (*C function*), 895
- `fexpr_vec_set` (*C function*), 895
- `fexpr_vec_set_length` (*C function*), 895
- `fexpr_vec_struct` (*C type*), 889
- `fexpr_vec_swap` (*C function*), 895
- `fexpr_vec_t` (*C type*), 889
- `fexpr_view_arg` (*C function*), 892
- `fexpr_view_func` (*C function*), 892
- `fexpr_view_next` (*C function*), 892
- `fexpr_write` (*C function*), 892
- `fexpr_write_latex` (*C function*), 892
- `fexpr_zero` (*C function*), 891
- `fft_adjust` (*C function*), 299
- `fft_adjust_limbs` (*C function*), 304
- `fft_adjust_sqrt2` (*C function*), 299
- `fft_butterfly` (*C function*), 299
- `fft_butterfly_sqrt2` (*C function*), 301
- `fft_butterfly_twiddle` (*C function*), 302
- `fft_combine_bits` (*C function*), 298
- `fft_combine_limbs` (*C function*), 298
- `fft_convolution` (*C function*), 305
- `fft_convolution_precache` (*C function*), 305
- `fft_mfa_truncate_sqrt2` (*C function*), 303
- `fft_mfa_truncate_sqrt2_inner` (*C function*), 303
- `fft_mfa_truncate_sqrt2_outer` (*C function*), 303
- `fft_mulmod_2expp1` (*C function*), 304
- `fft_naive_convolution_1` (*C function*), 304
- `fft_negacyclic` (*C function*), 304
- `fft_precache` (*C function*), 305
- `fft_radix2` (*C function*), 299
- `fft_radix2_twiddle` (*C function*), 302
- `fft_split_bits` (*C function*), 298
- `fft_split_limbs` (*C function*), 298
- `fft_truncate` (*C function*), 300
- `fft_truncate1` (*C function*), 300
- `fft_truncate1_twiddle` (*C function*), 302
- `fft_truncate_sqrt2` (*C function*), 302
- `Fibonacci` (*C macro*), 908
- `Fields` (*C macro*), 906
- `FiniteField` (*C macro*), 906
- `flint_abort` (*C function*), 22
- `FLINT_ABS` (*C macro*), 15
- `FLINT_BIT_COUNT` (*C function*), 278
- `flint_bitcnt_t` (*C type*), 16
- `FLINT_BITS` (*C macro*), 15
- `flint_calloc` (*C function*), 17
- `FLINT_CLOG2` (*C macro*), 278
- `flint_clz` (*C macro*), 278
- `flint_ctz` (*C macro*), 278
- `FLINT_D_BITS` (*C macro*), 15
- `flint_err_t` (*C enum*), 21
- `flint_err_t.FLINT_DIVZERO` (*C macro*), 21
- `flint_err_t.FLINT_DOMERR` (*C macro*), 21
- `flint_err_t.FLINT_ERROR` (*C macro*), 21
- `flint_err_t.FLINT_EXPOF` (*C macro*), 21
- `flint_err_t.FLINT_IMPINV` (*C macro*), 21
- `flint_err_t.FLINT_INEXACT` (*C macro*), 21
- `flint_err_t.FLINT_OVERFLOW` (*C macro*), 21
- `flint_err_t.FLINT_TEST_FAIL` (*C macro*), 22
- `FLINT_FLOG2` (*C macro*), 278
- `flint_fprintf` (*C function*), 18
- `flint_free` (*C function*), 17
- `flint_fscanf` (*C function*), 21
- `flint_get_num_available_threads` (*C function*), 13
- `flint_get_num_threads` (*C function*), 17
- `flint_malloc` (*C function*), 17
- `FLINT_MAX` (*C macro*), 15
- `flint_merge_sort` (*C function*), 22
- `FLINT_MIN` (*C macro*), 15
- `flint_mpn_addmod_2` (*C function*), 280
- `flint_mpn_addmod_n` (*C function*), 280
- `flint_mpn_addmod_n_m` (*C function*), 280
- `flint_mpn_debug` (*C function*), 279
- `flint_mpn_divides` (*C function*), 283
- `flint_mpn_divisible_1_odd` (*C function*), 282
- `flint_mpn_divrem_1_preinv` (*C function*), 283
- `flint_mpn_divrem_2_1_preinv_norm` (*C function*), 283
- `flint_mpn_divrem_2_1_preinv_unnorm` (*C function*), 283
- `flint_mpn_divrem_3_1_preinv_norm` (*C function*), 283
- `flint_mpn_divrem_3_1_preinv_unnorm` (*C function*), 283
- `flint_mpn_divrem_preinv1` (*C function*), 283
- `flint_mpn_divrem_preinvn` (*C function*), 284
- `flint_mpn_equal_p` (*C function*), 279
- `flint_mpn_factor_trial` (*C function*), 282
- `flint_mpn_factor_trial_tree` (*C function*), 282
- `flint_mpn_fmmod_precond_matrix` (*C function*), 285
- `flint_mpn_fmmod_preinvn` (*C function*), 284
- `flint_mpn_fmmod_preinvn_2` (*C function*), 284

`flint_mpn_fmms1` (*C function*), 280
`flint_mpn_gcd_full` (*C function*), 286
`flint_mpn_gcd_full2` (*C function*), 286
`flint_mpn_get_str` (*C function*), 279
`flint_mpn_mod_preinvn` (*C function*), 284
`flint_mpn_mul` (*C function*), 280
`flint_mpn_mul_fft_main` (*C function*), 305
`flint_mpn_mul_n` (*C function*), 280
`flint_mpn_mul_or_mulhigh_n` (*C function*), 282
`flint_mpn_mul_or_mulow_n` (*C function*), 282
`flint_mpn_mul_toom22` (*C function*), 280
`flint_mpn_mulhigh_n` (*C function*), 281
`flint_mpn_mulow_basecase` (*C function*), 282
`flint_mpn_mulow_n` (*C function*), 282
`flint_mpn_mulmod_precond_matrix` (*C function*), 285
`flint_mpn_mulmod_precond_matrix_alloc` (*C function*), 285
`flint_mpn_mulmod_precond_matrix_precompute` (*C function*), 285
`flint_mpn_mulmod_precond_shoup` (*C function*), 285
`flint_mpn_mulmod_precond_shoup_precompute` (*C function*), 285
`flint_mpn_mulmod_preinv1` (*C function*), 284
`flint_mpn_mulmod_preinvn` (*C function*), 284
`flint_mpn_mulmod_preinvn_2` (*C function*), 284
`flint_mpn_mulmod_want_precond` (*C function*), 285

`flint_mpn_negmod_2` (*C function*), 280
`flint_mpn_negmod_n` (*C function*), 280
`flint_mpn_preinv1` (*C function*), 283
`flint_mpn_preinvn` (*C function*), 284
`flint_mpn_remove_2exp` (*C function*), 282
`flint_mpn_remove_power_ascending` (*C function*), 282

`flint_mpn_rrandom` (*C function*), 286
`flint_mpn_rrandomb` (*C function*), 286
`flint_mpn_signed_div2` (*C function*), 283
`flint_mpn_signed_sub_n` (*C function*), 280
`flint_mpn_sqr` (*C function*), 280
`flint_mpn_sqrhigh` (*C function*), 281
`flint_mpn_submod_2` (*C function*), 280
`flint_mpn_submod_n` (*C function*), 280
`flint_mpn_submod_n_m` (*C function*), 280
`flint_mpn_sumdiff_n` (*C function*), 280
`flint_mpn_urandomb` (*C function*), 286
`flint_mpn_urandomm` (*C function*), 286
`flint_mpn_zero_p` (*C function*), 279
`flint_mpq_clear_readonly` (*C function*), 320
`flint_mpq_init_set_readonly` (*C function*), 320
`flint_mpz_clear_readonly` (*C function*), 160
`flint_mpz_init_set_readonly` (*C function*), 160
`flint_parallel_binary_splitting` (*C function*), 14
`flint_parallel_do` (*C function*), 13
`flint_printf` (*C function*), 18
`flint_rand_clear` (*C function*), 17
`flint_rand_init` (*C function*), 17
`flint_rand_struct` (*C type*), 17
`flint_rand_t` (*C type*), 17
`flint_realloc` (*C function*), 17
`flint_reset_num_workers` (*C function*), 18
`flint_scanf` (*C function*), 21
`flint_set_abort` (*C function*), 22
`flint_set_num_threads` (*C function*), 17
`flint_set_num_workers` (*C function*), 17
`flint_set_throw` (*C function*), 22
`FLINT_SGN` (*C macro*), 15
`flint_sort` (*C function*), 22
`flint_sprintf` (*C function*), 21
`flint_sscanf` (*C function*), 21
`FLINT_SWAP` (*C macro*), 15
`flint_throw` (*C function*), 22
`FLINT_UABS` (*C macro*), 15
`FLINT_VERSION` (*C macro*), 15
`flint_vfprintf` (*C function*), 18
`flint_vprintf` (*C function*), 18
`Floor` (*C macro*), 906
`mpq` (*C type*), 317
`mpq_abs` (*C function*), 318
`mpq_add` (*C function*), 322
`mpq_add_fmpz` (*C function*), 322
`mpq_add_si` (*C function*), 322
`mpq_add_ui` (*C function*), 322
`mpq_addmul` (*C function*), 322
`mpq_canonicalise` (*C function*), 318
`mpq_cfrac_bound` (*C function*), 326
`mpq_clear` (*C function*), 318
`mpq_clear_readonly` (*C function*), 321
`mpq_cmp` (*C function*), 319
`mpq_cmp_fmpz` (*C function*), 319
`mpq_cmp_si` (*C function*), 319
`mpq_cmp_ui` (*C function*), 319
`mpq_dedekind_sum` (*C function*), 326
`mpq_dedekind_sum_naive` (*C function*), 326
`mpq_denref` (*C function*), 317
`mpq_div` (*C function*), 322
`mpq_div_2exp` (*C function*), 323
`mpq_div_fmpz` (*C function*), 323
`mpq_equal` (*C function*), 319
`mpq_equal_fmpz` (*C function*), 319
`mpq_equal_si` (*C function*), 319
`mpq_equal_ui` (*C function*), 319
`mpq_farey_neighbors` (*C function*), 325
`mpq_fprint` (*C function*), 321
`mpq_gcd` (*C function*), 323
`mpq_gcd_cofactors` (*C function*), 323
`mpq_get_cfrac` (*C function*), 325
`mpq_get_cfrac_naive` (*C function*), 325
`mpq_get_d` (*C function*), 320
`mpq_get_mpfr` (*C function*), 320
`mpq_get_mpq` (*C function*), 320
`mpq_get_mpz_frac` (*C function*), 319
`mpq_get_str` (*C function*), 320
`mpq_harmonic_ui` (*C function*), 326

- `fmprq_height` (*C function*), 319
- `fmprq_height_bits` (*C function*), 319
- `fmprq_init` (*C function*), 318
- `fmprq_init_set_readonly` (*C function*), 320
- `fmprq_inv` (*C function*), 323
- `fmprq_is_canonical` (*C function*), 318
- `fmprq_is_one` (*C function*), 319
- `fmprq_is_pm1` (*C function*), 319
- `fmprq_is_zero` (*C function*), 319
- `fmprq_mat_add` (*C function*), 330
- `fmprq_mat_can_solve` (*C function*), 335
- `fmprq_mat_can_solve_fraction_free` (*C function*), 334
- `fmprq_mat_can_solve_multi_mod` (*C function*), 334
- `fmprq_mat_charpoly` (*C function*), 336
- `fmprq_mat_clear` (*C function*), 329
- `fmprq_mat_concat_horizontal` (*C function*), 331
- `fmprq_mat_concat_vertical` (*C function*), 331
- `fmprq_mat_det` (*C function*), 334
- `fmprq_mat_entry` (*C function*), 329
- `fmprq_mat_entry_den` (*C function*), 329
- `fmprq_mat_entry_num` (*C function*), 329
- `fmprq_mat_equal` (*C function*), 331
- `fmprq_mat_fmprq_vec_mul` (*C function*), 333
- `fmprq_mat_fmprq_vec_mul_ptr` (*C function*), 333
- `fmprq_mat_fmprz_vec_mul` (*C function*), 333
- `fmprq_mat_fmprz_vec_mul_ptr` (*C function*), 333
- `fmprq_mat_get_fmprz_mat` (*C function*), 332
- `fmprq_mat_get_fmprz_mat_colwise` (*C function*), 332
- `fmprq_mat_get_fmprz_mat_entrywise` (*C function*), 332
- `fmprq_mat_get_fmprz_mat_matwise` (*C function*), 332
- `fmprq_mat_get_fmprz_mat_mod_fmprz` (*C function*), 333
- `fmprq_mat_get_fmprz_mat_rowwise` (*C function*), 332
- `fmprq_mat_get_fmprz_mat_rowwise_2` (*C function*), 332
- `fmprq_mat_gso` (*C function*), 336
- `fmprq_mat_hilbert_matrix` (*C function*), 331
- `fmprq_mat_init` (*C function*), 329
- `fmprq_mat_init_set` (*C function*), 329
- `fmprq_mat_inv` (*C function*), 335
- `fmprq_mat_invert_cols` (*C function*), 330
- `fmprq_mat_invert_rows` (*C function*), 330
- `fmprq_mat_is_empty` (*C function*), 331
- `fmprq_mat_is_integral` (*C function*), 331
- `fmprq_mat_is_one` (*C function*), 331
- `fmprq_mat_is_square` (*C function*), 332
- `fmprq_mat_is_zero` (*C function*), 331
- `fmprq_mat_kronecker_product` (*C function*), 334
- `fmprq_mat_minpoly` (*C function*), 336
- `fmprq_mat_mul` (*C function*), 333
- `fmprq_mat_mul_cleared` (*C function*), 333
- `fmprq_mat_mul_direct` (*C function*), 333
- `fmprq_mat_mul_fmprq_vec` (*C function*), 333
- `fmprq_mat_mul_fmprq_vec_ptr` (*C function*), 333
- `fmprq_mat_mul_fmprz_mat` (*C function*), 333
- `fmprq_mat_mul_fmprz_vec` (*C function*), 333
- `fmprq_mat_mul_fmprz_vec_ptr` (*C function*), 333
- `fmprq_mat_mul_r_fmprz_mat` (*C function*), 333
- `fmprq_mat_ncols` (*C function*), 329
- `fmprq_mat_neg` (*C function*), 330
- `fmprq_mat_nrows` (*C function*), 329
- `fmprq_mat_one` (*C function*), 329
- `fmprq_mat_permanent` (*C function*), 334
- `fmprq_mat_pivot` (*C function*), 335
- `fmprq_mat_print` (*C function*), 330
- `fmprq_mat_randbits` (*C function*), 331
- `fmprq_mat_randtest` (*C function*), 331
- `fmprq_mat_rref` (*C function*), 335
- `fmprq_mat_rref_classical` (*C function*), 335
- `fmprq_mat_rref_fraction_free` (*C function*), 335
- `fmprq_mat_scalar_div_fmprz` (*C function*), 330
- `fmprq_mat_scalar_mul_fmprq` (*C function*), 330
- `fmprq_mat_scalar_mul_fmprz` (*C function*), 330
- `fmprq_mat_set` (*C function*), 329
- `fmprq_mat_set_fmprz_mat` (*C function*), 332
- `fmprq_mat_set_fmprz_mat_div_fmprz` (*C function*), 332
- `fmprq_mat_set_fmprz_mat_mod_fmprz` (*C function*), 333
- `fmprq_mat_similarity` (*C function*), 336
- `fmprq_mat_solve` (*C function*), 334
- `fmprq_mat_solve_dixon` (*C function*), 334
- `fmprq_mat_solve_fmprz_mat` (*C function*), 334
- `fmprq_mat_solve_fmprz_mat_dixon` (*C function*), 334
- `fmprq_mat_solve_fmprz_mat_fraction_free` (*C function*), 334
- `fmprq_mat_solve_fmprz_mat_multi_mod` (*C function*), 334
- `fmprq_mat_solve_fraction_free` (*C function*), 334
- `fmprq_mat_solve_multi_mod` (*C function*), 334
- `fmprq_mat_struct` (*C type*), 328
- `fmprq_mat_sub` (*C function*), 330
- `fmprq_mat_swap` (*C function*), 329
- `fmprq_mat_swap_cols` (*C function*), 330
- `fmprq_mat_swap_entrywise` (*C function*), 329
- `fmprq_mat_swap_rows` (*C function*), 329
- `fmprq_mat_t` (*C type*), 328
- `fmprq_mat_trace` (*C function*), 334
- `fmprq_mat_transpose` (*C function*), 329
- `fmprq_mat_window_clear` (*C function*), 331
- `fmprq_mat_window_init` (*C function*), 331
- `fmprq_mat_zero` (*C function*), 329
- `fmprq_mod_fmprz` (*C function*), 324
- `fmprq_mpoly_add` (*C function*), 368
- `fmprq_mpoly_add_fmprq` (*C function*), 367
- `fmprq_mpoly_add_fmprz` (*C function*), 367
- `fmprq_mpoly_add_si` (*C function*), 367
- `fmprq_mpoly_add_ui` (*C function*), 367

`fmpq_mpoly_clear` (*C function*), 362
`fmpq_mpoly_cmp` (*C function*), 365
`fmpq_mpoly_combine_like_terms` (*C function*), 367
`fmpq_mpoly_compose_fmpq_mpoly` (*C function*), 369
`fmpq_mpoly_compose_fmpq_mpoly_gen` (*C function*), 369
`fmpq_mpoly_compose_fmpq_poly` (*C function*), 369
`fmpq_mpoly_content` (*C function*), 370
`fmpq_mpoly_content_ref` (*C function*), 365
`fmpq_mpoly_content_vars` (*C function*), 370
`fmpq_mpoly_ctx_clear` (*C function*), 361
`fmpq_mpoly_ctx_init` (*C function*), 361
`fmpq_mpoly_ctx_nvars` (*C function*), 361
`fmpq_mpoly_ctx_ord` (*C function*), 361
`fmpq_mpoly_ctx_struct` (*C type*), 361
`fmpq_mpoly_ctx_t` (*C type*), 361
`fmpq_mpoly_degree_fmpz` (*C function*), 364
`fmpq_mpoly_degree_si` (*C function*), 364
`fmpq_mpoly_degrees_fit_si` (*C function*), 364
`fmpq_mpoly_degrees_fmpz` (*C function*), 364
`fmpq_mpoly_degrees_si` (*C function*), 364
`fmpq_mpoly_derivative` (*C function*), 369
`fmpq_mpoly_discriminant` (*C function*), 371
`fmpq_mpoly_div` (*C function*), 370
`fmpq_mpoly_divides` (*C function*), 370
`fmpq_mpoly_divrem` (*C function*), 370
`fmpq_mpoly_divrem_ideal` (*C function*), 370
`fmpq_mpoly_equal` (*C function*), 363
`fmpq_mpoly_equal_fmpq` (*C function*), 363
`fmpq_mpoly_equal_fmpz` (*C function*), 363
`fmpq_mpoly_equal_si` (*C function*), 363
`fmpq_mpoly_equal_ui` (*C function*), 363
`fmpq_mpoly_evaluate_all_fmpq` (*C function*), 369
`fmpq_mpoly_evaluate_one_fmpq` (*C function*), 369
`fmpq_mpoly_factor` (*C function*), 361
`fmpq_mpoly_factor_clear` (*C function*), 360
`fmpq_mpoly_factor_get_base` (*C function*), 360
`fmpq_mpoly_factor_get_constant_fmpq` (*C function*), 360
`fmpq_mpoly_factor_get_exp_si` (*C function*), 360
`fmpq_mpoly_factor_init` (*C function*), 360
`fmpq_mpoly_factor_length` (*C function*), 360
`fmpq_mpoly_factor_make_integral` (*C function*), 360
`fmpq_mpoly_factor_make_monic` (*C function*), 360
`fmpq_mpoly_factor_sort` (*C function*), 360
`fmpq_mpoly_factor_squarefree` (*C function*), 361
`fmpq_mpoly_factor_struct` (*C type*), 360
`fmpq_mpoly_factor_swap_base` (*C function*), 360
`fmpq_mpoly_factor_t` (*C type*), 360
`fmpq_mpoly_fit_bits` (*C function*), 362
`fmpq_mpoly_fit_length` (*C function*), 362
`fmpq_mpoly_fprint_pretty` (*C function*), 362
`fmpq_mpoly_from_univar` (*C function*), 371
`fmpq_mpoly_gcd` (*C function*), 370
`fmpq_mpoly_gcd_brown` (*C function*), 371
`fmpq_mpoly_gcd_cofactors` (*C function*), 370
`fmpq_mpoly_gcd_hensel` (*C function*), 371
`fmpq_mpoly_gcd_subresultant` (*C function*), 371
`fmpq_mpoly_gcd_zippel` (*C function*), 371
`fmpq_mpoly_gcd_zippel2` (*C function*), 371
`fmpq_mpoly_gen` (*C function*), 363
`fmpq_mpoly_get_coeff_fmpq_fmpz` (*C function*), 364
`fmpq_mpoly_get_coeff_fmpq_monomial` (*C function*), 364
`fmpq_mpoly_get_coeff_fmpq_ui` (*C function*), 364
`fmpq_mpoly_get_coeff_vars_ui` (*C function*), 364
`fmpq_mpoly_get_denominator` (*C function*), 364
`fmpq_mpoly_get_fmpq` (*C function*), 363
`fmpq_mpoly_get_str_pretty` (*C function*), 362
`fmpq_mpoly_get_term` (*C function*), 366
`fmpq_mpoly_get_term_coeff_fmpq` (*C function*), 365
`fmpq_mpoly_get_term_exp_fmpz` (*C function*), 366
`fmpq_mpoly_get_term_exp_si` (*C function*), 366
`fmpq_mpoly_get_term_exp_ui` (*C function*), 366
`fmpq_mpoly_get_term_monomial` (*C function*), 366
`fmpq_mpoly_get_term_var_exp_si` (*C function*), 366
`fmpq_mpoly_get_term_var_exp_ui` (*C function*), 366
`fmpq_mpoly_init` (*C function*), 362
`fmpq_mpoly_init2` (*C function*), 362
`fmpq_mpoly_init3` (*C function*), 362
`fmpq_mpoly_integral` (*C function*), 369
`fmpq_mpoly_is_canonical` (*C function*), 365
`fmpq_mpoly_is_fmpq` (*C function*), 363
`fmpq_mpoly_is_gen` (*C function*), 363
`fmpq_mpoly_is_one` (*C function*), 363
`fmpq_mpoly_is_square` (*C function*), 371
`fmpq_mpoly_is_zero` (*C function*), 363
`fmpq_mpoly_length` (*C function*), 365
`fmpq_mpoly_make_monic` (*C function*), 368
`fmpq_mpoly_mul` (*C function*), 369
`fmpq_mpoly_neg` (*C function*), 368
`fmpq_mpoly_one` (*C function*), 363
`fmpq_mpoly_pow_fmpz` (*C function*), 370
`fmpq_mpoly_pow_ui` (*C function*), 370
`fmpq_mpoly_print_pretty` (*C function*), 362
`fmpq_mpoly_push_term_fmpq_fmpz` (*C function*), 366
`fmpq_mpoly_push_term_fmpq_fmpz` (*C function*), 366

- fmpq_mpoly_push_term_fmpq_ui (*C function*), 366
 fmpq_mpoly_push_term_fmpz_ffmpz (*C function*), 366
 fmpq_mpoly_push_term_fmpz_fmpz (*C function*), 366
 fmpq_mpoly_push_term_fmpz_ui (*C function*), 366
 fmpq_mpoly_push_term_si_ffmpz (*C function*), 366
 fmpq_mpoly_push_term_si_fmpz (*C function*), 366
 fmpq_mpoly_push_term_si_ui (*C function*), 366
 fmpq_mpoly_push_term_ui_ffmpz (*C function*), 366
 fmpq_mpoly_push_term_ui_fmpz (*C function*), 366
 fmpq_mpoly_push_term_ui_ui (*C function*), 366
 fmpq_mpoly_randtest_bits (*C function*), 367
 fmpq_mpoly_randtest_bound (*C function*), 367
 fmpq_mpoly_randtest_bounds (*C function*), 367
 fmpq_mpoly_realloc (*C function*), 362
 fmpq_mpoly_reduce (*C function*), 367
 fmpq_mpoly_resize (*C function*), 365
 fmpq_mpoly_resultant (*C function*), 371
 fmpq_mpoly_reverse (*C function*), 367
 fmpq_mpoly_scalar_div_fmpq (*C function*), 368
 fmpq_mpoly_scalar_div_fmpz (*C function*), 368
 fmpq_mpoly_scalar_div_si (*C function*), 368
 fmpq_mpoly_scalar_div_ui (*C function*), 368
 fmpq_mpoly_scalar_mul_fmpq (*C function*), 368
 fmpq_mpoly_scalar_mul_fmpz (*C function*), 368
 fmpq_mpoly_scalar_mul_si (*C function*), 368
 fmpq_mpoly_scalar_mul_ui (*C function*), 368
 fmpq_mpoly_set (*C function*), 363
 fmpq_mpoly_set_coeff_fmpq_fmpz (*C function*), 364
 fmpq_mpoly_set_coeff_fmpq_monomial (*C function*), 364
 fmpq_mpoly_set_coeff_fmpq_ui (*C function*), 364
 fmpq_mpoly_set_fmpq (*C function*), 363
 fmpq_mpoly_set_fmpz (*C function*), 363
 fmpq_mpoly_set_si (*C function*), 363
 fmpq_mpoly_set_str_pretty (*C function*), 362
 fmpq_mpoly_set_term_coeff_fmpq (*C function*), 365
 fmpq_mpoly_set_term_exp_fmpz (*C function*), 366
 fmpq_mpoly_set_term_exp_ui (*C function*), 366
 fmpq_mpoly_set_ui (*C function*), 363
 fmpq_mpoly_sort_terms (*C function*), 367
 fmpq_mpoly_sqrt (*C function*), 371
 fmpq_mpoly_struct (*C type*), 361
 fmpq_mpoly_sub (*C function*), 368
 fmpq_mpoly_sub_fmpq (*C function*), 367
 fmpq_mpoly_sub_fmpz (*C function*), 367
 fmpq_mpoly_sub_si (*C function*), 367
 fmpq_mpoly_sub_ui (*C function*), 367
 fmpq_mpoly_swap (*C function*), 363
 fmpq_mpoly_t (*C type*), 361
 fmpq_mpoly_term_content (*C function*), 370
 fmpq_mpoly_term_exp_fits_si (*C function*), 365
 fmpq_mpoly_term_exp_fits_ui (*C function*), 365
 fmpq_mpoly_to_univar (*C function*), 371
 fmpq_mpoly_total_degree_fits_si (*C function*), 364
 fmpq_mpoly_total_degree_fmpz (*C function*), 364
 fmpq_mpoly_total_degree_si (*C function*), 364
 fmpq_mpoly_univar_clear (*C function*), 371
 fmpq_mpoly_univar_degree_fits_si (*C function*), 372
 fmpq_mpoly_univar_get_term_coeff (*C function*), 372
 fmpq_mpoly_univar_get_term_exp_si (*C function*), 372
 fmpq_mpoly_univar_init (*C function*), 371
 fmpq_mpoly_univar_length (*C function*), 372
 fmpq_mpoly_univar_swap (*C function*), 371
 fmpq_mpoly_univar_swap_term_coeff (*C function*), 372
 fmpq_mpoly_used_vars (*C function*), 364
 fmpq_mpoly_zero (*C function*), 363
 fmpq_mpoly_zpoly_ref (*C function*), 365
 fmpq_mpoly_zpoly_term_coeff_ref (*C function*), 365
 fmpq_mul (*C function*), 322
 fmpq_mul_2exp (*C function*), 323
 fmpq_mul_fmpz (*C function*), 323
 fmpq_mul_si (*C function*), 322
 fmpq_mul_ui (*C function*), 322
 fmpq_neg (*C function*), 318
 fmpq_next_calkin_wilf (*C function*), 324
 fmpq_next_minimal (*C function*), 324
 fmpq_next_signed_calkin_wilf (*C function*), 325
 fmpq_next_signed_minimal (*C function*), 324
 fmpq_numref (*C function*), 317
 fmpq_one (*C function*), 318
 fmpq_poly_add (*C function*), 342
 fmpq_poly_add_can (*C function*), 342
 fmpq_poly_add_series (*C function*), 342
 fmpq_poly_add_series_can (*C function*), 342
 fmpq_poly_addmul (*C function*), 345
 fmpq_poly_asin_series (*C function*), 352
 fmpq_poly_asinh_series (*C function*), 352
 fmpq_poly_atan_series (*C function*), 351
 fmpq_poly_atanh_series (*C function*), 352
 fmpq_poly_canonicalise (*C function*), 338
 fmpq_poly_clear (*C function*), 337
 fmpq_poly_cmp (*C function*), 341
 fmpq_poly_compose (*C function*), 355
 fmpq_poly_compose_series (*C function*), 357
 fmpq_poly_compose_series_brent_kung (*C function*), 356

fmpq_poly_compose_series_horner (*C function*), 356
 fmpq_poly_content (*C function*), 358
 fmpq_poly_cos_series (*C function*), 352
 fmpq_poly_cosh_series (*C function*), 353
 fmpq_poly_degree (*C function*), 338
 fmpq_poly_denref (*C function*), 338
 fmpq_poly_derivative (*C function*), 350
 fmpq_poly_discriminant (*C function*), 349
 fmpq_poly_div (*C function*), 346
 fmpq_poly_div_series (*C function*), 348
 fmpq_poly_divides (*C function*), 347
 fmpq_poly_divrem (*C function*), 346
 fmpq_poly_equal (*C function*), 341
 fmpq_poly_equal_trunc (*C function*), 341
 fmpq_poly_evaluate_fmpq (*C function*), 354
 fmpq_poly_evaluate_fmpz (*C function*), 354
 fmpq_poly_exp_expinv_series (*C function*), 351
 fmpq_poly_exp_series (*C function*), 351
 fmpq_poly_fit_length (*C function*), 337
 fmpq_poly_fprint (*C function*), 359
 fmpq_poly_fprint_pretty (*C function*), 359
 fmpq_poly_fread (*C function*), 359
 fmpq_poly_gcd (*C function*), 348
 fmpq_poly_gegenbauer_c (*C function*), 354
 fmpq_poly_get_coeff_fmpq (*C function*), 341
 fmpq_poly_get_coeff_fmpz (*C function*), 341
 fmpq_poly_get_denominator (*C function*), 338
 fmpq_poly_get_nmod_poly (*C function*), 339
 fmpq_poly_get_nmod_poly_den (*C function*), 339
 fmpq_poly_get_numerator (*C function*), 338
 fmpq_poly_get_slice (*C function*), 340
 fmpq_poly_get_str (*C function*), 340
 fmpq_poly_get_str_pretty (*C function*), 340
 fmpq_poly_init (*C function*), 337
 fmpq_poly_init2 (*C function*), 337
 fmpq_poly_integral (*C function*), 350
 fmpq_poly_interpolate_barycentric (*C function*), 355
 fmpq_poly_interpolate_fast (*C function*), 355
 fmpq_poly_interpolate_fmpq_vec (*C function*), 355
 fmpq_poly_interpolate_fmpz_fmpq_vec (*C function*), 355
 fmpq_poly_interpolate_fmpz_vec (*C function*), 355
 fmpq_poly_interpolate_multi_mod (*C function*), 355
 fmpq_poly_inv (*C function*), 340
 fmpq_poly_inv_series (*C function*), 348
 fmpq_poly_inv_series_newton (*C function*), 348
 fmpq_poly_invsqrt_series (*C function*), 350
 fmpq_poly_is_canonical (*C function*), 338
 fmpq_poly_is_gen (*C function*), 341
 fmpq_poly_is_monic (*C function*), 358
 fmpq_poly_is_one (*C function*), 341
 fmpq_poly_is_squarefree (*C function*), 359
 fmpq_poly_is_zero (*C function*), 341
 fmpq_poly_laguerre_l (*C function*), 354
 fmpq_poly_lcm (*C function*), 349
 fmpq_poly_legendre_p (*C function*), 354
 fmpq_poly_length (*C function*), 338
 fmpq_poly_log_series (*C function*), 351
 fmpq_poly_make_monic (*C function*), 358
 fmpq_poly_mul (*C function*), 345
 fmpq_poly_mullo (*C function*), 345
 fmpq_poly_neg (*C function*), 340
 fmpq_poly_nth_derivative (*C function*), 350
 fmpq_poly_numref (*C function*), 338
 fmpq_poly_one (*C function*), 340
 fmpq_poly_pow (*C function*), 345
 fmpq_poly_pow_trunc (*C function*), 345
 fmpq_poly_power_sums (*C function*), 351
 fmpq_poly_power_sums_to_fmpz_poly (*C function*), 351
 fmpq_poly_power_sums_to_poly (*C function*), 351
 fmpq_poly_powers_clear (*C function*), 347
 fmpq_poly_powers_precompute (*C function*), 347
 fmpq_poly_primitive_part (*C function*), 358
 fmpq_poly_print (*C function*), 359
 fmpq_poly_print_pretty (*C function*), 359
 fmpq_poly_randtest (*C function*), 339
 fmpq_poly_randtest_not_zero (*C function*), 339
 fmpq_poly_randtest_unsigned (*C function*), 339
 fmpq_poly_read (*C function*), 359
 fmpq_poly_realloc (*C function*), 337
 fmpq_poly_rem (*C function*), 346
 fmpq_poly_rem_powers_precomp (*C function*), 347
 fmpq_poly_remove (*C function*), 347
 fmpq_poly_rescale (*C function*), 356
 fmpq_poly_resultant (*C function*), 349
 fmpq_poly_resultant_div (*C function*), 349
 fmpq_poly_reverse (*C function*), 340
 fmpq_poly_revert_series (*C function*), 358
 fmpq_poly_revert_series_lagrange (*C function*), 357
 fmpq_poly_revert_series_lagrange_fast (*C function*), 357
 fmpq_poly_revert_series_newton (*C function*), 358
 fmpq_poly_scalar_div_fmpq (*C function*), 344
 fmpq_poly_scalar_div_fmpz (*C function*), 344
 fmpq_poly_scalar_div_si (*C function*), 344
 fmpq_poly_scalar_div_ui (*C function*), 344
 fmpq_poly_scalar_mul_fmpq (*C function*), 344
 fmpq_poly_scalar_mul_fmpz (*C function*), 344
 fmpq_poly_scalar_mul_si (*C function*), 344
 fmpq_poly_scalar_mul_ui (*C function*), 344
 fmpq_poly_set (*C function*), 339
 fmpq_poly_set_coeff_fmpq (*C function*), 341
 fmpq_poly_set_coeff_fmpz (*C function*), 341
 fmpq_poly_set_coeff_si (*C function*), 341
 fmpq_poly_set_coeff_ui (*C function*), 341
 fmpq_poly_set_fmpq (*C function*), 339

fmpq_poly_set_fmpz (*C function*), 339
 fmpq_poly_set_fmpz_poly (*C function*), 339
 fmpq_poly_set_nmod_poly (*C function*), 339
 fmpq_poly_set_si (*C function*), 339
 fmpq_poly_set_str (*C function*), 340
 fmpq_poly_set_trunc (*C function*), 340
 fmpq_poly_set_ui (*C function*), 339
 fmpq_poly_shift_left (*C function*), 346
 fmpq_poly_shift_right (*C function*), 346
 fmpq_poly_sin_cos_series (*C function*), 353
 fmpq_poly_sin_series (*C function*), 352
 fmpq_poly_sinh_cosh_series (*C function*), 353
 fmpq_poly_sinh_series (*C function*), 353
 fmpq_poly_sqrt_series (*C function*), 350
 fmpq_poly_struct (*C type*), 337
 fmpq_poly_sub (*C function*), 343
 fmpq_poly_sub_can (*C function*), 343
 fmpq_poly_sub_series (*C function*), 343
 fmpq_poly_sub_series_can (*C function*), 343
 fmpq_poly_submul (*C function*), 345
 fmpq_poly_swap (*C function*), 340
 fmpq_poly_t (*C type*), 337
 fmpq_poly_tan_series (*C function*), 352
 fmpq_poly_tanh_series (*C function*), 353
 fmpq_poly_truncate (*C function*), 340
 fmpq_poly_xgcd (*C function*), 349
 fmpq_poly_zero (*C function*), 340
 fmpq_pow_fmpz (*C function*), 323
 fmpq_pow_si (*C function*), 323
 fmpq_print (*C function*), 321
 fmpq_randbits (*C function*), 321
 fmpq_randtest (*C function*), 321
 fmpq_randtest_not_zero (*C function*), 321
 fmpq_reconstruct_fmpz (*C function*), 324
 fmpq_reconstruct_fmpz_2 (*C function*), 324
 fmpq_set (*C function*), 318
 fmpq_set_cfrac (*C function*), 326
 fmpq_set_fmpz_frac (*C function*), 319
 fmpq_set_mpq (*C function*), 319
 fmpq_set_si (*C function*), 319
 fmpq_set_str (*C function*), 319
 fmpq_set_ui (*C function*), 319
 fmpq_sgn (*C function*), 319
 fmpq_simplest_between (*C function*), 325
 fmpq_sub (*C function*), 322
 fmpq_sub_fmpz (*C function*), 322
 fmpq_sub_si (*C function*), 322
 fmpq_sub_ui (*C function*), 322
 fmpq_submul (*C function*), 322
 fmpq_swap (*C function*), 318
 fmpq_t (*C type*), 317
 fmpq_zero (*C function*), 318
 fmpz (*C type*), 156
 fmpz_abs (*C function*), 163
 fmpz_abs_fits_ui (*C function*), 162
 fmpz_abs_lbound_ui_2exp (*C function*), 162
 fmpz_abs_ubound_ui_2exp (*C function*), 162
 fmpz_add (*C function*), 163
 fmpz_add2_fmpz_si_inline (*C function*), 818
 fmpz_add_inline (*C function*), 818
 fmpz_add_si (*C function*), 163
 fmpz_add_si_inline (*C function*), 818
 fmpz_add_ui (*C function*), 163
 fmpz_add_ui_inline (*C function*), 818
 fmpz_addmul (*C function*), 164
 fmpz_addmul_si (*C function*), 164
 fmpz_addmul_ui (*C function*), 164
 fmpz_adev_q_2exp (*C function*), 818
 fmpz_allocated_bytes (*C function*), 818
 fmpz_and (*C function*), 170
 fmpz_bin_uiui (*C function*), 167
 fmpz_bit_pack (*C function*), 169
 fmpz_bit_unpack (*C function*), 169
 fmpz_bit_unpack_unsigned (*C function*), 169
 fmpz_bits (*C function*), 162
 fmpz_cdiv_q (*C function*), 164
 fmpz_cdiv_q_2exp (*C function*), 164
 fmpz_cdiv_q_si (*C function*), 164
 fmpz_cdiv_q_ui (*C function*), 164
 fmpz_cdiv_qr (*C function*), 164
 fmpz_cdiv_r_2exp (*C function*), 165
 fmpz_cdiv_ui (*C function*), 165
 fmpz_clear (*C function*), 157
 fmpz_clear_readonly (*C function*), 161
 fmpz_clog (*C function*), 166
 fmpz_clog_ui (*C function*), 166
 fmpz_clrbit (*C function*), 170
 fmpz_cmp (*C function*), 163
 fmpz_cmp2abs (*C function*), 163
 fmpz_cmp_si (*C function*), 163
 fmpz_cmp_ui (*C function*), 163
 fmpz_cmpabs (*C function*), 163
 fmpz_comb_clear (*C function*), 171
 fmpz_comb_init (*C function*), 171
 fmpz_comb_temp_clear (*C function*), 171
 fmpz_comb_temp_init (*C function*), 171
 fmpz_combit (*C function*), 170
 fmpz_complement (*C function*), 170
 fmpz_CRT (*C function*), 170
 fmpz_CRT_ui (*C function*), 170
 fmpz_divexact (*C function*), 165
 fmpz_divexact2_uiui (*C function*), 165
 fmpz_divexact_si (*C function*), 165
 fmpz_divexact_ui (*C function*), 165
 fmpz_divides (*C function*), 165
 fmpz_divides_mod_list (*C function*), 169
 fmpz_divisible (*C function*), 165
 fmpz_divisible_si (*C function*), 165
 fmpz_divisor_in_residue_class_lenstra (*C function*), 174
 fmpz_divisor_sigma (*C function*), 175
 fmpz_dlog (*C function*), 166
 fmpz_equal (*C function*), 163
 fmpz_equal_si (*C function*), 163
 fmpz_equal_ui (*C function*), 163
 fmpz_euler_phi (*C function*), 175

`mpz_fac_ui` (*C function*), 167
`mpz_factor` (*C function*), 181
`mpz_factor_clear` (*C function*), 180
`mpz_factor_divisor_sigma` (*C function*), 175
`mpz_factor_ecm` (*C function*), 184
`mpz_factor_ecm_add` (*C function*), 183
`mpz_factor_ecm_clear` (*C function*), 183
`mpz_factor_ecm_double` (*C function*), 183
`mpz_factor_ecm_init` (*C function*), 183
`mpz_factor_ecm_mul_montgomery_ladder` (*C function*), 183
`mpz_factor_ecm_select_curve` (*C function*), 183
`mpz_factor_ecm_stage_I` (*C function*), 183
`mpz_factor_ecm_stage_II` (*C function*), 183
`mpz_factor_euler_phi` (*C function*), 175
`mpz_factor_expand_iterative` (*C function*), 181
`mpz_factor_fprint` (*C function*), 182
`mpz_factor_init` (*C function*), 180
`mpz_factor_moebius_mu` (*C function*), 175
`mpz_factor_pollard_brent` (*C function*), 182
`mpz_factor_pollard_brent_single` (*C function*), 182
`mpz_factor_pp1` (*C function*), 182
`mpz_factor_print` (*C function*), 182
`mpz_factor_refine` (*C function*), 181
`mpz_factor_si` (*C function*), 181
`mpz_factor_smooth` (*C function*), 181
`mpz_factor_struct` (*C type*), 180
`mpz_factor_t` (*C type*), 180
`mpz_factor_trial` (*C function*), 181
`mpz_factor_trial_range` (*C function*), 181
`mpz_fdiv_q` (*C function*), 164
`mpz_fdiv_q_2exp` (*C function*), 164
`mpz_fdiv_q_si` (*C function*), 164
`mpz_fdiv_q_ui` (*C function*), 164
`mpz_fdiv_qr` (*C function*), 164
`mpz_fdiv_qr_preinvn` (*C function*), 166
`mpz_fdiv_r` (*C function*), 165
`mpz_fdiv_r_2exp` (*C function*), 165
`mpz_fdiv_ui` (*C function*), 165
`mpz_fib_ui` (*C function*), 167
`mpz_fits_si` (*C function*), 162
`mpz_flog` (*C function*), 166
`mpz_flog_ui` (*C function*), 166
`mpz_fmms` (*C function*), 164
`mpz_fmms` (*C function*), 164
`mpz_fprint` (*C function*), 161
`mpz_fread` (*C function*), 161
`mpz_gcd` (*C function*), 167
`mpz_gcd3` (*C function*), 167
`mpz_gcd_ui` (*C function*), 167
`mpz_gcdinv` (*C function*), 167
`mpz_get_d` (*C function*), 158
`mpz_get_d_2exp` (*C function*), 158
`mpz_get_mpf` (*C function*), 158
`mpz_get_mpf_r` (*C function*), 158
`mpz_get_mpn` (*C function*), 159
`FMPZ_GET_MPN_READONLY` (*C macro*), 819
`mpz_get_mpn` (*C function*), 159
`mpz_get_nmod` (*C function*), 158
`mpz_get_si` (*C function*), 158
`mpz_get_signed_ui_array` (*C function*), 159
`mpz_get_signed_uiui` (*C function*), 160
`mpz_get_str` (*C function*), 159
`mpz_get_ui` (*C function*), 158
`mpz_get_ui_array` (*C function*), 159
`mpz_get_uiui` (*C function*), 158
`mpz_init` (*C function*), 157
`mpz_init2` (*C function*), 157
`mpz_init_set` (*C function*), 157
`mpz_init_set_readonly` (*C function*), 161
`mpz_init_set_si` (*C function*), 157
`mpz_init_set_ui` (*C function*), 157
`mpz_inp_raw` (*C function*), 161
`mpz_invmod` (*C function*), 168
`mpz_is_even` (*C function*), 163
`mpz_is_odd` (*C function*), 163
`mpz_is_one` (*C function*), 163
`mpz_is_perfect_power` (*C function*), 167
`mpz_is_pm1` (*C function*), 163
`mpz_is_prime` (*C function*), 173
`mpz_is_prime_morrison` (*C function*), 173
`mpz_is_prime_pocklington` (*C function*), 172
`mpz_is_prime_pseudosquare` (*C function*), 172
`mpz_is_probabprime` (*C function*), 172
`mpz_is_probabprime_BPSW` (*C function*), 172
`mpz_is_probabprime_lucas` (*C function*), 172
`mpz_is_square` (*C function*), 166
`mpz_is_strong_probabprime` (*C function*), 172
`mpz_is_zero` (*C function*), 163
`mpz_jacobi` (*C function*), 169
`mpz_kronecker` (*C function*), 169
`mpz_lcm` (*C function*), 167
`mpz_lll` (*C function*), 206
`mpz_lll_advance_check_babai` (*C function*), 203
`mpz_lll_advance_check_babai_heuristic_d` (*C function*), 203
`mpz_lll_check_babai` (*C function*), 203
`mpz_lll_check_babai_heuristic` (*C function*), 203
`mpz_lll_check_babai_heuristic_d` (*C function*), 203
`mpz_lll_context_init` (*C function*), 202
`mpz_lll_context_init_default` (*C function*), 202
`mpz_lll_d` (*C function*), 204
`mpz_lll_d_heuristic` (*C function*), 204
`mpz_lll_d_heuristic_with_removal` (*C function*), 205
`mpz_lll_d_with_removal` (*C function*), 204
`mpz_lll_d_with_removal_knapsack` (*C function*), 205
`mpz_lll_heuristic_dot` (*C function*), 202

- `fmprz_lll_is_reduced` (*C function*), 206
- `fmprz_lll_is_reduced_d` (*C function*), 206
- `fmprz_lll_is_reduced_d_with_removal` (*C function*), 206
- `fmprz_lll_is_reduced_mprf` (*C function*), 206
- `fmprz_lll_is_reduced_mprf_with_removal` (*C function*), 206
- `fmprz_lll_is_reduced_with_removal` (*C function*), 206
- `fmprz_lll_mpf` (*C function*), 204
- `fmprz_lll_mpf2` (*C function*), 204
- `fmprz_lll_mpf2_with_removal` (*C function*), 205
- `fmprz_lll_mpf_with_removal` (*C function*), 205
- `fmprz_lll_randtest` (*C function*), 202
- `fmprz_lll_shift` (*C function*), 204
- `fmprz_lll_storjohann_ulll` (*C function*), 206
- `fmprz_lll_with_removal` (*C function*), 206
- `fmprz_lll_with_removal_ulll` (*C function*), 205
- `fmprz_lll_wrapper` (*C function*), 204
- `fmprz_lll_wrapper_with_removal` (*C function*), 205
- `fmprz_lll_wrapper_with_removal_knapsack` (*C function*), 205
- `fmprz_lshift_mpn` (*C function*), 819
- `fmprz_lucas_chain` (*C function*), 173
- `fmprz_lucas_chain_add` (*C function*), 174
- `fmprz_lucas_chain_double` (*C function*), 174
- `fmprz_lucas_chain_full` (*C function*), 174
- `fmprz_lucas_chain_mul` (*C function*), 174
- `fmprz_lucas_chain_VtoU` (*C function*), 174
- `fmprz_mat_add` (*C function*), 190
- `fmprz_mat_can_solve` (*C function*), 197
- `fmprz_mat_can_solve_fflu` (*C function*), 196
- `fmprz_mat_can_solve_multi_mod_den` (*C function*), 196
- `fmprz_mat_charpoly` (*C function*), 194
- `fmprz_mat_charpoly_berkowitz` (*C function*), 194
- `fmprz_mat_charpoly_modular` (*C function*), 194
- `fmprz_mat_chol_d` (*C function*), 201
- `fmprz_mat_clear` (*C function*), 185
- `fmprz_mat_col_partition` (*C function*), 195
- `fmprz_mat_concat_horizontal` (*C function*), 189
- `fmprz_mat_concat_vertical` (*C function*), 189
- `fmprz_mat_content` (*C function*), 192
- `fmprz_mat_CRT_ui` (*C function*), 189
- `fmprz_mat_det` (*C function*), 193
- `fmprz_mat_det_bareiss` (*C function*), 193
- `fmprz_mat_det_bound` (*C function*), 193
- `fmprz_mat_det_bound_nonzero` (*C function*), 193
- `fmprz_mat_det_cofactor` (*C function*), 193
- `fmprz_mat_det_divisor` (*C function*), 193
- `fmprz_mat_det_modular` (*C function*), 193
- `fmprz_mat_det_modular_accelerated` (*C function*), 193
- `fmprz_mat_det_modular_given_divisor` (*C function*), 193
- `fmprz_mat_entry` (*C function*), 185
- `fmprz_mat_equal` (*C function*), 188
- `fmprz_mat_equal_col` (*C function*), 188
- `fmprz_mat_equal_row` (*C function*), 188
- `fmprz_mat_fflu` (*C function*), 197
- `fmprz_mat_find_pivot_any` (*C function*), 197
- `fmprz_mat_fmprz_vec_mul` (*C function*), 192
- `fmprz_mat_fmprz_vec_mul_ptr` (*C function*), 192
- `fmprz_mat_fprint` (*C function*), 188
- `fmprz_mat_fprint_pretty` (*C function*), 188
- `fmprz_mat_fread` (*C function*), 188
- `fmprz_mat_get_d_mat` (*C function*), 201
- `fmprz_mat_get_d_mat_transpose` (*C function*), 201
- `fmprz_mat_get_nmod_mat` (*C function*), 189
- `fmprz_mat_gram` (*C function*), 200
- `fmprz_mat_hadamard` (*C function*), 200
- `fmprz_mat_hnf` (*C function*), 199
- `fmprz_mat_hnf_classical` (*C function*), 199
- `fmprz_mat_hnf_minors` (*C function*), 199
- `fmprz_mat_hnf_modular` (*C function*), 199
- `fmprz_mat_hnf_modular_eldiv` (*C function*), 199
- `fmprz_mat_hnf_pernet_stein` (*C function*), 199
- `fmprz_mat_hnf_transform` (*C function*), 199
- `fmprz_mat_hnf_xgcd` (*C function*), 199
- `fmprz_mat_howell_form_mod` (*C function*), 198
- `fmprz_mat_init` (*C function*), 185
- `fmprz_mat_init_set` (*C function*), 185
- `fmprz_mat_inv` (*C function*), 192
- `fmprz_mat_invert_cols` (*C function*), 186
- `fmprz_mat_invert_rows` (*C function*), 186
- `fmprz_mat_is_empty` (*C function*), 188
- `fmprz_mat_is_hadamard` (*C function*), 200
- `fmprz_mat_is_in_hnf` (*C function*), 199
- `fmprz_mat_is_in_rref_with_rank` (*C function*), 198
- `fmprz_mat_is_in_snf` (*C function*), 200
- `fmprz_mat_is_one` (*C function*), 188
- `fmprz_mat_is_reduced` (*C function*), 201
- `fmprz_mat_is_reduced_gram` (*C function*), 201
- `fmprz_mat_is_reduced_gram_with_removal` (*C function*), 201
- `fmprz_mat_is_reduced_with_removal` (*C function*), 201
- `fmprz_mat_is_spd` (*C function*), 201
- `fmprz_mat_is_square` (*C function*), 188
- `fmprz_mat_is_zero` (*C function*), 188
- `fmprz_mat_is_zero_row` (*C function*), 188
- `fmprz_mat_kronecker_product` (*C function*), 192
- `fmprz_mat_lll_original` (*C function*), 201
- `fmprz_mat_lll_storjohann` (*C function*), 202
- `fmprz_mat_minpoly` (*C function*), 195
- `fmprz_mat_minpoly_modular` (*C function*), 194
- `fmprz_mat_mul` (*C function*), 191
- `fmprz_mat_mul_blas` (*C function*), 191
- `fmprz_mat_mul_classical` (*C function*), 191
- `fmprz_mat_mul_fft` (*C function*), 191
- `fmprz_mat_mul_fmprz_vec` (*C function*), 192
- `fmprz_mat_mul_fmprz_vec_ptr` (*C function*), 192
- `fmprz_mat_mul_multi_mod` (*C function*), 191

`fmpr_mat_mul_strassen` (*C function*), 191
`fmpr_mat_mul_waksman` (*C function*), 191
`fmpr_mat_multi_CRT_ui` (*C function*), 189
`fmpr_mat_multi_CRT_ui_precomp` (*C function*), 189
`fmpr_mat_multi_mod_ui` (*C function*), 189
`fmpr_mat_multi_mod_ui_precomp` (*C function*), 189
`fmpr_mat_ncols` (*C function*), 185
`fmpr_mat_neg` (*C function*), 190
`fmpr_mat_nrows` (*C function*), 185
`fmpr_mat_nullspace` (*C function*), 198
`fmpr_mat_one` (*C function*), 186
`fmpr_mat_permanent` (*C function*), 194
`fmpr_mat_pow` (*C function*), 191
`fmpr_mat_print` (*C function*), 188
`fmpr_mat_print_pretty` (*C function*), 188
`fmpr_mat_randajtai` (*C function*), 187
`fmpr_mat_randbits` (*C function*), 186
`fmpr_mat_randedet` (*C function*), 187
`fmpr_mat_randintrel` (*C function*), 186
`fmpr_mat_randntrulike` (*C function*), 187
`fmpr_mat_randntrulike2` (*C function*), 187
`fmpr_mat_randops` (*C function*), 187
`fmpr_mat_randpermdiag` (*C function*), 187
`fmpr_mat_randrank` (*C function*), 187
`fmpr_mat_randsimdioph` (*C function*), 186
`fmpr_mat_randtest` (*C function*), 186
`fmpr_mat_rank` (*C function*), 195
`fmpr_mat_read` (*C function*), 188
`fmpr_mat_rref` (*C function*), 197
`fmpr_mat_rref_fflu` (*C function*), 197
`fmpr_mat_rref_fraction_free` (*C function*), 198
`fmpr_mat_rref_mul` (*C function*), 198
`fmpr_mat_scalar_addmul_fmpr` (*C function*), 190
`fmpr_mat_scalar_addmul_nmod_mat_fmpr` (*C function*), 190
`fmpr_mat_scalar_addmul_nmod_mat_ui` (*C function*), 190
`fmpr_mat_scalar_addmul_si` (*C function*), 190
`fmpr_mat_scalar_addmul_ui` (*C function*), 190
`fmpr_mat_scalar_divexact_fmpr` (*C function*), 190
`fmpr_mat_scalar_divexact_si` (*C function*), 190
`fmpr_mat_scalar_divexact_ui` (*C function*), 190
`fmpr_mat_scalar_mul_2exp` (*C function*), 190
`fmpr_mat_scalar_mul_fmpr` (*C function*), 190
`fmpr_mat_scalar_mul_si` (*C function*), 190
`fmpr_mat_scalar_mul_ui` (*C function*), 190
`fmpr_mat_scalar_smod` (*C function*), 190
`fmpr_mat_scalar_submul_fmpr` (*C function*), 190
`fmpr_mat_scalar_submul_si` (*C function*), 190
`fmpr_mat_scalar_submul_ui` (*C function*), 190
`fmpr_mat_scalar_tdiv_q_2exp` (*C function*), 190
`fmpr_mat_set` (*C function*), 185
`fmpr_mat_set_nmod_mat` (*C function*), 189
`fmpr_mat_set_nmod_mat_unsigned` (*C function*), 189
`fmpr_mat_similarity` (*C function*), 194
`fmpr_mat_snf` (*C function*), 200
`fmpr_mat_snf_diagonal` (*C function*), 200
`fmpr_mat_snf_iliopoulos` (*C function*), 200
`fmpr_mat_snf_kannan_bachem` (*C function*), 200
`fmpr_mat_solve` (*C function*), 195
`fmpr_mat_solve_bound` (*C function*), 196
`fmpr_mat_solve_cramer` (*C function*), 196
`fmpr_mat_solve_dixon` (*C function*), 196
`fmpr_mat_solve_dixon_den` (*C function*), 196
`fmpr_mat_solve_fflu` (*C function*), 195
`fmpr_mat_solve_fflu_precomp` (*C function*), 195
`fmpr_mat_solve_multi_mod_den` (*C function*), 196
`fmpr_mat_sqr` (*C function*), 191
`fmpr_mat_sqr_bodrato` (*C function*), 191
`fmpr_mat_strong_echelon_form_mod` (*C function*), 198
`fmpr_mat_struct` (*C type*), 185
`fmpr_mat_sub` (*C function*), 190
`fmpr_mat_swap` (*C function*), 185
`fmpr_mat_swap_cols` (*C function*), 186
`fmpr_mat_swap_entrywise` (*C function*), 185
`fmpr_mat_swap_rows` (*C function*), 186
`fmpr_mat_t` (*C type*), 185
`fmpr_mat_trace` (*C function*), 193
`fmpr_mat_transpose` (*C function*), 189
`fmpr_mat_window_clear` (*C function*), 186
`fmpr_mat_window_init` (*C function*), 186
`fmpr_mat_zero` (*C function*), 185
`fmpr_max` (*C function*), 818
`fmpr_min` (*C function*), 818
`fmpr_mod` (*C function*), 165
`fmpr_mod_add` (*C function*), 460
`fmpr_mod_add_fmpr` (*C function*), 460
`fmpr_mod_add_si` (*C function*), 460
`fmpr_mod_add_ui` (*C function*), 460
`fmpr_mod_berlekamp_massey_add_point` (*C function*), 497
`fmpr_mod_berlekamp_massey_add_points` (*C function*), 497
`fmpr_mod_berlekamp_massey_add_zeros` (*C function*), 497
`fmpr_mod_berlekamp_massey_clear` (*C function*), 496
`fmpr_mod_berlekamp_massey_init` (*C function*), 496
`fmpr_mod_berlekamp_massey_point_count` (*C function*), 497
`fmpr_mod_berlekamp_massey_points` (*C function*), 497
`fmpr_mod_berlekamp_massey_R_poly` (*C function*), 497
`fmpr_mod_berlekamp_massey_reduce` (*C function*), 497
`fmpr_mod_berlekamp_massey_start_over` (*C function*), 496

`fmpz_mod_berlekamp_massey_V_poly` (*C function*), 497
`fmpz_mod_ctx_clear` (*C function*), 460
`fmpz_mod_ctx_init` (*C function*), 460
`fmpz_mod_ctx_set_modulus` (*C function*), 460
`fmpz_mod_ctx_struct` (*C type*), 460
`fmpz_mod_ctx_t` (*C type*), 460
`fmpz_mod_discrete_log_pohlig_hellman_clear` (*C function*), 461
`fmpz_mod_discrete_log_pohlig_hellman_init` (*C function*), 461
`fmpz_mod_discrete_log_pohlig_hellman_precomp` (*C function*), 461
`fmpz_mod_discrete_log_pohlig_hellman_primitive` (*C function*), 461
`fmpz_mod_discrete_log_pohlig_hellman_run` (*C function*), 461
`fmpz_mod_divides` (*C function*), 461
`fmpz_mod_fmpz_sub` (*C function*), 460
`fmpz_mod_inv` (*C function*), 461
`fmpz_mod_is_canonical` (*C function*), 460
`fmpz_mod_is_one` (*C function*), 460
`fmpz_mod_mat_add` (*C function*), 465
`fmpz_mod_mat_can_solve` (*C function*), 468
`fmpz_mod_mat_charpoly` (*C function*), 469
`fmpz_mod_mat_clear` (*C function*), 463
`fmpz_mod_mat_concat_horizontal` (*C function*), 464
`fmpz_mod_mat_concat_vertical` (*C function*), 464
`fmpz_mod_mat_det` (*C function*), 467
`fmpz_mod_mat_entry` (*C function*), 463
`fmpz_mod_mat_equal` (*C function*), 465
`fmpz_mod_mat_fmpz_vec_mul` (*C function*), 466
`fmpz_mod_mat_fmpz_vec_mul_ptr` (*C function*), 466
`fmpz_mod_mat_get_fmpz_mat` (*C function*), 465
`fmpz_mod_mat_howell_form` (*C function*), 467
`fmpz_mod_mat_init` (*C function*), 463
`fmpz_mod_mat_init_set` (*C function*), 463
`fmpz_mod_mat_inv` (*C function*), 467
`fmpz_mod_mat_is_empty` (*C function*), 464
`fmpz_mod_mat_is_square` (*C function*), 464
`fmpz_mod_mat_is_zero` (*C function*), 465
`fmpz_mod_mat_lu` (*C function*), 468
`fmpz_mod_mat_minpoly` (*C function*), 469
`fmpz_mod_mat_mul` (*C function*), 466
`fmpz_mod_mat_mul_classical_threaded` (*C function*), 466
`fmpz_mod_mat_mul_fmpz_vec` (*C function*), 466
`fmpz_mod_mat_mul_fmpz_vec_ptr` (*C function*), 466
`fmpz_mod_mat_ncols` (*C function*), 463
`fmpz_mod_mat_neg` (*C function*), 465
`fmpz_mod_mat_nrows` (*C function*), 463
`fmpz_mod_mat_one` (*C function*), 463
`fmpz_mod_mat_pow_ui` (*C function*), 466
`fmpz_mod_mat_print_pretty` (*C function*), 465
`fmpz_mod_mat_rand` (*C function*), 464
`fmpz_mod_mat_randfull` (*C function*), 464
`fmpz_mod_mat_randtest` (*C function*), 464
`fmpz_mod_mat_rref` (*C function*), 467
`fmpz_mod_mat_scalar_mul_fmpz` (*C function*), 466
`fmpz_mod_mat_scalar_mul_si` (*C function*), 466
`fmpz_mod_mat_scalar_mul_ui` (*C function*), 466
`fmpz_mod_mat_set` (*C function*), 465
`fmpz_mod_mat_set_entry` (*C function*), 463
`fmpz_mod_mat_set_fmpz_mat` (*C function*), 465
`fmpz_mod_mat_similarity` (*C function*), 469
`fmpz_mod_mat_solve` (*C function*), 468
`fmpz_mod_mat_solve_tril` (*C function*), 468
`fmpz_mod_mat_solve_triu` (*C function*), 468
`fmpz_mod_mat_sqr` (*C function*), 466
`fmpz_mod_mat_strong_echelon_form` (*C function*), 467
`fmpz_mod_mat_struct` (*C type*), 463
`fmpz_mod_mat_sub` (*C function*), 465
`fmpz_mod_mat_swap` (*C function*), 463
`fmpz_mod_mat_swap_entrywise` (*C function*), 463
`fmpz_mod_mat_t` (*C type*), 463
`fmpz_mod_mat_trace` (*C function*), 467
`fmpz_mod_mat_transpose` (*C function*), 465
`fmpz_mod_mat_window_clear` (*C function*), 464
`fmpz_mod_mat_window_init` (*C function*), 464
`fmpz_mod_mat_zero` (*C function*), 463
`fmpz_mod_mpoly_add` (*C function*), 507
`fmpz_mod_mpoly_add_fmpz` (*C function*), 507
`fmpz_mod_mpoly_add_si` (*C function*), 507
`fmpz_mod_mpoly_add_ui` (*C function*), 507
`fmpz_mod_mpoly_buchberger_naive` (*C function*), 513
`fmpz_mod_mpoly_buchberger_naive_with_limits` (*C function*), 513
`fmpz_mod_mpoly_clear` (*C function*), 501
`fmpz_mod_mpoly_cmp` (*C function*), 504
`fmpz_mod_mpoly_combine_like_terms` (*C function*), 506
`fmpz_mod_mpoly_compose_fmpz_mod_mpoly` (*C function*), 508
`fmpz_mod_mpoly_compose_fmpz_mod_mpoly_gen` (*C function*), 508
`fmpz_mod_mpoly_compose_fmpz_mod_mpoly_geobucket` (*C function*), 508
`fmpz_mod_mpoly_compose_fmpz_mod_mpoly_horner` (*C function*), 508
`fmpz_mod_mpoly_compose_fmpz_poly` (*C function*), 508
`fmpz_mod_mpoly_content_vars` (*C function*), 510
`fmpz_mod_mpoly_ctx_clear` (*C function*), 501
`fmpz_mod_mpoly_ctx_get_modulus` (*C function*), 501
`fmpz_mod_mpoly_ctx_init` (*C function*), 501
`fmpz_mod_mpoly_ctx_nvars` (*C function*), 501
`fmpz_mod_mpoly_ctx_ord` (*C function*), 501
`fmpz_mod_mpoly_ctx_struct` (*C type*), 500

- `fmpz_mod_mpoly_ctx_t` (*C type*), 500
- `fmpz_mod_mpoly_deflate` (*C function*), 514
- `fmpz_mod_mpoly_deflation` (*C function*), 514
- `fmpz_mod_mpoly_degree_fmpz` (*C function*), 503
- `fmpz_mod_mpoly_degree_si` (*C function*), 503
- `fmpz_mod_mpoly_degrees_fit_si` (*C function*), 503
- `fmpz_mod_mpoly_degrees_fmpz` (*C function*), 503
- `fmpz_mod_mpoly_degrees_si` (*C function*), 503
- `fmpz_mod_mpoly_derivative` (*C function*), 508
- `fmpz_mod_mpoly_discriminant` (*C function*), 510
- `fmpz_mod_mpoly_div` (*C function*), 509
- `fmpz_mod_mpoly_divides` (*C function*), 509
- `fmpz_mod_mpoly_divrem` (*C function*), 509
- `fmpz_mod_mpoly_divrem_ideal` (*C function*), 509
- `fmpz_mod_mpoly_equal` (*C function*), 502
- `fmpz_mod_mpoly_equal_fmpz` (*C function*), 502
- `fmpz_mod_mpoly_equal_si` (*C function*), 502
- `fmpz_mod_mpoly_equal_ui` (*C function*), 502
- `fmpz_mod_mpoly_evaluate_all_fmpz` (*C function*), 508
- `fmpz_mod_mpoly_evaluate_one_fmpz` (*C function*), 508
- `fmpz_mod_mpoly_factor` (*C function*), 515
- `fmpz_mod_mpoly_factor_clear` (*C function*), 514
- `fmpz_mod_mpoly_factor_get_base` (*C function*), 515
- `fmpz_mod_mpoly_factor_get_constant_fmpz` (*C function*), 515
- `fmpz_mod_mpoly_factor_get_exp_si` (*C function*), 515
- `fmpz_mod_mpoly_factor_init` (*C function*), 514
- `fmpz_mod_mpoly_factor_length` (*C function*), 515
- `fmpz_mod_mpoly_factor_sort` (*C function*), 515
- `fmpz_mod_mpoly_factor_squarefree` (*C function*), 515
- `fmpz_mod_mpoly_factor_struct` (*C type*), 514
- `fmpz_mod_mpoly_factor_swap` (*C function*), 515
- `fmpz_mod_mpoly_factor_swap_base` (*C function*), 515
- `fmpz_mod_mpoly_factor_t` (*C type*), 514
- `fmpz_mod_mpoly_fprint_pretty` (*C function*), 501
- `fmpz_mod_mpoly_from_univar` (*C function*), 511
- `fmpz_mod_mpoly_gcd` (*C function*), 510
- `fmpz_mod_mpoly_gcd_brown` (*C function*), 510
- `fmpz_mod_mpoly_gcd_cofactors` (*C function*), 510
- `fmpz_mod_mpoly_gcd_hensel` (*C function*), 510
- `fmpz_mod_mpoly_gcd_subresultant` (*C function*), 510
- `fmpz_mod_mpoly_gcd_zippel` (*C function*), 510
- `fmpz_mod_mpoly_gcd_zippel2` (*C function*), 510
- `fmpz_mod_mpoly_gen` (*C function*), 502
- `fmpz_mod_mpoly_get_coeff_fmpz_fmpz` (*C function*), 504
- `fmpz_mod_mpoly_get_coeff_fmpz_monomial` (*C function*), 503
- `fmpz_mod_mpoly_get_coeff_fmpz_ui` (*C function*), 504
- `fmpz_mod_mpoly_get_coeff_vars_ui` (*C function*), 504
- `fmpz_mod_mpoly_get_fmpz` (*C function*), 502
- `fmpz_mod_mpoly_get_fmpz_mpoly` (*C function*), 514
- `fmpz_mod_mpoly_get_str_pretty` (*C function*), 501
- `fmpz_mod_mpoly_get_term` (*C function*), 505
- `fmpz_mod_mpoly_get_term_coeff_fmpz` (*C function*), 505
- `fmpz_mod_mpoly_get_term_exp_fmpz` (*C function*), 505
- `fmpz_mod_mpoly_get_term_exp_si` (*C function*), 505
- `fmpz_mod_mpoly_get_term_exp_ui` (*C function*), 505
- `fmpz_mod_mpoly_get_term_monomial` (*C function*), 505
- `fmpz_mod_mpoly_get_term_var_exp_si` (*C function*), 505
- `fmpz_mod_mpoly_get_term_var_exp_ui` (*C function*), 505
- `fmpz_mod_mpoly_inflate` (*C function*), 514
- `fmpz_mod_mpoly_init` (*C function*), 501
- `fmpz_mod_mpoly_init2` (*C function*), 501
- `fmpz_mod_mpoly_init3` (*C function*), 501
- `fmpz_mod_mpoly_is_canonical` (*C function*), 504
- `fmpz_mod_mpoly_is_fmpz` (*C function*), 502
- `fmpz_mod_mpoly_is_gen` (*C function*), 502
- `fmpz_mod_mpoly_is_one` (*C function*), 503
- `fmpz_mod_mpoly_is_square` (*C function*), 510
- `fmpz_mod_mpoly_is_zero` (*C function*), 503
- `fmpz_mod_mpoly_length` (*C function*), 504
- `fmpz_mod_mpoly_make_monic` (*C function*), 507
- `fmpz_mod_mpoly_mul` (*C function*), 509
- `fmpz_mod_mpoly_mul_dense` (*C function*), 509
- `fmpz_mod_mpoly_mul_johnson` (*C function*), 509
- `fmpz_mod_mpoly_neg` (*C function*), 507
- `fmpz_mod_mpoly_one` (*C function*), 502
- `fmpz_mod_mpoly_pow_fmpz` (*C function*), 509
- `fmpz_mod_mpoly_pow_ui` (*C function*), 509
- `fmpz_mod_mpoly_print_pretty` (*C function*), 501
- `fmpz_mod_mpoly_push_term_fmpz_ffmpz` (*C function*), 505
- `fmpz_mod_mpoly_push_term_fmpz_fmpz` (*C function*), 505
- `fmpz_mod_mpoly_push_term_fmpz_ui` (*C function*), 505
- `fmpz_mod_mpoly_push_term_si_ffmpz` (*C function*), 505
- `fmpz_mod_mpoly_push_term_si_fmpz` (*C function*), 505
- `fmpz_mod_mpoly_push_term_si_ui` (*C function*), 505

`fmpz_mod_mpoly_push_term_ui_ffmpz` (*C function*), 505
`fmpz_mod_mpoly_push_term_ui_fmpz` (*C function*), 505
`fmpz_mod_mpoly_push_term_ui_ui` (*C function*), 505
`fmpz_mod_mpoly_q_add` (*C function*), 518
`fmpz_mod_mpoly_q_add_fmpq` (*C function*), 518
`fmpz_mod_mpoly_q_add_fmpz` (*C function*), 518
`fmpz_mod_mpoly_q_add_fmpz_mod` (*C function*), 518
`fmpz_mod_mpoly_q_add_si` (*C function*), 518
`fmpz_mod_mpoly_q_canonicalise` (*C function*), 516
`fmpz_mod_mpoly_q_clear` (*C function*), 516
`fmpz_mod_mpoly_q_denref` (*C macro*), 516
`fmpz_mod_mpoly_q_div` (*C function*), 519
`fmpz_mod_mpoly_q_div_fmpq` (*C function*), 519
`fmpz_mod_mpoly_q_div_fmpz` (*C function*), 519
`fmpz_mod_mpoly_q_div_si` (*C function*), 519
`fmpz_mod_mpoly_q_equal` (*C function*), 518
`fmpz_mod_mpoly_q_gen` (*C function*), 517
`fmpz_mod_mpoly_q_get_str_pretty` (*C function*), 517
`fmpz_mod_mpoly_q_init` (*C function*), 516
`fmpz_mod_mpoly_q_inv` (*C function*), 519
`fmpz_mod_mpoly_q_is_canonical` (*C function*), 516
`fmpz_mod_mpoly_q_is_one` (*C function*), 517
`fmpz_mod_mpoly_q_is_zero` (*C function*), 517
`fmpz_mod_mpoly_q_mul` (*C function*), 518
`fmpz_mod_mpoly_q_mul_fmpq` (*C function*), 518
`fmpz_mod_mpoly_q_mul_fmpz` (*C function*), 518
`fmpz_mod_mpoly_q_mul_fmpz_mod` (*C function*), 518
`fmpz_mod_mpoly_q_mul_si` (*C function*), 518
`fmpz_mod_mpoly_q_neg` (*C function*), 518
`fmpz_mod_mpoly_q_numref` (*C macro*), 516
`fmpz_mod_mpoly_q_one` (*C function*), 517
`fmpz_mod_mpoly_q_print_pretty` (*C function*), 517
`fmpz_mod_mpoly_q_randtest` (*C function*), 518
`fmpz_mod_mpoly_q_set` (*C function*), 516
`fmpz_mod_mpoly_q_set_fmpq` (*C function*), 516
`fmpz_mod_mpoly_q_set_fmpz` (*C function*), 516
`fmpz_mod_mpoly_q_set_si` (*C function*), 516
`fmpz_mod_mpoly_q_set_str_pretty` (*C function*), 517
`fmpz_mod_mpoly_q_struct` (*C type*), 516
`fmpz_mod_mpoly_q_sub` (*C function*), 518
`fmpz_mod_mpoly_q_sub_fmpq` (*C function*), 518
`fmpz_mod_mpoly_q_sub_fmpz` (*C function*), 518
`fmpz_mod_mpoly_q_sub_fmpz_mod` (*C function*), 518
`fmpz_mod_mpoly_q_sub_si` (*C function*), 518
`fmpz_mod_mpoly_q_swap` (*C function*), 516
`fmpz_mod_mpoly_q_t` (*C type*), 516
`fmpz_mod_mpoly_q_used_vars` (*C function*), 517
`fmpz_mod_mpoly_q_used_vars_den` (*C function*), 517
`fmpz_mod_mpoly_q_used_vars_num` (*C function*), 517
`fmpz_mod_mpoly_q_zero` (*C function*), 517
`fmpz_mod_mpoly_quadratic_root` (*C function*), 510
`fmpz_mod_mpoly_randtest_bits` (*C function*), 506
`fmpz_mod_mpoly_randtest_bound` (*C function*), 506
`fmpz_mod_mpoly_randtest_bounds` (*C function*), 506
`fmpz_mod_mpoly_reduction_monic_part` (*C function*), 513
`fmpz_mod_mpoly_resize` (*C function*), 504
`fmpz_mod_mpoly_resultant` (*C function*), 510
`fmpz_mod_mpoly_reverse` (*C function*), 506
`fmpz_mod_mpoly_scalar_addmul_fmpz` (*C function*), 507
`fmpz_mod_mpoly_scalar_mul_fmpz` (*C function*), 507
`fmpz_mod_mpoly_scalar_mul_si` (*C function*), 507
`fmpz_mod_mpoly_scalar_mul_ui` (*C function*), 507
`fmpz_mod_mpoly_set` (*C function*), 502
`fmpz_mod_mpoly_set_coeff_fmpz_fmpz` (*C function*), 504
`fmpz_mod_mpoly_set_coeff_fmpz_monomial` (*C function*), 503
`fmpz_mod_mpoly_set_coeff_fmpz_ui` (*C function*), 504
`fmpz_mod_mpoly_set_coeff_si_fmpz` (*C function*), 504
`fmpz_mod_mpoly_set_coeff_si_ui` (*C function*), 504
`fmpz_mod_mpoly_set_coeff_ui_fmpz` (*C function*), 504
`fmpz_mod_mpoly_set_coeff_ui_ui` (*C function*), 504
`fmpz_mod_mpoly_set_fmpz` (*C function*), 502
`fmpz_mod_mpoly_set_fmpz_mpoly` (*C function*), 514
`fmpz_mod_mpoly_set_si` (*C function*), 502
`fmpz_mod_mpoly_set_str_pretty` (*C function*), 501
`fmpz_mod_mpoly_set_term_coeff_fmpz` (*C function*), 505
`fmpz_mod_mpoly_set_term_coeff_si` (*C function*), 505
`fmpz_mod_mpoly_set_term_coeff_ui` (*C function*), 505
`fmpz_mod_mpoly_set_term_exp_fmpz` (*C function*), 505
`fmpz_mod_mpoly_set_term_exp_ui` (*C function*), 505
`fmpz_mod_mpoly_set_ui` (*C function*), 502

- `fmpz_mod_mpoly_sort_terms` (*C function*), 506
- `fmpz_mod_mpoly_spoly` (*C function*), 513
- `fmpz_mod_mpoly_sqrt` (*C function*), 510
- `fmpz_mod_mpoly_struct` (*C type*), 500
- `fmpz_mod_mpoly_sub` (*C function*), 507
- `fmpz_mod_mpoly_sub_fmpz` (*C function*), 507
- `fmpz_mod_mpoly_sub_si` (*C function*), 507
- `fmpz_mod_mpoly_sub_ui` (*C function*), 507
- `fmpz_mod_mpoly_swap` (*C function*), 502
- `fmpz_mod_mpoly_t` (*C type*), 500
- `fmpz_mod_mpoly_term_content` (*C function*), 510
- `fmpz_mod_mpoly_term_exp_fits_si` (*C function*), 505
- `fmpz_mod_mpoly_term_exp_fits_ui` (*C function*), 505
- `fmpz_mod_mpoly_to_univar` (*C function*), 511
- `fmpz_mod_mpoly_total_degree_fits_si` (*C function*), 503
- `fmpz_mod_mpoly_total_degree_fmpz` (*C function*), 503
- `fmpz_mod_mpoly_total_degree_si` (*C function*), 503
- `fmpz_mod_mpoly_univar_clear` (*C function*), 511
- `fmpz_mod_mpoly_univar_degree_fits_si` (*C function*), 511
- `fmpz_mod_mpoly_univar_discriminant` (*C function*), 511
- `fmpz_mod_mpoly_univar_get_term_coeff` (*C function*), 511
- `fmpz_mod_mpoly_univar_get_term_exp_si` (*C function*), 511
- `fmpz_mod_mpoly_univar_init` (*C function*), 511
- `fmpz_mod_mpoly_univar_length` (*C function*), 511
- `fmpz_mod_mpoly_univar_resultant` (*C function*), 511
- `fmpz_mod_mpoly_univar_set_coeff_ui` (*C function*), 511
- `fmpz_mod_mpoly_univar_swap` (*C function*), 511
- `fmpz_mod_mpoly_univar_swap_term_coeff` (*C function*), 511
- `fmpz_mod_mpoly_used_vars` (*C function*), 503
- `fmpz_mod_mpoly_vec_append` (*C function*), 512
- `fmpz_mod_mpoly_vec_autoreduction` (*C function*), 513
- `fmpz_mod_mpoly_vec_autoreduction_groebner` (*C function*), 513
- `fmpz_mod_mpoly_vec_clear` (*C function*), 512
- `fmpz_mod_mpoly_vec_entry` (*C macro*), 512
- `fmpz_mod_mpoly_vec_fit_length` (*C function*), 512
- `fmpz_mod_mpoly_vec_init` (*C function*), 512
- `fmpz_mod_mpoly_vec_insert_unique` (*C function*), 512
- `fmpz_mod_mpoly_vec_is_autoreduced` (*C function*), 513
- `fmpz_mod_mpoly_vec_is_groebner` (*C function*), 513
- `fmpz_mod_mpoly_vec_print` (*C function*), 512
- `fmpz_mod_mpoly_vec_randtest_not_zero` (*C function*), 512
- `fmpz_mod_mpoly_vec_set` (*C function*), 512
- `fmpz_mod_mpoly_vec_set_length` (*C function*), 512
- `fmpz_mod_mpoly_vec_struct` (*C type*), 512
- `fmpz_mod_mpoly_vec_swap` (*C function*), 512
- `fmpz_mod_mpoly_vec_t` (*C type*), 512
- `fmpz_mod_mpoly_zero` (*C function*), 502
- `fmpz_mod_mul` (*C function*), 461
- `fmpz_mod_neg` (*C function*), 461
- `fmpz_mod_poly_add` (*C function*), 474
- `fmpz_mod_poly_add_series` (*C function*), 474
- `fmpz_mod_poly_clear` (*C function*), 470
- `fmpz_mod_poly_compose` (*C function*), 490
- `fmpz_mod_poly_compose_mod` (*C function*), 491
- `fmpz_mod_poly_compose_mod_brent_kung` (*C function*), 491
- `fmpz_mod_poly_compose_mod_brent_kung_precomp_preinv` (*C function*), 492
- `fmpz_mod_poly_compose_mod_brent_kung_preinv` (*C function*), 492
- `fmpz_mod_poly_compose_mod_brent_kung_vec_preinv` (*C function*), 493
- `fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded` (*C function*), 493
- `fmpz_mod_poly_compose_mod_brent_kung_vec_preinv_threaded` (*C function*), 493
- `fmpz_mod_poly_compose_mod_horner` (*C function*), 491
- `fmpz_mod_poly_deflate` (*C function*), 496
- `fmpz_mod_poly_deflation` (*C function*), 496
- `fmpz_mod_poly_degree` (*C function*), 472
- `fmpz_mod_poly_derivative` (*C function*), 488
- `fmpz_mod_poly_discriminant` (*C function*), 488
- `fmpz_mod_poly_div` (*C function*), 482
- `fmpz_mod_poly_div_newton_n_preinv` (*C function*), 481
- `fmpz_mod_poly_div_series` (*C function*), 483
- `fmpz_mod_poly_divides` (*C function*), 483
- `fmpz_mod_poly_divides_classical` (*C function*), 483
- `fmpz_mod_poly_divrem` (*C function*), 482
- `fmpz_mod_poly_divrem_basecase` (*C function*), 480
- `fmpz_mod_poly_divrem_f` (*C function*), 482
- `fmpz_mod_poly_divrem_newton_n_preinv` (*C function*), 481
- `fmpz_mod_poly_equal` (*C function*), 473
- `fmpz_mod_poly_equal_trunc` (*C function*), 473
- `fmpz_mod_poly_evaluate_fmpz` (*C function*), 489
- `fmpz_mod_poly_evaluate_fmpz_vec` (*C function*), 489
- `fmpz_mod_poly_evaluate_fmpz_vec_fast` (*C function*), 489
- `fmpz_mod_poly_evaluate_fmpz_vec_iter` (*C function*), 489

-
- `fmpr_mod_poly_factor` (*C function*), 499
 - `fmpr_mod_poly_factor_berlekamp` (*C function*), 500
 - `fmpr_mod_poly_factor_cantor_zassenhaus` (*C function*), 499
 - `fmpr_mod_poly_factor_clear` (*C function*), 498
 - `fmpr_mod_poly_factor_concat` (*C function*), 498
 - `fmpr_mod_poly_factor_distinct_deg` (*C function*), 499
 - `fmpr_mod_poly_factor_distinct_deg_threaded` (*C function*), 499
 - `fmpr_mod_poly_factor_equal_deg` (*C function*), 499
 - `fmpr_mod_poly_factor_equal_deg_prob` (*C function*), 499
 - `fmpr_mod_poly_factor_fit_length` (*C function*), 498
 - `fmpr_mod_poly_factor_init` (*C function*), 498
 - `fmpr_mod_poly_factor_insert` (*C function*), 498
 - `fmpr_mod_poly_factor_kaltofen_shoup` (*C function*), 499
 - `fmpr_mod_poly_factor_pow` (*C function*), 498
 - `fmpr_mod_poly_factor_print` (*C function*), 498
 - `fmpr_mod_poly_factor_realloc` (*C function*), 498
 - `fmpr_mod_poly_factor_set` (*C function*), 498
 - `fmpr_mod_poly_factor_squarefree` (*C function*), 499
 - `fmpr_mod_poly_factor_struct` (*C type*), 497
 - `fmpr_mod_poly_factor_t` (*C type*), 497
 - `fmpr_mod_poly_find_distinct_nonzero_roots` (*C function*), 477
 - `fmpr_mod_poly_fit_length` (*C function*), 471
 - `fmpr_mod_poly_fprint` (*C function*), 495
 - `fmpr_mod_poly_fprint_pretty` (*C function*), 495
 - `fmpr_mod_poly_frobenius_power` (*C function*), 480
 - `fmpr_mod_poly_frobenius_powers_2exp_clear` (*C function*), 480
 - `fmpr_mod_poly_frobenius_powers_2exp_precomp` (*C function*), 479
 - `fmpr_mod_poly_frobenius_powers_clear` (*C function*), 480
 - `fmpr_mod_poly_frobenius_powers_precomp` (*C function*), 480
 - `fmpr_mod_poly_gcd` (*C function*), 484
 - `fmpr_mod_poly_gcd_euclidean_f` (*C function*), 484
 - `fmpr_mod_poly_gcd_f` (*C function*), 484
 - `fmpr_mod_poly_gcdinv` (*C function*), 486
 - `fmpr_mod_poly_gcdinv_euclidean` (*C function*), 486
 - `fmpr_mod_poly_gcdinv_euclidean_f` (*C function*), 486
 - `fmpr_mod_poly_gcdinv_f` (*C function*), 486
 - `fmpr_mod_poly_get_coeff_fmpr` (*C function*), 474
 - `fmpr_mod_poly_get_coeff_mpr` (*C function*), 474
 - `fmpr_mod_poly_get_fmpr_poly` (*C function*), 473
 - `fmpr_mod_poly_get_nmod_poly` (*C function*), 473
 - `fmpr_mod_poly_inflate` (*C function*), 496
 - `fmpr_mod_poly_init` (*C function*), 470
 - `fmpr_mod_poly_init2` (*C function*), 470
 - `fmpr_mod_poly_inv_series` (*C function*), 483
 - `fmpr_mod_poly_inv_series_f` (*C function*), 483
 - `fmpr_mod_poly_invmod` (*C function*), 486
 - `fmpr_mod_poly_invmod_f` (*C function*), 487
 - `fmpr_mod_poly_invsqrt_series` (*C function*), 490
 - `fmpr_mod_poly_is_gen` (*C function*), 473
 - `fmpr_mod_poly_is_irreducible` (*C function*), 498
 - `fmpr_mod_poly_is_irreducible_ddf` (*C function*), 498
 - `fmpr_mod_poly_is_irreducible_rabin` (*C function*), 498
 - `fmpr_mod_poly_is_irreducible_rabin_f` (*C function*), 498
 - `fmpr_mod_poly_is_one` (*C function*), 473
 - `fmpr_mod_poly_is_squarefree` (*C function*), 499
 - `fmpr_mod_poly_is_squarefree_f` (*C function*), 499
 - `fmpr_mod_poly_is_zero` (*C function*), 473
 - `fmpr_mod_poly_lead` (*C function*), 472
 - `fmpr_mod_poly_length` (*C function*), 472
 - `fmpr_mod_poly_make_monic` (*C function*), 484
 - `fmpr_mod_poly_make_monic_f` (*C function*), 484
 - `fmpr_mod_poly_minpoly` (*C function*), 487
 - `fmpr_mod_poly_minpoly_bm` (*C function*), 487
 - `fmpr_mod_poly_minpoly_hgcd` (*C function*), 487
 - `fmpr_mod_poly_mul` (*C function*), 476
 - `fmpr_mod_poly_mulhigh` (*C function*), 476
 - `fmpr_mod_poly_mullo` (*C function*), 476
 - `fmpr_mod_poly_mulmid` (*C function*), 476
 - `fmpr_mod_poly_mulmod` (*C function*), 476
 - `fmpr_mod_poly_mulmod_preinv` (*C function*), 477
 - `fmpr_mod_poly_neg` (*C function*), 475
 - `fmpr_mod_poly_one` (*C function*), 472
 - `fmpr_mod_poly_pow` (*C function*), 477
 - `fmpr_mod_poly_pow_trunc` (*C function*), 477
 - `fmpr_mod_poly_pow_trunc_binexp` (*C function*), 478
 - `fmpr_mod_poly_powers_mod_bsbs` (*C function*), 479
 - `fmpr_mod_poly_powers_mod_naive` (*C function*), 479
 - `fmpr_mod_poly_powmod_fmpr_binexp` (*C function*), 478
 - `fmpr_mod_poly_powmod_fmpr_binexp_preinv` (*C function*), 479
 - `fmpr_mod_poly_powmod_ui_binexp` (*C function*), 478
 - `fmpr_mod_poly_powmod_ui_binexp_preinv` (*C function*), 478
 - `fmpr_mod_poly_powmod_x_fmpr_preinv` (*C function*), 479

`fmpr_mod_poly_precompute_matrix` (*C function*), 491
`fmpr_mod_poly_print` (*C function*), 495
`fmpr_mod_poly_print_pretty` (*C function*), 495
`fmpr_mod_poly_product_roots_fmpr_vec` (*C function*), 477
`fmpr_mod_poly_radix` (*C function*), 495
`fmpr_mod_poly_radix_init` (*C function*), 494
`fmpr_mod_poly_randtest` (*C function*), 471
`fmpr_mod_poly_randtest_irreducible` (*C function*), 471
`fmpr_mod_poly_randtest_monic` (*C function*), 471
`fmpr_mod_poly_randtest_monic_irreducible` (*C function*), 471
`fmpr_mod_poly_randtest_monic_primitive` (*C function*), 471
`fmpr_mod_poly_randtest_not_zero` (*C function*), 471
`fmpr_mod_poly_randtest_pentomial` (*C function*), 472
`fmpr_mod_poly_randtest_pentomial_irreducible` (*C function*), 472
`fmpr_mod_poly_randtest_sparse_irreducible` (*C function*), 472
`fmpr_mod_poly_randtest_trinomial` (*C function*), 471
`fmpr_mod_poly_randtest_trinomial_irreducible` (*C function*), 471
`fmpr_mod_poly_realloc` (*C function*), 470
`fmpr_mod_poly_rem` (*C function*), 482
`fmpr_mod_poly_rem_basecase` (*C function*), 481
`fmpr_mod_poly_rem_f` (*C function*), 482
`fmpr_mod_poly_remove` (*C function*), 481
`fmpr_mod_poly_resultant` (*C function*), 488
`fmpr_mod_poly_reverse` (*C function*), 472
`fmpr_mod_poly_roots` (*C function*), 500
`fmpr_mod_poly_roots_factored` (*C function*), 500
`fmpr_mod_poly_scalar_addmul_fmpr` (*C function*), 475
`fmpr_mod_poly_scalar_div_fmpr` (*C function*), 475
`fmpr_mod_poly_scalar_mul_fmpr` (*C function*), 475
`fmpr_mod_poly_scalar_mul_ui` (*C function*), 475
`fmpr_mod_poly_set` (*C function*), 472
`fmpr_mod_poly_set_coeff_fmpr` (*C function*), 474
`fmpr_mod_poly_set_coeff_mpr` (*C function*), 474
`fmpr_mod_poly_set_coeff_ui` (*C function*), 474
`fmpr_mod_poly_set_fmpr` (*C function*), 473
`fmpr_mod_poly_set_fmpr_poly` (*C function*), 473
`fmpr_mod_poly_set_nmod_poly` (*C function*), 473
`fmpr_mod_poly_set_trunc` (*C function*), 471
`fmpr_mod_poly_set_ui` (*C function*), 473
`fmpr_mod_poly_shift_left` (*C function*), 474
`fmpr_mod_poly_shift_right` (*C function*), 474
`fmpr_mod_poly_sqr` (*C function*), 476
`fmpr_mod_poly_sqrt` (*C function*), 490
`fmpr_mod_poly_sqrt_series` (*C function*), 490
`fmpr_mod_poly_struct` (*C type*), 470
`fmpr_mod_poly_sub` (*C function*), 475
`fmpr_mod_poly_sub_series` (*C function*), 475
`fmpr_mod_poly_swap` (*C function*), 472
`fmpr_mod_poly_t` (*C type*), 470
`fmpr_mod_poly_truncate` (*C function*), 471
`fmpr_mod_poly_xgcd` (*C function*), 485
`fmpr_mod_poly_xgcd_euclidean_f` (*C function*), 485
`fmpr_mod_poly_xgcd_f` (*C function*), 485
`fmpr_mod_poly_zero` (*C function*), 472
`fmpr_mod_poly_zero_coeffs` (*C function*), 472
`fmpr_mod_pow_fmpr` (*C function*), 461
`fmpr_mod_pow_ui` (*C function*), 461
`fmpr_mod_set_fmpr` (*C function*), 460
`fmpr_mod_si_sub` (*C function*), 460
`fmpr_mod_sub` (*C function*), 460
`fmpr_mod_sub_fmpr` (*C function*), 460
`fmpr_mod_sub_si` (*C function*), 460
`fmpr_mod_sub_ui` (*C function*), 460
`fmpr_mod_ui` (*C function*), 165
`fmpr_mod_ui_sub` (*C function*), 460
`fmpr_moebius_mu` (*C function*), 175
`fmpr_mpoly_add` (*C function*), 266
`fmpr_mpoly_add_fmpr` (*C function*), 265
`fmpr_mpoly_add_si` (*C function*), 265
`fmpr_mpoly_add_ui` (*C function*), 265
`fmpr_mpoly_buchberger_naive` (*C function*), 275
`fmpr_mpoly_buchberger_naive_with_limits` (*C function*), 275
`fmpr_mpoly_clear` (*C function*), 259
`fmpr_mpoly_cmp` (*C function*), 263
`fmpr_mpoly_combine_like_terms` (*C function*), 265
`fmpr_mpoly_compose_fmpr_mpoly` (*C function*), 267
`fmpr_mpoly_compose_fmpr_mpoly_gen` (*C function*), 267
`fmpr_mpoly_compose_fmpr_mpoly_geobucket` (*C function*), 267
`fmpr_mpoly_compose_fmpr_mpoly_horner` (*C function*), 267
`fmpr_mpoly_compose_fmpr_poly` (*C function*), 267
`fmpr_mpoly_content_vars` (*C function*), 269
`fmpr_mpoly_ctx_clear` (*C function*), 259
`fmpr_mpoly_ctx_init` (*C function*), 259
`fmpr_mpoly_ctx_nvars` (*C function*), 259
`fmpr_mpoly_ctx_ord` (*C function*), 259
`fmpr_mpoly_ctx_struct` (*C type*), 259
`fmpr_mpoly_ctx_t` (*C type*), 259
`fmpr_mpoly_deflate` (*C function*), 271
`fmpr_mpoly_deflation` (*C function*), 271
`fmpr_mpoly_degree_fmpr` (*C function*), 261
`fmpr_mpoly_degree_si` (*C function*), 261

- `fmpz_mpoly_degrees_fit_si` (*C function*), 261
- `fmpz_mpoly_degrees_fmpz` (*C function*), 261
- `fmpz_mpoly_degrees_si` (*C function*), 261
- `fmpz_mpoly_derivative` (*C function*), 267
- `fmpz_mpoly_discriminant` (*C function*), 270
- `fmpz_mpoly_div` (*C function*), 268
- `fmpz_mpoly_div_monagan_pearce` (*C function*), 272
- `fmpz_mpoly_divides` (*C function*), 268
- `fmpz_mpoly_divides_array` (*C function*), 271
- `fmpz_mpoly_divides_heap_threaded` (*C function*), 272
- `fmpz_mpoly_divides_monagan_pearce` (*C function*), 272
- `fmpz_mpoly_divrem` (*C function*), 268
- `fmpz_mpoly_divrem_array` (*C function*), 273
- `fmpz_mpoly_divrem_ideal` (*C function*), 269
- `fmpz_mpoly_divrem_ideal_monagan_pearce` (*C function*), 274
- `fmpz_mpoly_divrem_monagan_pearce` (*C function*), 273
- `fmpz_mpoly_equal` (*C function*), 260
- `fmpz_mpoly_equal_fmpz` (*C function*), 261
- `fmpz_mpoly_equal_si` (*C function*), 261
- `fmpz_mpoly_equal_ui` (*C function*), 261
- `fmpz_mpoly_evaluate_all_fmpz` (*C function*), 267
- `fmpz_mpoly_evaluate_one_fmpz` (*C function*), 267
- `fmpz_mpoly_factor` (*C function*), 277
- `fmpz_mpoly_factor_clear` (*C function*), 276
- `fmpz_mpoly_factor_get_base` (*C function*), 276
- `fmpz_mpoly_factor_get_constant_fmpz` (*C function*), 276
- `fmpz_mpoly_factor_get_constant_fmpz` (*C function*), 276
- `fmpz_mpoly_factor_get_exp_si` (*C function*), 276
- `fmpz_mpoly_factor_init` (*C function*), 276
- `fmpz_mpoly_factor_length` (*C function*), 276
- `fmpz_mpoly_factor_sort` (*C function*), 276
- `fmpz_mpoly_factor_squarefree` (*C function*), 277
- `fmpz_mpoly_factor_struct` (*C type*), 276
- `fmpz_mpoly_factor_swap` (*C function*), 276
- `fmpz_mpoly_factor_swap_base` (*C function*), 276
- `fmpz_mpoly_factor_t` (*C type*), 276
- `fmpz_mpoly_fit_bits` (*C function*), 259
- `fmpz_mpoly_fit_length` (*C function*), 259
- `fmpz_mpoly_fprint_pretty` (*C function*), 260
- `fmpz_mpoly_from_univar` (*C function*), 270
- `fmpz_mpoly_gcd` (*C function*), 269
- `fmpz_mpoly_gcd_brown` (*C function*), 269
- `fmpz_mpoly_gcd_cofactors` (*C function*), 269
- `fmpz_mpoly_gcd_hensel` (*C function*), 269
- `fmpz_mpoly_gcd_subresultant` (*C function*), 269
- `fmpz_mpoly_gcd_zippel` (*C function*), 269
- `fmpz_mpoly_gcd_zippel2` (*C function*), 269
- `fmpz_mpoly_gen` (*C function*), 260
- `fmpz_mpoly_get_coeff_fmpz_fmpz` (*C function*), 262
- `fmpz_mpoly_get_coeff_fmpz_monomial` (*C function*), 262
- `fmpz_mpoly_get_coeff_fmpz_ui` (*C function*), 262
- `fmpz_mpoly_get_coeff_si_fmpz` (*C function*), 262
- `fmpz_mpoly_get_coeff_si_ui` (*C function*), 262
- `fmpz_mpoly_get_coeff_ui_fmpz` (*C function*), 262
- `fmpz_mpoly_get_coeff_ui_ui` (*C function*), 262
- `fmpz_mpoly_get_coeff_vars_ui` (*C function*), 262
- `fmpz_mpoly_get_fmpz` (*C function*), 261
- `fmpz_mpoly_get_fmpz_poly` (*C function*), 263
- `fmpz_mpoly_get_str_pretty` (*C function*), 260
- `fmpz_mpoly_get_term` (*C function*), 264
- `fmpz_mpoly_get_term_coeff_fmpz` (*C function*), 263
- `fmpz_mpoly_get_term_coeff_si` (*C function*), 263
- `fmpz_mpoly_get_term_coeff_ui` (*C function*), 263
- `fmpz_mpoly_get_term_exp_fmpz` (*C function*), 264
- `fmpz_mpoly_get_term_exp_si` (*C function*), 264
- `fmpz_mpoly_get_term_exp_ui` (*C function*), 264
- `fmpz_mpoly_get_term_monomial` (*C function*), 264
- `fmpz_mpoly_get_term_var_exp_si` (*C function*), 264
- `fmpz_mpoly_get_term_var_exp_ui` (*C function*), 264
- `fmpz_mpoly_inflate` (*C function*), 271
- `fmpz_mpoly_init` (*C function*), 259
- `fmpz_mpoly_init2` (*C function*), 259
- `fmpz_mpoly_init3` (*C function*), 259
- `fmpz_mpoly_integral` (*C function*), 267
- `fmpz_mpoly_is_canonical` (*C function*), 263
- `fmpz_mpoly_is_fmpz` (*C function*), 261
- `fmpz_mpoly_is_fmpz_poly` (*C function*), 263
- `fmpz_mpoly_is_gen` (*C function*), 260
- `fmpz_mpoly_is_one` (*C function*), 261
- `fmpz_mpoly_is_square` (*C function*), 270
- `fmpz_mpoly_is_zero` (*C function*), 261
- `fmpz_mpoly_length` (*C function*), 263
- `fmpz_mpoly_max_bits` (*C function*), 260
- `fmpz_mpoly_mul` (*C function*), 268
- `fmpz_mpoly_mul_array` (*C function*), 268
- `fmpz_mpoly_mul_array_threaded` (*C function*), 268
- `fmpz_mpoly_mul_dense` (*C function*), 268
- `fmpz_mpoly_mul_heap_threaded` (*C function*), 268
- `fmpz_mpoly_mul_johnson` (*C function*), 268
- `fmpz_mpoly_mul_threaded` (*C function*), 268

`fmpz_mpoly_neg` (*C function*), 266
`fmpz_mpoly_one` (*C function*), 261
`fmpz_mpoly_pow_fmpz` (*C function*), 268
`fmpz_mpoly_pow_fps` (*C function*), 271
`fmpz_mpoly_pow_ui` (*C function*), 268
`fmpz_mpoly_primitive_part` (*C function*), 270
`fmpz_mpoly_print_pretty` (*C function*), 260
`fmpz_mpoly_push_term_fmpz_ffmpz` (*C function*), 264
`fmpz_mpoly_push_term_fmpz_fmpz` (*C function*), 264
`fmpz_mpoly_push_term_fmpz_ui` (*C function*), 264
`fmpz_mpoly_push_term_si_ffmpz` (*C function*), 264
`fmpz_mpoly_push_term_si_fmpz` (*C function*), 264
`fmpz_mpoly_push_term_si_ui` (*C function*), 264
`fmpz_mpoly_push_term_ui_ffmpz` (*C function*), 264
`fmpz_mpoly_push_term_ui_fmpz` (*C function*), 264
`fmpz_mpoly_push_term_ui_ui` (*C function*), 264
`fmpz_mpoly_q_add` (*C function*), 379
`fmpz_mpoly_q_add_fmpz` (*C function*), 379
`fmpz_mpoly_q_add_fmpz` (*C function*), 379
`fmpz_mpoly_q_add_si` (*C function*), 379
`fmpz_mpoly_q_canonicalise` (*C function*), 377
`fmpz_mpoly_q_clear` (*C function*), 377
`fmpz_mpoly_q_content` (*C function*), 380
`fmpz_mpoly_q_denref` (*C macro*), 377
`fmpz_mpoly_q_div` (*C function*), 379
`fmpz_mpoly_q_div_fmpz` (*C function*), 379
`fmpz_mpoly_q_div_fmpz` (*C function*), 379
`fmpz_mpoly_q_div_si` (*C function*), 379
`fmpz_mpoly_q_equal` (*C function*), 379
`fmpz_mpoly_q_gen` (*C function*), 378
`fmpz_mpoly_q_get_str_pretty` (*C function*), 378
`fmpz_mpoly_q_init` (*C function*), 377
`fmpz_mpoly_q_inv` (*C function*), 380
`fmpz_mpoly_q_is_canonical` (*C function*), 377
`fmpz_mpoly_q_is_one` (*C function*), 378
`fmpz_mpoly_q_is_zero` (*C function*), 378
`fmpz_mpoly_q_mul` (*C function*), 379
`fmpz_mpoly_q_mul_fmpz` (*C function*), 379
`fmpz_mpoly_q_mul_fmpz` (*C function*), 379
`fmpz_mpoly_q_mul_si` (*C function*), 379
`fmpz_mpoly_q_neg` (*C function*), 379
`fmpz_mpoly_q_numref` (*C macro*), 377
`fmpz_mpoly_q_one` (*C function*), 378
`fmpz_mpoly_q_print_pretty` (*C function*), 378
`fmpz_mpoly_q_randtest` (*C function*), 379
`fmpz_mpoly_q_set` (*C function*), 377
`fmpz_mpoly_q_set_fmpz` (*C function*), 377
`fmpz_mpoly_q_set_fmpz` (*C function*), 377
`fmpz_mpoly_q_set_si` (*C function*), 377
`fmpz_mpoly_q_set_str_pretty` (*C function*), 378
`fmpz_mpoly_q_struct` (*C type*), 377
`fmpz_mpoly_q_sub` (*C function*), 379
`fmpz_mpoly_q_sub_fmpz` (*C function*), 379
`fmpz_mpoly_q_sub_fmpz` (*C function*), 379
`fmpz_mpoly_q_sub_si` (*C function*), 379
`fmpz_mpoly_q_swap` (*C function*), 377
`fmpz_mpoly_q_t` (*C type*), 377
`fmpz_mpoly_q_used_vars` (*C function*), 378
`fmpz_mpoly_q_used_vars_den` (*C function*), 378
`fmpz_mpoly_q_used_vars_num` (*C function*), 378
`fmpz_mpoly_q_zero` (*C function*), 378
`fmpz_mpoly_quasidiv` (*C function*), 269
`fmpz_mpoly_quasidivrem` (*C function*), 268
`fmpz_mpoly_quasidivrem_heap` (*C function*), 273
`fmpz_mpoly_quasidivrem_ideal` (*C function*), 269
`fmpz_mpoly_randtest_bits` (*C function*), 265
`fmpz_mpoly_randtest_bound` (*C function*), 265
`fmpz_mpoly_randtest_bounds` (*C function*), 265
`fmpz_mpoly_realloc` (*C function*), 259
`fmpz_mpoly_reduction_primitive_part` (*C function*), 275
`fmpz_mpoly_resize` (*C function*), 263
`fmpz_mpoly_resultant` (*C function*), 270
`fmpz_mpoly_reverse` (*C function*), 265
`fmpz_mpoly_scalar_divexact_fmpz` (*C function*), 266
`fmpz_mpoly_scalar_divexact_si` (*C function*), 266
`fmpz_mpoly_scalar_divexact_ui` (*C function*), 266
`fmpz_mpoly_scalar_divides_fmpz` (*C function*), 266
`fmpz_mpoly_scalar_divides_si` (*C function*), 266
`fmpz_mpoly_scalar_divides_ui` (*C function*), 266
`fmpz_mpoly_scalar_fmma` (*C function*), 266
`fmpz_mpoly_scalar_mul_fmpz` (*C function*), 266
`fmpz_mpoly_scalar_mul_si` (*C function*), 266
`fmpz_mpoly_scalar_mul_ui` (*C function*), 266
`fmpz_mpoly_set` (*C function*), 260
`fmpz_mpoly_set_coeff_fmpz_fmpz` (*C function*), 262
`fmpz_mpoly_set_coeff_fmpz_monomial` (*C function*), 262
`fmpz_mpoly_set_coeff_fmpz_ui` (*C function*), 262
`fmpz_mpoly_set_coeff_si_fmpz` (*C function*), 262
`fmpz_mpoly_set_coeff_si_ui` (*C function*), 262
`fmpz_mpoly_set_coeff_ui_fmpz` (*C function*), 262
`fmpz_mpoly_set_coeff_ui_ui` (*C function*), 262
`fmpz_mpoly_set_fmpz` (*C function*), 261
`fmpz_mpoly_set_fmpz_poly` (*C function*), 263
`fmpz_mpoly_set_gen_fmpz_poly` (*C function*), 263
`fmpz_mpoly_set_si` (*C function*), 261

- fmpz_mpoly_set_str_pretty (*C function*), 260
 fmpz_mpoly_set_term_coeff_fmpz (*C function*), 263
 fmpz_mpoly_set_term_coeff_si (*C function*), 263
 fmpz_mpoly_set_term_coeff_ui (*C function*), 263
 fmpz_mpoly_set_term_exp_fmpz (*C function*), 264
 fmpz_mpoly_set_term_exp_ui (*C function*), 264
 fmpz_mpoly_set_ui (*C function*), 261
 fmpz_mpoly_sort_terms (*C function*), 265
 fmpz_mpoly_spoly (*C function*), 275
 fmpz_mpoly_sqrt (*C function*), 270
 fmpz_mpoly_sqrt_heap (*C function*), 270
 fmpz_mpoly_struct (*C type*), 259
 fmpz_mpoly_sub (*C function*), 266
 fmpz_mpoly_sub_fmpz (*C function*), 265
 fmpz_mpoly_sub_si (*C function*), 265
 fmpz_mpoly_sub_ui (*C function*), 265
 fmpz_mpoly_swap (*C function*), 260
 fmpz_mpoly_symmetric (*C function*), 275
 fmpz_mpoly_symmetric_gens (*C function*), 275
 fmpz_mpoly_t (*C type*), 259
 fmpz_mpoly_term_coeff_ref (*C function*), 263
 fmpz_mpoly_term_content (*C function*), 269
 fmpz_mpoly_term_exp_fits_si (*C function*), 264
 fmpz_mpoly_term_exp_fits_ui (*C function*), 264
 fmpz_mpoly_to_univar (*C function*), 270
 fmpz_mpoly_total_degree_fits_si (*C function*), 261
 fmpz_mpoly_total_degree_fmpz (*C function*), 261
 fmpz_mpoly_total_degree_si (*C function*), 261
 fmpz_mpoly_univar_clear (*C function*), 270
 fmpz_mpoly_univar_degree_fits_si (*C function*), 271
 fmpz_mpoly_univar_get_term_coeff (*C function*), 271
 fmpz_mpoly_univar_get_term_exp_si (*C function*), 271
 fmpz_mpoly_univar_init (*C function*), 270
 fmpz_mpoly_univar_length (*C function*), 271
 fmpz_mpoly_univar_swap (*C function*), 270
 fmpz_mpoly_univar_swap_term_coeff (*C function*), 271
 fmpz_mpoly_used_vars (*C function*), 261
 fmpz_mpoly_vec_append (*C function*), 274
 fmpz_mpoly_vec_autoreduction (*C function*), 275
 fmpz_mpoly_vec_autoreduction_groebner (*C function*), 275
 fmpz_mpoly_vec_clear (*C function*), 274
 fmpz_mpoly_vec_entry (*C macro*), 274
 fmpz_mpoly_vec_fit_length (*C function*), 274
 fmpz_mpoly_vec_init (*C function*), 274
 fmpz_mpoly_vec_insert_unique (*C function*), 274
 fmpz_mpoly_vec_is_autoreduced (*C function*), 275
 fmpz_mpoly_vec_is_groebner (*C function*), 275
 fmpz_mpoly_vec_print (*C function*), 274
 fmpz_mpoly_vec_randtest_not_zero (*C function*), 274
 fmpz_mpoly_vec_set (*C function*), 274
 fmpz_mpoly_vec_set_length (*C function*), 274
 fmpz_mpoly_vec_set_primitive_unique (*C function*), 274
 fmpz_mpoly_vec_struct (*C type*), 274
 fmpz_mpoly_vec_swap (*C function*), 274
 fmpz_mpoly_vec_t (*C type*), 274
 fmpz_mpoly_zero (*C function*), 261
 fmpz_mul (*C function*), 164
 fmpz_mul2_uiui (*C function*), 164
 fmpz_mul_2exp (*C function*), 164
 fmpz_mul_si (*C function*), 164
 fmpz_mul_si_tdiv_q_2exp (*C function*), 167
 fmpz_mul_tdiv_q_2exp (*C function*), 167
 fmpz_mul_ui (*C function*), 164
 fmpz_multi_CRT (*C function*), 171
 fmpz_multi_CRT_clear (*C function*), 171
 fmpz_multi_CRT_init (*C function*), 171
 fmpz_multi_CRT_precomp (*C function*), 171
 fmpz_multi_CRT_precompute (*C function*), 171
 fmpz_multi_CRT_ui (*C function*), 171
 fmpz_multi_mod_ui (*C function*), 170
 fmpz_ndiv_qr (*C function*), 164
 fmpz_neg (*C function*), 163
 fmpz_neg_ui (*C function*), 159
 fmpz_neg_uiui (*C function*), 159
 fmpz_negmod (*C function*), 169
 fmpz_next_smooth_prime (*C function*), 461
 fmpz_nextprime (*C function*), 174
 fmpz_one (*C function*), 162
 fmpz_one_2exp (*C function*), 164
 fmpz_or (*C function*), 170
 fmpz_out_raw (*C function*), 161
 fmpz_poly_2norm (*C function*), 221
 fmpz_poly_add (*C function*), 212
 fmpz_poly_add_series (*C function*), 212
 fmpz_poly_attach_shift (*C function*), 208
 fmpz_poly_attach_truncate (*C function*), 208
 fmpz_poly_bit_pack (*C function*), 214
 fmpz_poly_bit_unpack (*C function*), 214
 fmpz_poly_bit_unpack_unsigned (*C function*), 214
 fmpz_poly_bound_roots (*C function*), 247
 fmpz_poly_chebyshev_t (*C function*), 248
 fmpz_poly_chebyshev_u (*C function*), 248
 fmpz_poly_CLD_bound (*C function*), 250
 fmpz_poly_clear (*C function*), 208
 fmpz_poly_compose (*C function*), 237
 fmpz_poly_compose_divconquer (*C function*), 237
 fmpz_poly_compose_horner (*C function*), 237
 fmpz_poly_compose_series (*C function*), 239

`fmprz_poly_compose_series_brent_kung` (*C function*), 238
`fmprz_poly_compose_series_horner` (*C function*), 238
`fmprz_poly_content` (*C function*), 224
`fmprz_poly_cos_minpoly` (*C function*), 248
`fmprz_poly_CRT_ui` (*C function*), 246
`fmprz_poly_cyclotomic` (*C function*), 248
`fmprz_poly_deflate` (*C function*), 237
`fmprz_poly_deflation` (*C function*), 237
`fmprz_poly_degree` (*C function*), 209
`fmprz_poly_derivative` (*C function*), 234
`fmprz_poly_discriminant` (*C function*), 224
`fmprz_poly_div` (*C function*), 228
`fmprz_poly_div_basecase` (*C function*), 227
`fmprz_poly_div_divconquer` (*C function*), 228
`fmprz_poly_div_preinv` (*C function*), 229
`fmprz_poly_div_root_fmprz` (*C function*), 229
`fmprz_poly_div_series` (*C function*), 232
`fmprz_poly_div_series_basecase` (*C function*), 231
`fmprz_poly_div_series_divconquer` (*C function*), 231
`fmprz_poly_divexact` (*C function*), 229
`fmprz_poly_divexact_root_fmprz` (*C function*), 229
`fmprz_poly_divhigh_smodp` (*C function*), 231
`fmprz_poly_divides` (*C function*), 230
`fmprz_poly_divlow_smodp` (*C function*), 231
`fmprz_poly_divrem` (*C function*), 226
`fmprz_poly_divrem_basecase` (*C function*), 225
`fmprz_poly_divrem_divconquer` (*C function*), 226
`fmprz_poly_divrem_preinv` (*C function*), 230
`fmprz_poly_equal` (*C function*), 212
`fmprz_poly_equal_trunc` (*C function*), 212
`fmprz_poly_eta_qexp` (*C function*), 250
`fmprz_poly_eulerian_polynomial` (*C function*), 249
`fmprz_poly_evaluate_divconquer_fmprz` (*C function*), 234
`fmprz_poly_evaluate_divconquer_fmprz` (*C function*), 234
`fmprz_poly_evaluate_fmprz` (*C function*), 235
`fmprz_poly_evaluate_fmprz` (*C function*), 234
`fmprz_poly_evaluate_fmprz_vec` (*C function*), 235
`fmprz_poly_evaluate_horner_d` (*C function*), 235
`fmprz_poly_evaluate_horner_d_2exp` (*C function*), 235
`fmprz_poly_evaluate_horner_fmprz` (*C function*), 235
`fmprz_poly_evaluate_horner_fmprz` (*C function*), 234
`fmprz_poly_evaluate_mod` (*C function*), 235
`fmprz_poly_factor` (*C function*), 258
`fmprz_poly_factor_clear` (*C function*), 257
`fmprz_poly_factor_concat` (*C function*), 257
`fmprz_poly_factor_fit_length` (*C function*), 257
`fmprz_poly_factor_init` (*C function*), 257
`fmprz_poly_factor_init2` (*C function*), 257
`fmprz_poly_factor_insert` (*C function*), 257
`fmprz_poly_factor_print` (*C function*), 258
`fmprz_poly_factor_realloc` (*C function*), 257
`fmprz_poly_factor_set` (*C function*), 257
`fmprz_poly_factor_squarefree` (*C function*), 258
`fmprz_poly_factor_struct` (*C type*), 257
`fmprz_poly_factor_t` (*C type*), 257
`fmprz_poly_factor_zassenhaus` (*C function*), 258
`fmprz_poly_factor_zassenhaus_recombination` (*C function*), 258
`fmprz_poly_fibonacci` (*C function*), 249
`fmprz_poly_fit_length` (*C function*), 208
`fmprz_poly_fprint` (*C function*), 244
`fmprz_poly_fprint_pretty` (*C function*), 245
`fmprz_poly_fread` (*C function*), 245
`fmprz_poly_fread_pretty` (*C function*), 245
`fmprz_poly_gcd` (*C function*), 222
`fmprz_poly_gcd_heuristic` (*C function*), 222
`fmprz_poly_gcd_modular` (*C function*), 222
`fmprz_poly_gcd_subresultant` (*C function*), 221
`fmprz_poly_get_coeff_fmprz` (*C function*), 211
`fmprz_poly_get_coeff_ptr` (*C function*), 211
`fmprz_poly_get_coeff_si` (*C function*), 211
`fmprz_poly_get_coeff_ui` (*C function*), 211
`fmprz_poly_get_nmod_poly` (*C function*), 245
`fmprz_poly_get_str` (*C function*), 209
`fmprz_poly_get_str_pretty` (*C function*), 209
`fmprz_poly_height` (*C function*), 221
`fmprz_poly_hensel_build_tree` (*C function*), 241
`fmprz_poly_hensel_lift` (*C function*), 242
`fmprz_poly_hensel_lift_once` (*C function*), 243
`fmprz_poly_hensel_lift_only_inverse` (*C function*), 242
`fmprz_poly_hensel_lift_tree` (*C function*), 242
`fmprz_poly_hensel_lift_tree_recursive` (*C function*), 242
`fmprz_poly_hensel_lift_without_inverse` (*C function*), 242
`fmprz_poly_hermite_h` (*C function*), 249
`fmprz_poly_hermite_he` (*C function*), 249
`fmprz_poly_inflate` (*C function*), 237
`fmprz_poly_init` (*C function*), 208
`fmprz_poly_init2` (*C function*), 208
`fmprz_poly_interpolate` (*C function*), 236
`fmprz_poly_interpolate_exact` (*C function*), 236
`fmprz_poly_interpolate_exact_newton` (*C function*), 236
`fmprz_poly_interpolate_fmprz_vec` (*C function*), 236
`fmprz_poly_interpolate_multi_mod` (*C function*), 236
`fmprz_poly_interpolate_newton` (*C function*), 236
`fmprz_poly_inv_series` (*C function*), 231
`fmprz_poly_inv_series_basecase` (*C function*), 231
`fmprz_poly_inv_series_newton` (*C function*), 231

- `mpz_poly_is_cyclotomic` (*C function*), 248
- `mpz_poly_is_gen` (*C function*), 212
- `mpz_poly_is_one` (*C function*), 212
- `mpz_poly_is_squarefree` (*C function*), 225
- `mpz_poly_is_unit` (*C function*), 212
- `mpz_poly_is_zero` (*C function*), 212
- `mpz_poly_lcm` (*C function*), 223
- `mpz_poly_lead` (*C function*), 211
- `mpz_poly_legendre_pt` (*C function*), 248
- `mpz_poly_length` (*C function*), 209
- `mpz_poly_mat_add` (*C function*), 253
- `mpz_poly_mat_clear` (*C function*), 251
- `mpz_poly_mat_det` (*C function*), 256
- `mpz_poly_mat_det_fflu` (*C function*), 256
- `mpz_poly_mat_det_interpolate` (*C function*), 256
- `mpz_poly_mat_entry` (*C function*), 252
- `mpz_poly_mat_equal` (*C function*), 253
- `mpz_poly_mat_evaluate_mpz` (*C function*), 253
- `mpz_poly_mat_fflu` (*C function*), 255
- `mpz_poly_mat_find_pivot_any` (*C function*), 255
- `mpz_poly_mat_find_pivot_partial` (*C function*), 255
- `mpz_poly_mat_init` (*C function*), 251
- `mpz_poly_mat_init_set` (*C function*), 251
- `mpz_poly_mat_inv` (*C function*), 256
- `mpz_poly_mat_is_empty` (*C function*), 253
- `mpz_poly_mat_is_one` (*C function*), 253
- `mpz_poly_mat_is_square` (*C function*), 253
- `mpz_poly_mat_is_zero` (*C function*), 253
- `mpz_poly_mat_max_bits` (*C function*), 253
- `mpz_poly_mat_max_length` (*C function*), 253
- `mpz_poly_mat_mul` (*C function*), 254
- `mpz_poly_mat_mul_classical` (*C function*), 254
- `mpz_poly_mat_mul_KS` (*C function*), 254
- `mpz_poly_mat_mullo` (*C function*), 254
- `mpz_poly_mat_ncols` (*C function*), 251
- `mpz_poly_mat_neg` (*C function*), 254
- `mpz_poly_mat_nrows` (*C function*), 251
- `mpz_poly_mat_nullspace` (*C function*), 256
- `mpz_poly_mat_one` (*C function*), 252
- `mpz_poly_mat_pow` (*C function*), 254
- `mpz_poly_mat_pow_trunc` (*C function*), 254
- `mpz_poly_mat_print` (*C function*), 252
- `mpz_poly_mat_prod` (*C function*), 254
- `mpz_poly_mat_randtest` (*C function*), 252
- `mpz_poly_mat_randtest_sparse` (*C function*), 252
- `mpz_poly_mat_randtest_unsigned` (*C function*), 252
- `mpz_poly_mat_rank` (*C function*), 256
- `mpz_poly_mat_rref` (*C function*), 255
- `mpz_poly_mat_scalar_mul_mpz` (*C function*), 253
- `mpz_poly_mat_scalar_mul_mpz_poly` (*C function*), 253
- `mpz_poly_mat_set` (*C function*), 252
- `mpz_poly_mat_solve` (*C function*), 256
- `mpz_poly_mat_solve_fflu` (*C function*), 256
- `mpz_poly_mat_solve_fflu_precomp` (*C function*), 256
- `mpz_poly_mat_sqr` (*C function*), 254
- `mpz_poly_mat_sqr_classical` (*C function*), 254
- `mpz_poly_mat_sqr_KS` (*C function*), 254
- `mpz_poly_mat_sqr_low` (*C function*), 254
- `mpz_poly_mat_struct` (*C type*), 251
- `mpz_poly_mat_sub` (*C function*), 253
- `mpz_poly_mat_swap` (*C function*), 252
- `mpz_poly_mat_swap_entrywise` (*C function*), 252
- `mpz_poly_mat_t` (*C type*), 251
- `mpz_poly_mat_trace` (*C function*), 255
- `mpz_poly_mat_transpose` (*C function*), 253
- `mpz_poly_mat_zero` (*C function*), 252
- `mpz_poly_max_bits` (*C function*), 221
- `mpz_poly_max_limbs` (*C function*), 221
- `mpz_poly_mul` (*C function*), 216
- `mpz_poly_mul_classical` (*C function*), 214
- `mpz_poly_mul_karatsuba` (*C function*), 215
- `mpz_poly_mul_KS` (*C function*), 216
- `mpz_poly_mul_precache_clear` (*C function*), 217
- `mpz_poly_mul_SS` (*C function*), 216
- `mpz_poly_mul_SS_precache` (*C function*), 218
- `mpz_poly_mul_SS_precache_init` (*C function*), 217
- `mpz_poly_mulhigh_classical` (*C function*), 215
- `mpz_poly_mulhigh_karatsuba_n` (*C function*), 216
- `mpz_poly_mulhigh_n` (*C function*), 217
- `mpz_poly_mullo` (*C function*), 217
- `mpz_poly_mullo_classical` (*C function*), 215
- `mpz_poly_mullo_karatsuba_n` (*C function*), 215
- `mpz_poly_mullo_KS` (*C function*), 216
- `mpz_poly_mullo_SS` (*C function*), 216
- `mpz_poly_mullo_SS_precache` (*C function*), 217
- `mpz_poly_mulmid` (*C function*), 215
- `mpz_poly_mulmid_classical` (*C function*), 215
- `mpz_poly_mulmid_KS` (*C function*), 215
- `mpz_poly_mulmid_SS` (*C function*), 215
- `mpz_poly_neg` (*C function*), 212
- `mpz_poly_nth_derivative` (*C function*), 234
- `mpz_poly_num_real_roots` (*C function*), 247
- `mpz_poly_num_real_roots_sturm` (*C function*), 247
- `mpz_poly_one` (*C function*), 210
- `mpz_poly_pow` (*C function*), 220
- `mpz_poly_pow_addchains` (*C function*), 220
- `mpz_poly_pow_binexp` (*C function*), 220
- `mpz_poly_pow_binomial` (*C function*), 219
- `mpz_poly_pow_multinomial` (*C function*), 219
- `mpz_poly_pow_trunc` (*C function*), 220
- `mpz_poly_power_sums` (*C function*), 241

`fmprz_poly_power_sums_naive` (*C function*), 241
`fmprz_poly_power_sums_to_poly` (*C function*), 241
`fmprz_poly_powers_clear` (*C function*), 230
`fmprz_poly_powers_precompute` (*C function*), 230
`fmprz_poly_preinvert` (*C function*), 229
`fmprz_poly_primitive_part` (*C function*), 225
`fmprz_poly_print` (*C function*), 244
`fmprz_poly_print_pretty` (*C function*), 244
`fmprz_poly_product_roots_fmpz_vec` (*C function*), 246
`fmprz_poly_product_roots_fmpz_vec` (*C function*), 246
`fmprz_poly_pseudo_div` (*C function*), 233
`fmprz_poly_pseudo_divrem` (*C function*), 233
`fmprz_poly_pseudo_divrem_basecase` (*C function*), 232
`fmprz_poly_pseudo_divrem_cohen` (*C function*), 232
`fmprz_poly_pseudo_divrem_divconquer` (*C function*), 232
`fmprz_poly_pseudo_rem` (*C function*), 233
`fmprz_poly_pseudo_rem_cohen` (*C function*), 233
`fmprz_poly_q_add` (*C function*), 375
`fmprz_poly_q_addmul` (*C function*), 375
`fmprz_poly_q_canonicalise` (*C function*), 373
`fmprz_poly_q_clear` (*C function*), 373
`fmprz_poly_q_denref` (*C function*), 373
`fmprz_poly_q_derivative` (*C function*), 376
`fmprz_poly_q_div` (*C function*), 375
`fmprz_poly_q_equal` (*C function*), 374
`fmprz_poly_q_evaluate_fmpz` (*C function*), 376
`fmprz_poly_q_get_str` (*C function*), 376
`fmprz_poly_q_get_str_pretty` (*C function*), 376
`fmprz_poly_q_init` (*C function*), 373
`fmprz_poly_q_inv` (*C function*), 374
`fmprz_poly_q_is_canonical` (*C function*), 373
`fmprz_poly_q_is_one` (*C function*), 374
`fmprz_poly_q_is_zero` (*C function*), 374
`fmprz_poly_q_mul` (*C function*), 375
`fmprz_poly_q_neg` (*C function*), 374
`fmprz_poly_q_numref` (*C function*), 373
`fmprz_poly_q_one` (*C function*), 374
`fmprz_poly_q_pow` (*C function*), 375
`fmprz_poly_q_print` (*C function*), 376
`fmprz_poly_q_print_pretty` (*C function*), 376
`fmprz_poly_q_randtest` (*C function*), 374
`fmprz_poly_q_randtest_not_zero` (*C function*), 374
`fmprz_poly_q_scalar_div_fmpz` (*C function*), 375
`fmprz_poly_q_scalar_div_fmpz` (*C function*), 375
`fmprz_poly_q_scalar_div_si` (*C function*), 375
`fmprz_poly_q_scalar_mul_fmpz` (*C function*), 375
`fmprz_poly_q_scalar_mul_fmpz` (*C function*), 375
`fmprz_poly_q_scalar_mul_si` (*C function*), 375
`fmprz_poly_q_set` (*C function*), 374
`fmprz_poly_q_set_si` (*C function*), 374
`fmprz_poly_q_set_str` (*C function*), 376
`fmprz_poly_q_struct` (*C type*), 373
`fmprz_poly_q_sub` (*C function*), 375
`fmprz_poly_q_submul` (*C function*), 375
`fmprz_poly_q_swap` (*C function*), 374
`fmprz_poly_q_t` (*C type*), 373
`fmprz_poly_q_zero` (*C function*), 374
`fmprz_poly_randtest` (*C function*), 210
`fmprz_poly_randtest_irreducible` (*C function*), 210
`fmprz_poly_randtest_irreducible1` (*C function*), 210
`fmprz_poly_randtest_irreducible2` (*C function*), 210
`fmprz_poly_randtest_no_real_root` (*C function*), 210
`fmprz_poly_randtest_not_zero` (*C function*), 210
`fmprz_poly_randtest_unsigned` (*C function*), 210
`fmprz_poly_read` (*C function*), 245
`fmprz_poly_read_pretty` (*C function*), 245
`fmprz_poly_realloc` (*C function*), 208
`fmprz_poly_rem` (*C function*), 229
`fmprz_poly_rem_basecase` (*C function*), 228
`fmprz_poly_rem_powers_precomp` (*C function*), 230
`fmprz_poly_remove` (*C function*), 230
`fmprz_poly_resultant` (*C function*), 224
`fmprz_poly_resultant_euclidean` (*C function*), 224
`fmprz_poly_resultant_modular` (*C function*), 223
`fmprz_poly_resultant_modular_div` (*C function*), 223
`fmprz_poly_reverse` (*C function*), 210
`fmprz_poly_revert_series` (*C function*), 239
`fmprz_poly_scalar_abs` (*C function*), 213
`fmprz_poly_scalar_addmul_fmpz` (*C function*), 213
`fmprz_poly_scalar_addmul_si` (*C function*), 213
`fmprz_poly_scalar_addmul_ui` (*C function*), 213
`fmprz_poly_scalar_divexact_fmpz` (*C function*), 213
`fmprz_poly_scalar_divexact_si` (*C function*), 213
`fmprz_poly_scalar_divexact_ui` (*C function*), 213
`fmprz_poly_scalar_fdiv_2exp` (*C function*), 213
`fmprz_poly_scalar_fdiv_fmpz` (*C function*), 213
`fmprz_poly_scalar_fdiv_si` (*C function*), 213
`fmprz_poly_scalar_fdiv_ui` (*C function*), 213
`fmprz_poly_scalar_mod_fmpz` (*C function*), 213
`fmprz_poly_scalar_mul_2exp` (*C function*), 213
`fmprz_poly_scalar_mul_fmpz` (*C function*), 213
`fmprz_poly_scalar_mul_si` (*C function*), 213
`fmprz_poly_scalar_mul_ui` (*C function*), 213
`fmprz_poly_scalar_smod_fmpz` (*C function*), 213
`fmprz_poly_scalar_submul_fmpz` (*C function*), 213
`fmprz_poly_scalar_tdiv_2exp` (*C function*), 213
`fmprz_poly_scalar_tdiv_fmpz` (*C function*), 213

fmpz_poly_scalar_tdiv_si (*C function*), 213
 fmpz_poly_scalar_tdiv_ui (*C function*), 213
 fmpz_poly_set (*C function*), 209
 fmpz_poly_set_coeff_fmpz (*C function*), 211
 fmpz_poly_set_coeff_si (*C function*), 211
 fmpz_poly_set_coeff_ui (*C function*), 211
 fmpz_poly_set_fmpz (*C function*), 209
 fmpz_poly_set_nmod_poly (*C function*), 245
 fmpz_poly_set_nmod_poly_unsigned (*C function*), 245
 fmpz_poly_set_si (*C function*), 209
 fmpz_poly_set_str (*C function*), 209
 fmpz_poly_set_trunc (*C function*), 210
 fmpz_poly_set_ui (*C function*), 209
 fmpz_poly_shift_left (*C function*), 220
 fmpz_poly_shift_right (*C function*), 221
 fmpz_poly_signature (*C function*), 241
 fmpz_poly_sqr (*C function*), 218
 fmpz_poly_sqr_classical (*C function*), 218
 fmpz_poly_sqr_karatsuba (*C function*), 218
 fmpz_poly_sqr_KS (*C function*), 218
 fmpz_poly_sqr_low (*C function*), 219
 fmpz_poly_sqr_low_classical (*C function*), 219
 fmpz_poly_sqr_low_karatsuba_n (*C function*), 218
 fmpz_poly_sqr_low_KS (*C function*), 218
 fmpz_poly_sqrt (*C function*), 240
 fmpz_poly_sqrt_classical (*C function*), 240
 fmpz_poly_sqrt_divconquer (*C function*), 240
 fmpz_poly_sqrt_KS (*C function*), 240
 fmpz_poly_sqrt_series (*C function*), 240
 fmpz_poly_sqrtrem_classical (*C function*), 239
 fmpz_poly_sqrtrem_divconquer (*C function*), 239
 fmpz_poly_struct (*C type*), 208
 fmpz_poly_sub (*C function*), 212
 fmpz_poly_sub_series (*C function*), 212
 fmpz_poly_swap (*C function*), 210
 fmpz_poly_swinnerton_dyer (*C function*), 248
 fmpz_poly_t (*C type*), 208
 fmpz_poly_taylor_shift (*C function*), 238
 fmpz_poly_taylor_shift_divconquer (*C function*), 238
 fmpz_poly_taylor_shift_horner (*C function*), 238
 fmpz_poly_taylor_shift_multi_mod (*C function*), 238
 fmpz_poly_theta_qexp (*C function*), 250
 fmpz_poly_truncate (*C function*), 210
 fmpz_poly_xgcd (*C function*), 223
 fmpz_poly_xgcd_modular (*C function*), 222
 fmpz_poly_zero (*C function*), 209
 fmpz_poly_zero_coeffs (*C function*), 210
 fmpz_popcnt (*C function*), 170
 fmpz_pow_fmpz (*C function*), 166
 fmpz_pow_ui (*C function*), 166
 fmpz_powm (*C function*), 166
 fmpz_powm_ui (*C function*), 166
 fmpz_preinvn_clear (*C function*), 166
 fmpz_preinvn_init (*C function*), 165
 fmpz_primorial (*C function*), 175
 fmpz_print (*C function*), 161
 fmpz_randbits (*C function*), 157
 fmpz_randbits_unsigned (*C function*), 157
 fmpz_randm (*C function*), 158
 fmpz_randprime (*C function*), 158
 fmpz_randtest (*C function*), 157
 fmpz_randtest_mod (*C function*), 158
 fmpz_randtest_mod_signed (*C function*), 158
 fmpz_randtest_not_zero (*C function*), 158
 fmpz_randtest_unsigned (*C function*), 157
 fmpz_read (*C function*), 161
 fmpz_remove (*C function*), 168
 fmpz_rfac_ui (*C function*), 167
 fmpz_rfac_uiui (*C function*), 167
 fmpz_root (*C function*), 167
 fmpz_set (*C function*), 162
 fmpz_set_d (*C function*), 159
 fmpz_set_d_2exp (*C function*), 159
 fmpz_set_mpf (*C function*), 158
 fmpz_set_mpn_large (*C function*), 160
 fmpz_set_mpz (*C function*), 160
 fmpz_set_si (*C function*), 159
 fmpz_set_signed_ui_array (*C function*), 159
 fmpz_set_signed_uiui (*C function*), 159
 fmpz_set_signed_uiuiui (*C function*), 159
 fmpz_set_str (*C function*), 160
 fmpz_set_ui (*C function*), 159
 fmpz_set_ui_array (*C function*), 159
 fmpz_set_ui_smod (*C function*), 160
 fmpz_set_uiui (*C function*), 159
 fmpz_setbit (*C function*), 162
 fmpz_sgn (*C function*), 162
 fmpz_size (*C function*), 162
 fmpz_sizeinbase (*C function*), 162
 fmpz_smod (*C function*), 165
 fmpz_sqrt (*C function*), 166
 fmpz_sqrtmod (*C function*), 166
 fmpz_sqrtrem (*C function*), 166
 fmpz_sub (*C function*), 163
 fmpz_sub_si (*C function*), 163
 fmpz_sub_si_inline (*C function*), 818
 fmpz_sub_ui (*C function*), 163
 fmpz_submul (*C function*), 164
 fmpz_submul_si (*C function*), 164
 fmpz_submul_ui (*C function*), 164
 fmpz_swap (*C function*), 162
 fmpz_t (*C type*), 156
 fmpz_tdiv_q (*C function*), 164
 fmpz_tdiv_q_2exp (*C function*), 165
 fmpz_tdiv_q_si (*C function*), 164
 fmpz_tdiv_q_ui (*C function*), 164
 fmpz_tdiv_qr (*C function*), 164
 fmpz_tdiv_r_2exp (*C function*), 165
 fmpz_tdiv_ui (*C function*), 165
 fmpz_tstbit (*C function*), 162

`fmpz_ui_mul_ui` (*C function*), 818
`fmpz_ui_pow_ui` (*C function*), 166
`fmpz_val2` (*C function*), 162
`fmpz_xgcd` (*C function*), 167
`fmpz_xgcd_canonical_bezout` (*C function*), 168
`fmpz_xgcd_partial` (*C function*), 168
`fmpz_xor` (*C function*), 170
`fmpz_zero` (*C function*), 162
`fmpzi_add` (*C function*), 543
`fmpzi_bits` (*C function*), 543
`fmpzi_canonical_unit_i_pow` (*C function*), 543
`fmpzi_canonicalise_unit` (*C function*), 543
`fmpzi_clear` (*C function*), 542
`fmpzi_conj` (*C function*), 543
`fmpzi_divexact` (*C function*), 543
`fmpzi_divrem` (*C function*), 543
`fmpzi_divrem_approx` (*C function*), 543
`fmpzi_equal` (*C function*), 542
`fmpzi_gcd` (*C function*), 543
`fmpzi_gcd_binary` (*C function*), 543
`fmpzi_gcd_euclidean` (*C function*), 543
`fmpzi_gcd_euclidean_improved` (*C function*), 543
`fmpzi_gcd_shortest` (*C function*), 543
`fmpzi_imagref` (*C macro*), 542
`fmpzi_init` (*C function*), 542
`fmpzi_is_one` (*C function*), 542
`fmpzi_is_prime` (*C function*), 544
`fmpzi_is_probabprime` (*C function*), 544
`fmpzi_is_unit` (*C function*), 543
`fmpzi_is_zero` (*C function*), 542
`fmpzi_mul` (*C function*), 543
`fmpzi_neg` (*C function*), 543
`fmpzi_norm` (*C function*), 543
`fmpzi_one` (*C function*), 542
`fmpzi_pow_ui` (*C function*), 543
`fmpzi_print` (*C function*), 542
`fmpzi_randtest` (*C function*), 542
`fmpzi_realref` (*C macro*), 542
`fmpzi_remove_one_plus_i` (*C function*), 543
`fmpzi_set` (*C function*), 542
`fmpzi_set_si_si` (*C function*), 542
`fmpzi_sqr` (*C function*), 543
`fmpzi_struct` (*C type*), 542
`fmpzi_sub` (*C function*), 543
`fmpzi_swap` (*C function*), 542
`fmpzi_t` (*C type*), 542
`fmpzi_zero` (*C function*), 542
`For` (*C macro*), 896
`FormalLaurentSeries` (*C macro*), 905
`FormalPowerSeries` (*C macro*), 905
`FormalPuisseuxSeries` (*C macro*), 905
`fprint_memory_usage` (*C function*), 25
`FPWRAP_ACCURATE_PARTS` (*C macro*), 807
`FPWRAP_CORRECT_ROUNDING` (*C macro*), 807
`FPWRAP_SUCCESS` (*C macro*), 807
`FPWRAP_UNABLE` (*C macro*), 807
`FPWRAP_WORK_LIMIT` (*C macro*), 808
`fq_add` (*C function*), 917
`fq_bit_pack` (*C function*), 921
`fq_bit_unpack` (*C function*), 921
`fq_clear` (*C function*), 916
`fq_ctx_clear` (*C function*), 916
`fq_ctx_degree` (*C function*), 916
`fq_ctx_fprint` (*C function*), 916
`fq_ctx_init` (*C function*), 915
`fq_ctx_init_conway` (*C function*), 915
`fq_ctx_init_modulus` (*C function*), 916
`fq_ctx_init_randtest` (*C function*), 916
`fq_ctx_init_randtest_reducible` (*C function*), 916
`fq_ctx_modulus` (*C function*), 916
`fq_ctx_order` (*C function*), 916
`fq_ctx_prime` (*C function*), 916
`fq_ctx_print` (*C function*), 916
`fq_ctx_struct` (*C type*), 915
`fq_ctx_t` (*C type*), 915
`fq_default_add` (*C function*), 923
`fq_default_clear` (*C function*), 923
`fq_default_ctx_clear` (*C function*), 922
`fq_default_ctx_degree` (*C function*), 922
`fq_default_ctx_fprint` (*C function*), 922
`fq_default_ctx_init` (*C function*), 921
`fq_default_ctx_init_modulus` (*C function*), 921
`fq_default_ctx_init_modulus_nmod` (*C function*), 921
`fq_default_ctx_init_modulus_nmod_type` (*C function*), 921
`fq_default_ctx_init_modulus_type` (*C function*), 921
`fq_default_ctx_init_randtest` (*C function*), 922
`fq_default_ctx_init_type` (*C function*), 921
`fq_default_ctx_inner` (*C function*), 922
`fq_default_ctx_modulus` (*C function*), 922
`fq_default_ctx_order` (*C function*), 922
`fq_default_ctx_prime` (*C function*), 922
`fq_default_ctx_print` (*C function*), 922
`fq_default_ctx_t` (*C type*), 921
`fq_default_ctx_type` (*C function*), 922
`fq_default_div` (*C function*), 923
`fq_default_equal` (*C function*), 926
`fq_default_fprint` (*C function*), 924
`fq_default_fprint_pretty` (*C function*), 924
`fq_default_frobenius` (*C function*), 926
`fq_default_gen` (*C function*), 925
`fq_default_get_coeff_fmpz` (*C function*), 922
`fq_default_get_fmpz` (*C function*), 925
`fq_default_get_fmpz_mod_poly` (*C function*), 925
`fq_default_get_fmpz_poly` (*C function*), 926
`fq_default_get_nmod_poly` (*C function*), 925
`fq_default_get_str` (*C function*), 924
`fq_default_get_str_pretty` (*C function*), 924
`fq_default_init` (*C function*), 923
`fq_default_init2` (*C function*), 923

- `fq_default_inv` (*C function*), 923
- `fq_default_is_invertible` (*C function*), 923
- `fq_default_is_one` (*C function*), 926
- `fq_default_is_square` (*C function*), 924
- `fq_default_is_zero` (*C function*), 926
- `fq_default_mat_add` (*C function*), 938
- `fq_default_mat_can_solve` (*C function*), 939
- `fq_default_mat_charpoly` (*C function*), 940
- `fq_default_mat_clear` (*C function*), 934
- `fq_default_mat_concat_horizontal` (*C function*), 936
- `fq_default_mat_concat_vertical` (*C function*), 936
- `fq_default_mat_entry` (*C function*), 935
- `fq_default_mat_entry_set` (*C function*), 935
- `fq_default_mat_entry_set_fmpz` (*C function*), 935
- `fq_default_mat_equal` (*C function*), 937
- `fq_default_mat_fprint` (*C function*), 936
- `fq_default_mat_fprint_pretty` (*C function*), 936
- `fq_default_mat_init` (*C function*), 934
- `fq_default_mat_init_set` (*C function*), 934
- `fq_default_mat_inv` (*C function*), 938
- `fq_default_mat_invert_cols` (*C function*), 935
- `fq_default_mat_invert_rows` (*C function*), 935
- `fq_default_mat_is_empty` (*C function*), 937
- `fq_default_mat_is_one` (*C function*), 937
- `fq_default_mat_is_square` (*C function*), 937
- `fq_default_mat_is_zero` (*C function*), 937
- `fq_default_mat_lu` (*C function*), 938
- `fq_default_mat_minpoly` (*C function*), 940
- `fq_default_mat_mul` (*C function*), 938
- `fq_default_mat_ncols` (*C function*), 935
- `fq_default_mat_neg` (*C function*), 938
- `fq_default_mat_nrows` (*C function*), 935
- `fq_default_mat_one` (*C function*), 935
- `fq_default_mat_print` (*C function*), 936
- `fq_default_mat_print_pretty` (*C function*), 936
- `fq_default_mat_randops` (*C function*), 937
- `fq_default_mat_randpermdiag` (*C function*), 937
- `fq_default_mat_randrank` (*C function*), 937
- `fq_default_mat_randtest` (*C function*), 937
- `fq_default_mat_randtril` (*C function*), 937
- `fq_default_mat_randtriu` (*C function*), 937
- `fq_default_mat_rref` (*C function*), 939
- `fq_default_mat_set` (*C function*), 934
- `fq_default_mat_set_fmpz_mat` (*C function*), 935
- `fq_default_mat_set_fmpz_mod_mat` (*C function*), 935
- `fq_default_mat_set_nmod_mat` (*C function*), 935
- `fq_default_mat_similarity` (*C function*), 940
- `fq_default_mat_solve` (*C function*), 939
- `fq_default_mat_solve_tril` (*C function*), 939
- `fq_default_mat_solve_triu` (*C function*), 939
- `fq_default_mat_sub` (*C function*), 938
- `fq_default_mat_submul` (*C function*), 938
- `fq_default_mat_swap` (*C function*), 935
- `fq_default_mat_swap_cols` (*C function*), 935
- `fq_default_mat_swap_rows` (*C function*), 935
- `fq_default_mat_t` (*C type*), 934
- `fq_default_mat_transpose` (*C function*), 938
- `fq_default_mat_window_clear` (*C function*), 936
- `fq_default_mat_window_init` (*C function*), 936
- `fq_default_mat_zero` (*C function*), 935
- `fq_default_mul` (*C function*), 923
- `fq_default_mul_fmpz` (*C function*), 923
- `fq_default_mul_si` (*C function*), 923
- `fq_default_mul_ui` (*C function*), 923
- `fq_default_neg` (*C function*), 923
- `fq_default_norm` (*C function*), 926
- `fq_default_one` (*C function*), 925
- `fq_default_poly_add` (*C function*), 961
- `fq_default_poly_add_series` (*C function*), 961
- `fq_default_poly_add_si` (*C function*), 961
- `fq_default_poly_clear` (*C function*), 959
- `fq_default_poly_compose` (*C function*), 965
- `fq_default_poly_compose_mod` (*C function*), 965
- `fq_default_poly_deflate` (*C function*), 966
- `fq_default_poly_deflation` (*C function*), 966
- `fq_default_poly_degree` (*C function*), 959
- `fq_default_poly_derivative` (*C function*), 964
- `fq_default_poly_div_series` (*C function*), 963
- `fq_default_poly_divides` (*C function*), 964
- `fq_default_poly_divrem` (*C function*), 963
- `fq_default_poly_equal` (*C function*), 961
- `fq_default_poly_equal_fq_default` (*C function*), 961
- `fq_default_poly_equal_trunc` (*C function*), 961
- `fq_default_poly_evaluate_fq_default` (*C function*), 965
- `fq_default_poly_factor` (*C function*), 971
- `fq_default_poly_factor_clear` (*C function*), 969
- `fq_default_poly_factor_concat` (*C function*), 970
- `fq_default_poly_factor_distinct_deg` (*C function*), 970
- `fq_default_poly_factor_equal_deg` (*C function*), 970
- `fq_default_poly_factor_exp` (*C function*), 970
- `fq_default_poly_factor_fit_length` (*C function*), 969
- `fq_default_poly_factor_get_poly` (*C function*), 970
- `fq_default_poly_factor_init` (*C function*), 969
- `fq_default_poly_factor_insert` (*C function*), 969
- `fq_default_poly_factor_length` (*C function*), 970
- `fq_default_poly_factor_pow` (*C function*), 970
- `fq_default_poly_factor_print` (*C function*), 969
- `fq_default_poly_factor_print_pretty` (*C function*), 969

`fq_default_poly_factor_realloc` (*C function*), 969

`fq_default_poly_factor_set` (*C function*), 969

`fq_default_poly_factor_split_single` (*C function*), 970

`fq_default_poly_factor_squarefree` (*C function*), 971

`fq_default_poly_factor_t` (*C type*), 969

`fq_default_poly_fit_length` (*C function*), 959

`fq_default_poly_fprint` (*C function*), 965

`fq_default_poly_fprint_pretty` (*C function*), 965

`fq_default_poly_gcd` (*C function*), 964

`fq_default_poly_gen` (*C function*), 960

`fq_default_poly_get_coeff` (*C function*), 960

`fq_default_poly_get_str` (*C function*), 965

`fq_default_poly_get_str_pretty` (*C function*), 965

`fq_default_poly_hamming_weight` (*C function*), 963

`fq_default_poly_inflate` (*C function*), 966

`fq_default_poly_init` (*C function*), 958

`fq_default_poly_init2` (*C function*), 958

`fq_default_poly_inv_series` (*C function*), 963

`fq_default_poly_invsqrt_series` (*C function*), 964

`fq_default_poly_is_gen` (*C function*), 961

`fq_default_poly_is_irreducible` (*C function*), 970

`fq_default_poly_is_one` (*C function*), 961

`fq_default_poly_is_squarefree` (*C function*), 970

`fq_default_poly_is_unit` (*C function*), 961

`fq_default_poly_is_zero` (*C function*), 961

`fq_default_poly_length` (*C function*), 959

`fq_default_poly_make_monic` (*C function*), 960

`fq_default_poly_mul` (*C function*), 962

`fq_default_poly_mulhigh` (*C function*), 962

`fq_default_poly_mulalow` (*C function*), 962

`fq_default_poly_mulmod` (*C function*), 962

`fq_default_poly_neg` (*C function*), 961

`fq_default_poly_one` (*C function*), 960

`fq_default_poly_pow` (*C function*), 963

`fq_default_poly_pow_trunc` (*C function*), 963

`fq_default_poly_powmod_fmpz_binexp` (*C function*), 963

`fq_default_poly_powmod_ui_binexp` (*C function*), 963

`fq_default_poly_print` (*C function*), 965

`fq_default_poly_print_pretty` (*C function*), 965

`fq_default_poly_randtest` (*C function*), 959

`fq_default_poly_randtest_irreducible` (*C function*), 959

`fq_default_poly_randtest_monic` (*C function*), 959

`fq_default_poly_randtest_not_zero` (*C function*), 959

`fq_default_poly_realloc` (*C function*), 958

`fq_default_poly_rem` (*C function*), 963

`fq_default_poly_remove` (*C function*), 970

`fq_default_poly_reverse` (*C function*), 959

`fq_default_poly_roots` (*C function*), 971

`fq_default_poly_scalar_addmul_fq_default` (*C function*), 962

`fq_default_poly_scalar_div_fq_default` (*C function*), 962

`fq_default_poly_scalar_mul_fq_default` (*C function*), 962

`fq_default_poly_scalar_submul_fq_default` (*C function*), 962

`fq_default_poly_set` (*C function*), 960

`fq_default_poly_set_coeff` (*C function*), 960

`fq_default_poly_set_coeff_fmpz` (*C function*), 960

`fq_default_poly_set_fmpz_mod_poly` (*C function*), 960

`fq_default_poly_set_fmpz_poly` (*C function*), 960

`fq_default_poly_set_fq_default` (*C function*), 960

`fq_default_poly_set_nmod_poly` (*C function*), 960

`fq_default_poly_set_trunc` (*C function*), 959

`fq_default_poly_shift_left` (*C function*), 963

`fq_default_poly_shift_right` (*C function*), 963

`fq_default_poly_sqr` (*C function*), 962

`fq_default_poly_sqrt` (*C function*), 964

`fq_default_poly_sqrt_series` (*C function*), 964

`fq_default_poly_sub` (*C function*), 961

`fq_default_poly_sub_series` (*C function*), 961

`fq_default_poly_swap` (*C function*), 960

`fq_default_poly_t` (*C type*), 958

`fq_default_poly_truncate` (*C function*), 959

`fq_default_poly_xgcd` (*C function*), 964

`fq_default_poly_zero` (*C function*), 960

`fq_default_pow` (*C function*), 923

`fq_default_pow_ui` (*C function*), 924

`fq_default_print` (*C function*), 924

`fq_default_print_pretty` (*C function*), 924

`fq_default_pth_root` (*C function*), 924

`fq_default_rand` (*C function*), 925

`fq_default_rand_not_zero` (*C function*), 925

`fq_default_randtest` (*C function*), 925

`fq_default_randtest_not_zero` (*C function*), 925

`fq_default_set` (*C function*), 925

`fq_default_set_fmpz` (*C function*), 925

`fq_default_set_fmpz_mod_poly` (*C function*), 925

`fq_default_set_fmpz_poly` (*C function*), 926

`fq_default_set_nmod_poly` (*C function*), 925

`fq_default_set_si` (*C function*), 925

`fq_default_set_ui` (*C function*), 925

`fq_default_sqr` (*C function*), 923

`fq_default_sqrt` (*C function*), 924

fq_default_sub (*C function*), 923
 fq_default_sub_one (*C function*), 923
 fq_default_swap (*C function*), 925
 fq_default_t (*C type*), 921
 fq_default_trace (*C function*), 926
 fq_default_zero (*C function*), 925
 fq_div (*C function*), 917
 fq_embed_composition_matrix (*C function*), 972
 fq_embed_composition_matrix_sub (*C function*), 972
 fq_embed_dual_to_mono_matrix (*C function*), 972
 fq_embed_gens (*C function*), 971
 fq_embed_matrices (*C function*), 971
 fq_embed_mono_to_dual_matrix (*C function*), 972
 fq_embed_mul_matrix (*C function*), 972
 fq_embed_trace_matrix (*C function*), 972
 fq_equal (*C function*), 920
 fq_fprint (*C function*), 918
 fq_fprint_pretty (*C function*), 918
 fq_frobenius (*C function*), 920
 fq_gcdinv (*C function*), 917
 fq_gen (*C function*), 919
 fq_get_fmpz (*C function*), 919
 fq_get_fmpz_mod_mat (*C function*), 919
 fq_get_fmpz_mod_poly (*C function*), 919
 fq_get_fmpz_poly (*C function*), 919
 fq_get_str (*C function*), 918
 fq_get_str_pretty (*C function*), 918
 fq_init (*C function*), 916
 fq_init2 (*C function*), 916
 fq_inv (*C function*), 917
 fq_is_invertible (*C function*), 920
 fq_is_invertible_f (*C function*), 920
 fq_is_one (*C function*), 920
 fq_is_primitive (*C function*), 921
 fq_is_square (*C function*), 918
 fq_is_zero (*C function*), 920
 fq_mat_add (*C function*), 931
 fq_mat_can_solve (*C function*), 933
 fq_mat_charpoly (*C function*), 934
 fq_mat_clear (*C function*), 928
 fq_mat_concat_horizontal (*C function*), 930
 fq_mat_concat_vertical (*C function*), 930
 fq_mat_entry (*C function*), 929
 fq_mat_entry_set (*C function*), 929
 fq_mat_equal (*C function*), 931
 fq_mat_fprint (*C function*), 930
 fq_mat_fprint_pretty (*C function*), 930
 fq_mat_init (*C function*), 928
 fq_mat_init_set (*C function*), 928
 fq_mat_inv (*C function*), 932
 fq_mat_invert_cols (*C function*), 929
 fq_mat_invert_rows (*C function*), 929
 fq_mat_is_empty (*C function*), 931
 fq_mat_is_one (*C function*), 931
 fq_mat_is_square (*C function*), 931
 fq_mat_is_zero (*C function*), 931
 fq_mat_lu (*C function*), 932
 fq_mat_minpoly (*C function*), 934
 fq_mat_mul (*C function*), 932
 fq_mat_mul_classical (*C function*), 932
 fq_mat_mul_KS (*C function*), 932
 fq_mat_mul_vec (*C function*), 932
 fq_mat_mul_vec_ptr (*C function*), 932
 fq_mat_ncols (*C function*), 929
 fq_mat_neg (*C function*), 931
 fq_mat_nrows (*C function*), 929
 fq_mat_one (*C function*), 929
 fq_mat_print (*C function*), 930
 fq_mat_print_pretty (*C function*), 930
 fq_mat_randops (*C function*), 931
 fq_mat_randpermdiag (*C function*), 930
 fq_mat_randrank (*C function*), 930
 fq_mat_randtest (*C function*), 930
 fq_mat_randtril (*C function*), 931
 fq_mat_randtriu (*C function*), 931
 fq_mat_reduce_row (*C function*), 933
 fq_mat_rref (*C function*), 933
 fq_mat_set (*C function*), 928
 fq_mat_set_fmpz_mod_mat (*C function*), 929
 fq_mat_set_nmod_mat (*C function*), 929
 fq_mat_similarity (*C function*), 934
 fq_mat_solve (*C function*), 933
 fq_mat_solve_tril (*C function*), 933
 fq_mat_solve_triu (*C function*), 933
 fq_mat_struct (*C type*), 928
 fq_mat_sub (*C function*), 931
 fq_mat_submul (*C function*), 932
 fq_mat_swap (*C function*), 929
 fq_mat_swap_cols (*C function*), 929
 fq_mat_swap_entrywise (*C function*), 929
 fq_mat_swap_rows (*C function*), 929
 fq_mat_t (*C type*), 928
 fq_mat_transpose (*C function*), 931
 fq_mat_vec_mul (*C function*), 932
 fq_mat_vec_mul_ptr (*C function*), 932
 fq_mat_window_clear (*C function*), 930
 fq_mat_window_init (*C function*), 930
 fq_mat_zero (*C function*), 929
 fq_modulus_derivative_inv (*C function*), 972
 fq_modulus_pow_series_inv (*C function*), 972
 fq_mul (*C function*), 917
 fq_mul_fmpz (*C function*), 917
 fq_mul_si (*C function*), 917
 fq_mul_ui (*C function*), 917
 fq_multiplicative_order (*C function*), 920
 fq_neg (*C function*), 917
 fq_nmod_add (*C function*), 975
 fq_nmod_bit_pack (*C function*), 979
 fq_nmod_bit_unpack (*C function*), 979
 fq_nmod_clear (*C function*), 974
 fq_nmod_cmp (*C function*), 978
 fq_nmod_ctx_clear (*C function*), 974
 fq_nmod_ctx_degree (*C function*), 974

`fq_nmod_ctx_fprint` (*C function*), 974
`fq_nmod_ctx_init_conway_ui` (*C function*), 973
`fq_nmod_ctx_init_minimal_weight_ui` (*C function*), 973
`fq_nmod_ctx_init_modulus` (*C function*), 973
`fq_nmod_ctx_init_randtest` (*C function*), 974
`fq_nmod_ctx_init_randtest_reducible` (*C function*), 974
`fq_nmod_ctx_init_ui` (*C function*), 973
`fq_nmod_ctx_modulus` (*C function*), 974
`fq_nmod_ctx_order` (*C function*), 974
`fq_nmod_ctx_prime` (*C function*), 974
`fq_nmod_ctx_print` (*C function*), 974
`fq_nmod_ctx_struct` (*C type*), 973
`fq_nmod_ctx_t` (*C type*), 973
`fq_nmod_embed_composition_matrix` (*C function*), 1012
`fq_nmod_embed_composition_matrix_sub` (*C function*), 1012
`fq_nmod_embed_dual_to_mono_matrix` (*C function*), 1012
`fq_nmod_embed_gens` (*C function*), 1011
`fq_nmod_embed_matrices` (*C function*), 1011
`fq_nmod_embed_mono_to_dual_matrix` (*C function*), 1012
`fq_nmod_embed_mul_matrix` (*C function*), 1012
`fq_nmod_embed_trace_matrix` (*C function*), 1011
`fq_nmod_equal` (*C function*), 978
`fq_nmod_fprint` (*C function*), 976
`fq_nmod_fprint_pretty` (*C function*), 976
`fq_nmod_frobenius` (*C function*), 978
`fq_nmod_gcdinv` (*C function*), 975
`fq_nmod_gen` (*C function*), 977
`fq_nmod_get_fmpz` (*C function*), 977
`fq_nmod_get_nmod_mat` (*C function*), 977
`fq_nmod_get_nmod_poly` (*C function*), 977
`fq_nmod_get_str` (*C function*), 976
`fq_nmod_get_str_pretty` (*C function*), 976
`fq_nmod_init` (*C function*), 974
`fq_nmod_init2` (*C function*), 974
`fq_nmod_inv` (*C function*), 975
`fq_nmod_is_invertible` (*C function*), 978
`fq_nmod_is_invertible_f` (*C function*), 978
`fq_nmod_is_one` (*C function*), 978
`fq_nmod_is_primitive` (*C function*), 979
`fq_nmod_is_square` (*C function*), 976
`fq_nmod_is_zero` (*C function*), 978
`fq_nmod_mat_add` (*C function*), 984
`fq_nmod_mat_can_solve` (*C function*), 986
`fq_nmod_mat_charpoly` (*C function*), 987
`fq_nmod_mat_clear` (*C function*), 981
`fq_nmod_mat_concat_horizontal` (*C function*), 982
`fq_nmod_mat_concat_vertical` (*C function*), 982
`fq_nmod_mat_entry` (*C function*), 981
`fq_nmod_mat_entry_set` (*C function*), 981
`fq_nmod_mat_equal` (*C function*), 984
`fq_nmod_mat_fprint` (*C function*), 983
`fq_nmod_mat_fprint_pretty` (*C function*), 983
`fq_nmod_mat_init` (*C function*), 981
`fq_nmod_mat_init_set` (*C function*), 981
`fq_nmod_mat_inv` (*C function*), 985
`fq_nmod_mat_invert_cols` (*C function*), 982
`fq_nmod_mat_invert_rows` (*C function*), 982
`fq_nmod_mat_is_empty` (*C function*), 984
`fq_nmod_mat_is_one` (*C function*), 984
`fq_nmod_mat_is_square` (*C function*), 984
`fq_nmod_mat_is_zero` (*C function*), 984
`fq_nmod_mat_lu` (*C function*), 985
`fq_nmod_mat_minpoly` (*C function*), 987
`fq_nmod_mat_mul` (*C function*), 985
`fq_nmod_mat_mul_classical` (*C function*), 985
`fq_nmod_mat_mul_KS` (*C function*), 985
`fq_nmod_mat_mul_vec` (*C function*), 985
`fq_nmod_mat_mul_vec_ptr` (*C function*), 985
`fq_nmod_mat_ncols` (*C function*), 981
`fq_nmod_mat_neg` (*C function*), 984
`fq_nmod_mat_nrows` (*C function*), 981
`fq_nmod_mat_one` (*C function*), 982
`fq_nmod_mat_print` (*C function*), 983
`fq_nmod_mat_print_pretty` (*C function*), 983
`fq_nmod_mat_randops` (*C function*), 983
`fq_nmod_mat_randpermdiag` (*C function*), 983
`fq_nmod_mat_randrank` (*C function*), 983
`fq_nmod_mat_randtest` (*C function*), 983
`fq_nmod_mat_randtril` (*C function*), 983
`fq_nmod_mat_randtriu` (*C function*), 983
`fq_nmod_mat_reduce_row` (*C function*), 986
`fq_nmod_mat_rref` (*C function*), 986
`fq_nmod_mat_set` (*C function*), 981
`fq_nmod_mat_set_fmpz_mod_mat` (*C function*), 982
`fq_nmod_mat_set_nmod_mat` (*C function*), 982
`fq_nmod_mat_similarity` (*C function*), 987
`fq_nmod_mat_solve` (*C function*), 986
`fq_nmod_mat_solve_tril` (*C function*), 986
`fq_nmod_mat_solve_triu` (*C function*), 986
`fq_nmod_mat_struct` (*C type*), 981
`fq_nmod_mat_sub` (*C function*), 984
`fq_nmod_mat_submul` (*C function*), 985
`fq_nmod_mat_swap` (*C function*), 981
`fq_nmod_mat_swap_cols` (*C function*), 982
`fq_nmod_mat_swap_entrywise` (*C function*), 981
`fq_nmod_mat_swap_rows` (*C function*), 982
`fq_nmod_mat_t` (*C type*), 981
`fq_nmod_mat_transpose` (*C function*), 984
`fq_nmod_mat_vec_mul` (*C function*), 985
`fq_nmod_mat_vec_mul_ptr` (*C function*), 985
`fq_nmod_mat_window_clear` (*C function*), 983
`fq_nmod_mat_window_init` (*C function*), 983
`fq_nmod_mat_zero` (*C function*), 982
`fq_nmod_modulus_derivative_inv` (*C function*), 1012
`fq_nmod_modulus_pow_series_inv` (*C function*), 1012
`fq_nmod_mpoly_add` (*C function*), 1018

`fq_nmod_mpoly_add_fq_nmod` (*C function*), 1018
`fq_nmod_mpoly_clear` (*C function*), 1013
`fq_nmod_mpoly_cmp` (*C function*), 1016
`fq_nmod_mpoly_combine_like_terms` (*C function*), 1017
`fq_nmod_mpoly_compose_fq_nmod_mpoly` (*C function*), 1019
`fq_nmod_mpoly_compose_fq_nmod_mpoly_gen` (*C function*), 1019
`fq_nmod_mpoly_compose_fq_nmod_poly` (*C function*), 1019
`fq_nmod_mpoly_content_vars` (*C function*), 1020
`fq_nmod_mpoly_ctx_clear` (*C function*), 1013
`fq_nmod_mpoly_ctx_init` (*C function*), 1013
`fq_nmod_mpoly_ctx_nvars` (*C function*), 1013
`fq_nmod_mpoly_ctx_ord` (*C function*), 1013
`fq_nmod_mpoly_ctx_struct` (*C type*), 1012
`fq_nmod_mpoly_ctx_t` (*C type*), 1012
`fq_nmod_mpoly_degree_fmpz` (*C function*), 1015
`fq_nmod_mpoly_degree_si` (*C function*), 1015
`fq_nmod_mpoly_degrees_fit_si` (*C function*), 1015
`fq_nmod_mpoly_degrees_fmpz` (*C function*), 1015
`fq_nmod_mpoly_degrees_si` (*C function*), 1015
`fq_nmod_mpoly_derivative` (*C function*), 1019
`fq_nmod_mpoly_discriminant` (*C function*), 1021
`fq_nmod_mpoly_div` (*C function*), 1020
`fq_nmod_mpoly_divides` (*C function*), 1020
`fq_nmod_mpoly_divrem` (*C function*), 1020
`fq_nmod_mpoly_divrem_ideal` (*C function*), 1020
`fq_nmod_mpoly_equal` (*C function*), 1014
`fq_nmod_mpoly_equal_fq_nmod` (*C function*), 1014
`fq_nmod_mpoly_evaluate_all_fq_nmod` (*C function*), 1019
`fq_nmod_mpoly_evaluate_one_fq_nmod` (*C function*), 1019
`fq_nmod_mpoly_factor` (*C function*), 1023
`fq_nmod_mpoly_factor_clear` (*C function*), 1022
`fq_nmod_mpoly_factor_get_base` (*C function*), 1022
`fq_nmod_mpoly_factor_get_constant_fq_nmod` (*C function*), 1022
`fq_nmod_mpoly_factor_get_exp_si` (*C function*), 1023
`fq_nmod_mpoly_factor_init` (*C function*), 1022
`fq_nmod_mpoly_factor_length` (*C function*), 1022
`fq_nmod_mpoly_factor_sort` (*C function*), 1023
`fq_nmod_mpoly_factor_squarefree` (*C function*), 1023
`fq_nmod_mpoly_factor_struct` (*C type*), 1022
`fq_nmod_mpoly_factor_swap` (*C function*), 1022
`fq_nmod_mpoly_factor_swap_base` (*C function*), 1022
`fq_nmod_mpoly_factor_t` (*C type*), 1022
`fq_nmod_mpoly_fit_length` (*C function*), 1013
`fq_nmod_mpoly_fprint_pretty` (*C function*), 1013
`fq_nmod_mpoly_from_univar` (*C function*), 1021
`fq_nmod_mpoly_gcd` (*C function*), 1020
`fq_nmod_mpoly_gcd_brown` (*C function*), 1020
`fq_nmod_mpoly_gcd_cofactors` (*C function*), 1020
`fq_nmod_mpoly_gcd_hensel` (*C function*), 1020
`fq_nmod_mpoly_gcd_zippel` (*C function*), 1020
`fq_nmod_mpoly_gen` (*C function*), 1014
`fq_nmod_mpoly_get_coeff_fq_nmod_fmpz` (*C function*), 1015
`fq_nmod_mpoly_get_coeff_fq_nmod_monomial` (*C function*), 1015
`fq_nmod_mpoly_get_coeff_fq_nmod_ui` (*C function*), 1015
`fq_nmod_mpoly_get_coeff_vars_ui` (*C function*), 1016
`fq_nmod_mpoly_get_fq_nmod` (*C function*), 1014
`fq_nmod_mpoly_get_str_pretty` (*C function*), 1013
`fq_nmod_mpoly_get_term` (*C function*), 1017
`fq_nmod_mpoly_get_term_coeff_fq_nmod` (*C function*), 1016
`fq_nmod_mpoly_get_term_exp_fmpz` (*C function*), 1017
`fq_nmod_mpoly_get_term_exp_si` (*C function*), 1017
`fq_nmod_mpoly_get_term_exp_ui` (*C function*), 1017
`fq_nmod_mpoly_get_term_monomial` (*C function*), 1017
`fq_nmod_mpoly_get_term_var_exp_si` (*C function*), 1017
`fq_nmod_mpoly_get_term_var_exp_ui` (*C function*), 1017
`fq_nmod_mpoly_init` (*C function*), 1013
`fq_nmod_mpoly_init2` (*C function*), 1013
`fq_nmod_mpoly_init3` (*C function*), 1013
`fq_nmod_mpoly_is_canonical` (*C function*), 1016
`fq_nmod_mpoly_is_fq_nmod` (*C function*), 1014
`fq_nmod_mpoly_is_gen` (*C function*), 1014
`fq_nmod_mpoly_is_one` (*C function*), 1015
`fq_nmod_mpoly_is_square` (*C function*), 1021
`fq_nmod_mpoly_is_zero` (*C function*), 1014
`fq_nmod_mpoly_length` (*C function*), 1016
`fq_nmod_mpoly_make_monic` (*C function*), 1018
`fq_nmod_mpoly_mul` (*C function*), 1019
`fq_nmod_mpoly_neg` (*C function*), 1018
`fq_nmod_mpoly_one` (*C function*), 1014
`fq_nmod_mpoly_pow_fmpz` (*C function*), 1020
`fq_nmod_mpoly_pow_ui` (*C function*), 1020
`fq_nmod_mpoly_print_pretty` (*C function*), 1013
`fq_nmod_mpoly_push_term_fq_nmod_ffmpz` (*C function*), 1017
`fq_nmod_mpoly_push_term_fq_nmod_fmpz` (*C function*), 1017
`fq_nmod_mpoly_push_term_fq_nmod_ui` (*C function*), 1017

`fq_nmod_mpoly_quadratic_root` (*C function*), 1021

`fq_nmod_mpoly_randtest_bits` (*C function*), 1018

`fq_nmod_mpoly_randtest_bound` (*C function*), 1018

`fq_nmod_mpoly_randtest_bounds` (*C function*), 1018

`fq_nmod_mpoly_realloc` (*C function*), 1013

`fq_nmod_mpoly_resize` (*C function*), 1016

`fq_nmod_mpoly_resultant` (*C function*), 1021

`fq_nmod_mpoly_reverse` (*C function*), 1017

`fq_nmod_mpoly_scalar_mul_fq_nmod` (*C function*), 1018

`fq_nmod_mpoly_set` (*C function*), 1014

`fq_nmod_mpoly_set_coeff_fq_nmod_fmpz` (*C function*), 1016

`fq_nmod_mpoly_set_coeff_fq_nmod_monomial` (*C function*), 1015

`fq_nmod_mpoly_set_coeff_fq_nmod_ui` (*C function*), 1016

`fq_nmod_mpoly_set_fq_nmod` (*C function*), 1014

`fq_nmod_mpoly_set_fq_nmod_gen` (*C function*), 1014

`fq_nmod_mpoly_set_str_pretty` (*C function*), 1013

`fq_nmod_mpoly_set_term_coeff_ui` (*C function*), 1016

`fq_nmod_mpoly_set_term_exp_fmpz` (*C function*), 1017

`fq_nmod_mpoly_set_term_exp_ui` (*C function*), 1017

`fq_nmod_mpoly_set_ui` (*C function*), 1014

`fq_nmod_mpoly_sort_terms` (*C function*), 1017

`fq_nmod_mpoly_sqrt` (*C function*), 1021

`fq_nmod_mpoly_struct` (*C type*), 1012

`fq_nmod_mpoly_sub` (*C function*), 1018

`fq_nmod_mpoly_sub_fq_nmod` (*C function*), 1018

`fq_nmod_mpoly_swap` (*C function*), 1014

`fq_nmod_mpoly_t` (*C type*), 1012

`fq_nmod_mpoly_term_content` (*C function*), 1020

`fq_nmod_mpoly_term_exp_fits_si` (*C function*), 1016

`fq_nmod_mpoly_term_exp_fits_ui` (*C function*), 1016

`fq_nmod_mpoly_to_univar` (*C function*), 1021

`fq_nmod_mpoly_total_degree_fits_si` (*C function*), 1015

`fq_nmod_mpoly_total_degree_fmpz` (*C function*), 1015

`fq_nmod_mpoly_total_degree_si` (*C function*), 1015

`fq_nmod_mpoly_univar_clear` (*C function*), 1021

`fq_nmod_mpoly_univar_degree_fits_si` (*C function*), 1021

`fq_nmod_mpoly_univar_get_term_coeff` (*C function*), 1022

`fq_nmod_mpoly_univar_get_term_exp_si` (*C function*), 1022

`fq_nmod_mpoly_univar_init` (*C function*), 1021

`fq_nmod_mpoly_univar_length` (*C function*), 1022

`fq_nmod_mpoly_univar_swap` (*C function*), 1021

`fq_nmod_mpoly_univar_swap_term_coeff` (*C function*), 1022

`fq_nmod_mpoly_used_vars` (*C function*), 1015

`fq_nmod_mpoly_zero` (*C function*), 1014

`fq_nmod_mul` (*C function*), 975

`fq_nmod_mul_fmpz` (*C function*), 975

`fq_nmod_mul_si` (*C function*), 975

`fq_nmod_mul_ui` (*C function*), 975

`fq_nmod_multiplicative_order` (*C function*), 978

`fq_nmod_neg` (*C function*), 975

`fq_nmod_norm` (*C function*), 978

`fq_nmod_one` (*C function*), 977

`fq_nmod_poly_add` (*C function*), 991

`fq_nmod_poly_add_series` (*C function*), 991

`fq_nmod_poly_add_si` (*C function*), 991

`fq_nmod_poly_clear` (*C function*), 988

`fq_nmod_poly_compose` (*C function*), 1003

`fq_nmod_poly_compose_mod` (*C function*), 1005

`fq_nmod_poly_compose_mod_brent_kung` (*C function*), 1004

`fq_nmod_poly_compose_mod_brent_kung_precomp_preinv` (*C function*), 1006

`fq_nmod_poly_compose_mod_brent_kung_preinv` (*C function*), 1004

`fq_nmod_poly_compose_mod_horner` (*C function*), 1004

`fq_nmod_poly_compose_mod_horner_preinv` (*C function*), 1004

`fq_nmod_poly_compose_mod_preinv` (*C function*), 1005

`fq_nmod_poly_deflate` (*C function*), 1007

`fq_nmod_poly_deflation` (*C function*), 1007

`fq_nmod_poly_degree` (*C function*), 988

`fq_nmod_poly_derivative` (*C function*), 1002

`fq_nmod_poly_div` (*C function*), 999

`fq_nmod_poly_div_newton_n_preinv` (*C function*), 999

`fq_nmod_poly_div_series` (*C function*), 1000

`fq_nmod_poly_divides` (*C function*), 1002

`fq_nmod_poly_divrem` (*C function*), 998

`fq_nmod_poly_divrem_f` (*C function*), 999

`fq_nmod_poly_divrem_newton_n_preinv` (*C function*), 1000

`fq_nmod_poly_equal` (*C function*), 990

`fq_nmod_poly_equal_fq_nmod` (*C function*), 990

`fq_nmod_poly_equal_trunc` (*C function*), 990

`fq_nmod_poly_evaluate_fq_nmod` (*C function*), 1003

`fq_nmod_poly_factor` (*C function*), 1009

`fq_nmod_poly_factor_berlekamp` (*C function*), 1010

`fq_nmod_poly_factor_cantor_zassenhaus` (*C function*), 1009
`fq_nmod_poly_factor_clear` (*C function*), 1008
`fq_nmod_poly_factor_concat` (*C function*), 1008
`fq_nmod_poly_factor_distinct_deg` (*C function*), 1009
`fq_nmod_poly_factor_equal_deg` (*C function*), 1009
`fq_nmod_poly_factor_equal_deg_prob` (*C function*), 1009
`fq_nmod_poly_factor_fit_length` (*C function*), 1008
`fq_nmod_poly_factor_init` (*C function*), 1008
`fq_nmod_poly_factor_insert` (*C function*), 1008
`fq_nmod_poly_factor_kaltofen_shoup` (*C function*), 1010
`fq_nmod_poly_factor_pow` (*C function*), 1008
`fq_nmod_poly_factor_print` (*C function*), 1008
`fq_nmod_poly_factor_print_pretty` (*C function*), 1008
`fq_nmod_poly_factor_realloc` (*C function*), 1008
`fq_nmod_poly_factor_set` (*C function*), 1008
`fq_nmod_poly_factor_split_single` (*C function*), 1009
`fq_nmod_poly_factor_squarefree` (*C function*), 1009
`fq_nmod_poly_factor_struct` (*C type*), 1007
`fq_nmod_poly_factor_t` (*C type*), 1007
`fq_nmod_poly_factor_with_berlekamp` (*C function*), 1010
`fq_nmod_poly_factor_with_cantor_zassenhaus` (*C function*), 1010
`fq_nmod_poly_factor_with_kaltofen_shoup` (*C function*), 1010
`fq_nmod_poly_fit_length` (*C function*), 988
`fq_nmod_poly_fprint` (*C function*), 1006
`fq_nmod_poly_fprint_pretty` (*C function*), 1006
`fq_nmod_poly_gcd` (*C function*), 1001
`fq_nmod_poly_gcd_euclidean_f` (*C function*), 1001
`fq_nmod_poly_gen` (*C function*), 989
`fq_nmod_poly_get_coeff` (*C function*), 990
`fq_nmod_poly_get_str` (*C function*), 1007
`fq_nmod_poly_get_str_pretty` (*C function*), 1007
`fq_nmod_poly_hamming_weight` (*C function*), 998
`fq_nmod_poly_inflate` (*C function*), 1007
`fq_nmod_poly_init` (*C function*), 987
`fq_nmod_poly_init2` (*C function*), 987
`fq_nmod_poly_inv_series` (*C function*), 1000
`fq_nmod_poly_inv_series_newton` (*C function*), 1000
`fq_nmod_poly_invsqrt_series` (*C function*), 1003
`fq_nmod_poly_is_gen` (*C function*), 990
`fq_nmod_poly_is_irreducible` (*C function*), 1009
`fq_nmod_poly_is_irreducible_ben_or` (*C function*), 1009
`fq_nmod_poly_is_irreducible_ddf` (*C function*), 1009
`fq_nmod_poly_is_one` (*C function*), 990
`fq_nmod_poly_is_squarefree` (*C function*), 1009
`fq_nmod_poly_is_unit` (*C function*), 990
`fq_nmod_poly_is_zero` (*C function*), 990
`fq_nmod_poly_iterated_frobenius_preinv` (*C function*), 1010
`fq_nmod_poly_lead` (*C function*), 988
`fq_nmod_poly_length` (*C function*), 988
`fq_nmod_poly_make_monic` (*C function*), 989
`fq_nmod_poly_mul` (*C function*), 993
`fq_nmod_poly_mul_classical` (*C function*), 992
`fq_nmod_poly_mul_KS` (*C function*), 993
`fq_nmod_poly_mul_reorder` (*C function*), 992
`fq_nmod_poly_mul_univariate` (*C function*), 993
`fq_nmod_poly_mulhigh` (*C function*), 994
`fq_nmod_poly_mulhigh_classical` (*C function*), 994
`fq_nmod_poly_mulmod` (*C function*), 994
`fq_nmod_poly_mulmod_preinv` (*C function*), 995
`fq_nmod_poly_neg` (*C function*), 991
`fq_nmod_poly_one` (*C function*), 989
`fq_nmod_poly_pow` (*C function*), 996
`fq_nmod_poly_pow_trunc` (*C function*), 998
`fq_nmod_poly_pow_trunc_binexp` (*C function*), 997
`fq_nmod_poly_powmod_fmpz_binexp` (*C function*), 996
`fq_nmod_poly_powmod_fmpz_binexp_preinv` (*C function*), 997
`fq_nmod_poly_powmod_fmpz_sliding_preinv` (*C function*), 997
`fq_nmod_poly_powmod_ui_binexp` (*C function*), 996
`fq_nmod_poly_powmod_ui_binexp_preinv` (*C function*), 996
`fq_nmod_poly_powmod_x_fmpz_preinv` (*C function*), 997
`fq_nmod_poly_precompute_matrix` (*C function*), 1005
`fq_nmod_poly_print` (*C function*), 1007
`fq_nmod_poly_print_pretty` (*C function*), 1006
`fq_nmod_poly_randtest` (*C function*), 989
`fq_nmod_poly_randtest_irreducible` (*C function*), 989
`fq_nmod_poly_randtest_monic` (*C function*), 989
`fq_nmod_poly_randtest_not_zero` (*C function*), 989
`fq_nmod_poly_realloc` (*C function*), 987

`fq_nmod_poly_rem` (*C function*), 999
`fq_nmod_poly_remove` (*C function*), 1008
`fq_nmod_poly_reverse` (*C function*), 988
`fq_nmod_poly_roots` (*C function*), 1011
`fq_nmod_poly_scalar_addmul_fq_nmod` (*C function*), 991
`fq_nmod_poly_scalar_div_fq` (*C function*), 992
`fq_nmod_poly_scalar_mul_fq_nmod` (*C function*), 991
`fq_nmod_poly_scalar_submul_fq_nmod` (*C function*), 992
`fq_nmod_poly_set` (*C function*), 989
`fq_nmod_poly_set_coeff` (*C function*), 990
`fq_nmod_poly_set_coeff_fmpz` (*C function*), 990
`fq_nmod_poly_set_fmpz_mod_poly` (*C function*), 989
`fq_nmod_poly_set_fq_nmod` (*C function*), 989
`fq_nmod_poly_set_nmod_poly` (*C function*), 989
`fq_nmod_poly_set_trunc` (*C function*), 988
`fq_nmod_poly_shift_left` (*C function*), 998
`fq_nmod_poly_shift_right` (*C function*), 998
`fq_nmod_poly_sqr` (*C function*), 995
`fq_nmod_poly_sqr_classical` (*C function*), 995
`fq_nmod_poly_sqr_KS` (*C function*), 995
`fq_nmod_poly_sqrt` (*C function*), 1003
`fq_nmod_poly_sqrt_series` (*C function*), 1003
`fq_nmod_poly_struct` (*C type*), 987
`fq_nmod_poly_sub` (*C function*), 991
`fq_nmod_poly_sub_series` (*C function*), 991
`fq_nmod_poly_swap` (*C function*), 989
`fq_nmod_poly_t` (*C type*), 987
`fq_nmod_poly_truncate` (*C function*), 988
`fq_nmod_poly_xgcd` (*C function*), 1001
`fq_nmod_poly_xgcd_euclidean_f` (*C function*), 1002
`fq_nmod_poly_zero` (*C function*), 989
`fq_nmod_pow` (*C function*), 975
`fq_nmod_pow_ui` (*C function*), 975
`fq_nmod_print` (*C function*), 976
`fq_nmod_print_pretty` (*C function*), 976
`fq_nmod_pth_root` (*C function*), 976
`fq_nmod_rand` (*C function*), 976
`fq_nmod_rand_not_zero` (*C function*), 977
`fq_nmod_randtest` (*C function*), 976
`fq_nmod_randtest_dense` (*C function*), 976
`fq_nmod_randtest_not_zero` (*C function*), 976
`fq_nmod_reduce` (*C function*), 974
`fq_nmod_set` (*C function*), 977
`fq_nmod_set_fmpz` (*C function*), 977
`fq_nmod_set_nmod_mat` (*C function*), 977
`fq_nmod_set_nmod_poly` (*C function*), 977
`fq_nmod_set_si` (*C function*), 977
`fq_nmod_set_ui` (*C function*), 977
`fq_nmod_sqr` (*C function*), 975
`fq_nmod_sqrt` (*C function*), 976
`fq_nmod_struct` (*C type*), 973
`fq_nmod_sub` (*C function*), 975
`fq_nmod_sub_one` (*C function*), 975
`fq_nmod_swap` (*C function*), 977
`fq_nmod_t` (*C type*), 973
`fq_nmod_trace` (*C function*), 978
`fq_nmod_zero` (*C function*), 977
`fq_norm` (*C function*), 920
`fq_one` (*C function*), 919
`fq_poly_add` (*C function*), 943
`fq_poly_add_series` (*C function*), 943
`fq_poly_add_si` (*C function*), 943
`fq_poly_clear` (*C function*), 941
`fq_poly_compose` (*C function*), 955
`fq_poly_compose_mod` (*C function*), 956
`fq_poly_compose_mod_brent_kung` (*C function*), 956
`fq_poly_compose_mod_brent_kung_precomp_preinv` (*C function*), 957
`fq_poly_compose_mod_brent_kung_preinv` (*C function*), 956
`fq_poly_compose_mod_horner` (*C function*), 955
`fq_poly_compose_mod_horner_preinv` (*C function*), 955
`fq_poly_compose_mod_preinv` (*C function*), 956
`fq_poly_deflate` (*C function*), 958
`fq_poly_deflation` (*C function*), 958
`fq_poly_degree` (*C function*), 941
`fq_poly_derivative` (*C function*), 954
`fq_poly_div` (*C function*), 951
`fq_poly_div_newton_n_preinv` (*C function*), 951
`fq_poly_div_series` (*C function*), 952
`fq_poly_divides` (*C function*), 954
`fq_poly_divrem` (*C function*), 950
`fq_poly_divrem_f` (*C function*), 950
`fq_poly_divrem_newton_n_preinv` (*C function*), 951
`fq_poly_equal` (*C function*), 943
`fq_poly_equal_fq` (*C function*), 943
`fq_poly_equal_trunc` (*C function*), 943
`fq_poly_evaluate_fq` (*C function*), 955
`fq_poly_factor` (*C function*), 968
`fq_poly_factor_berlekamp` (*C function*), 968
`fq_poly_factor_cantor_zassenhaus` (*C function*), 968
`fq_poly_factor_clear` (*C function*), 966
`fq_poly_factor_concat` (*C function*), 967
`fq_poly_factor_distinct_deg` (*C function*), 967
`fq_poly_factor_equal_deg` (*C function*), 967
`fq_poly_factor_equal_deg_prob` (*C function*), 967
`fq_poly_factor_fit_length` (*C function*), 966
`fq_poly_factor_init` (*C function*), 966
`fq_poly_factor_insert` (*C function*), 966
`fq_poly_factor_kaltofen_shoup` (*C function*), 968
`fq_poly_factor_pow` (*C function*), 967
`fq_poly_factor_print` (*C function*), 966
`fq_poly_factor_print_pretty` (*C function*), 966
`fq_poly_factor_realloc` (*C function*), 966
`fq_poly_factor_set` (*C function*), 966

fq_poly_factor_split_single (*C function*), 967
 fq_poly_factor_squarefree (*C function*), 968
 fq_poly_factor_struct (*C type*), 966
 fq_poly_factor_t (*C type*), 966
 fq_poly_factor_with_berlekamp (*C function*), 968
 fq_poly_factor_with_cantor_zassenhaus (*C function*), 968
 fq_poly_factor_with_kaltofen_shoup (*C function*), 968
 fq_poly_fit_length (*C function*), 941
 fq_poly_fprint (*C function*), 957
 fq_poly_fprint_pretty (*C function*), 957
 fq_poly_gcd (*C function*), 952
 fq_poly_gcd_euclidean_f (*C function*), 953
 fq_poly_gen (*C function*), 942
 fq_poly_get_coeff (*C function*), 943
 fq_poly_get_str (*C function*), 958
 fq_poly_get_str_pretty (*C function*), 958
 fq_poly_hamming_weight (*C function*), 950
 fq_poly_inflate (*C function*), 958
 fq_poly_init (*C function*), 940
 fq_poly_init2 (*C function*), 940
 fq_poly_inv_series (*C function*), 952
 fq_poly_inv_series_newton (*C function*), 952
 fq_poly_invsqrt_series (*C function*), 954
 fq_poly_is_gen (*C function*), 943
 fq_poly_is_irreducible (*C function*), 967
 fq_poly_is_irreducible_ben_or (*C function*), 967
 fq_poly_is_irreducible_ddf (*C function*), 967
 fq_poly_is_one (*C function*), 943
 fq_poly_is_squarefree (*C function*), 967
 fq_poly_is_unit (*C function*), 943
 fq_poly_is_zero (*C function*), 943
 fq_poly_iterated_frobenius_preinv (*C function*), 968
 fq_poly_lead (*C function*), 941
 fq_poly_length (*C function*), 941
 fq_poly_make_monict (*C function*), 942
 fq_poly_mul (*C function*), 945
 fq_poly_mul_classical (*C function*), 944
 fq_poly_mul_KS (*C function*), 945
 fq_poly_mul_reorder (*C function*), 945
 fq_poly_mul_univariate (*C function*), 945
 fq_poly_mulhigh (*C function*), 946
 fq_poly_mulhigh_classical (*C function*), 946
 fq_poly_mullow (*C function*), 946
 fq_poly_mullow_classical (*C function*), 945
 fq_poly_mullow_KS (*C function*), 946
 fq_poly_mullow_univariate (*C function*), 946
 fq_poly_mulmod (*C function*), 947
 fq_poly_mulmod_preinv (*C function*), 947
 fq_poly_neg (*C function*), 944
 fq_poly_one (*C function*), 942
 fq_poly_pow (*C function*), 948
 fq_poly_pow_trunc (*C function*), 950
 fq_poly_pow_trunc_binexp (*C function*), 948
 fq_poly_powmod_fmpz_binexp (*C function*), 948
 fq_poly_powmod_fmpz_binexp_preinv (*C function*), 949
 fq_poly_powmod_fmpz_sliding_preinv (*C function*), 949
 fq_poly_powmod_ui_binexp (*C function*), 948
 fq_poly_powmod_ui_binexp_preinv (*C function*), 948
 fq_poly_powmod_x_fmpz_preinv (*C function*), 949
 fq_poly_precompute_matrix (*C function*), 956
 fq_poly_print (*C function*), 958
 fq_poly_print_pretty (*C function*), 957
 fq_poly_randtest (*C function*), 942
 fq_poly_randtest_irreducible (*C function*), 942
 fq_poly_randtest_monic (*C function*), 942
 fq_poly_randtest_not_zero (*C function*), 942
 fq_poly_realloc (*C function*), 940
 fq_poly_rem (*C function*), 951
 fq_poly_remove (*C function*), 967
 fq_poly_reverse (*C function*), 941
 fq_poly_roots (*C function*), 969
 fq_poly_scalar_addmul_fq (*C function*), 944
 fq_poly_scalar_div_fq (*C function*), 944
 fq_poly_scalar_mul_fq (*C function*), 944
 fq_poly_scalar_submul_fq (*C function*), 944
 fq_poly_set (*C function*), 942
 fq_poly_set_coeff (*C function*), 943
 fq_poly_set_coeff_fmpz (*C function*), 943
 fq_poly_set_fmpz_mod_poly (*C function*), 942
 fq_poly_set_fq (*C function*), 942
 fq_poly_set_nmod_poly (*C function*), 942
 fq_poly_set_trunc (*C function*), 941
 fq_poly_shift_left (*C function*), 950
 fq_poly_shift_right (*C function*), 950
 fq_poly_sqr (*C function*), 948
 fq_poly_sqr_classical (*C function*), 947
 fq_poly_sqr_KS (*C function*), 947
 fq_poly_sqr_reorder (*C function*), 947
 fq_poly_sqrt (*C function*), 954
 fq_poly_sqrt_series (*C function*), 954
 fq_poly_struct (*C type*), 940
 fq_poly_sub (*C function*), 943
 fq_poly_sub_series (*C function*), 943
 fq_poly_swap (*C function*), 942
 fq_poly_t (*C type*), 940
 fq_poly_truncate (*C function*), 941
 fq_poly_xgcd (*C function*), 953
 fq_poly_xgcd_euclidean_f (*C function*), 953
 fq_poly_zero (*C function*), 942
 fq_pow (*C function*), 917
 fq_pow_ui (*C function*), 918
 fq_print (*C function*), 918
 fq_print_pretty (*C function*), 918
 fq_pth_root (*C function*), 918
 fq_rand (*C function*), 919
 fq_rand_not_zero (*C function*), 919

`fq_randtest` (*C function*), 919
`fq_randtest_dense` (*C function*), 919
`fq_randtest_not_zero` (*C function*), 919
`fq_reduce` (*C function*), 917
`fq_set` (*C function*), 919
`fq_set_fmpz` (*C function*), 919
`fq_set_fmpz_mod_mat` (*C function*), 920
`fq_set_fmpz_mod_poly` (*C function*), 919
`fq_set_fmpz_poly` (*C function*), 919
`fq_set_si` (*C function*), 919
`fq_set_ui` (*C function*), 919
`fq_sqr` (*C function*), 917
`fq_sqrt` (*C function*), 918
`fq_struct` (*C type*), 915
`fq_sub` (*C function*), 917
`fq_sub_one` (*C function*), 917
`fq_swap` (*C function*), 919
`fq_t` (*C type*), 915
`fq_trace` (*C function*), 920
`fq_zech_add` (*C function*), 1026
`fq_zech_bit_pack` (*C function*), 1030
`fq_zech_bit_unpack` (*C function*), 1030
`fq_zech_clear` (*C function*), 1025
`fq_zech_ctx_clear` (*C function*), 1025
`fq_zech_ctx_degree` (*C function*), 1025
`fq_zech_ctx_fprint` (*C function*), 1025
`fq_zech_ctx_init_conway_ui` (*C function*), 1024
`fq_zech_ctx_init_fq_nmod_ctx` (*C function*), 1024
`fq_zech_ctx_init_fq_nmod_ctx_check` (*C function*), 1024
`fq_zech_ctx_init_modulus` (*C function*), 1024
`fq_zech_ctx_init_modulus_check` (*C function*), 1024
`fq_zech_ctx_init_random_ui` (*C function*), 1024
`fq_zech_ctx_init_randtest` (*C function*), 1024
`fq_zech_ctx_init_randtest_reducible` (*C function*), 1024
`fq_zech_ctx_init_ui` (*C function*), 1024
`fq_zech_ctx_modulus` (*C function*), 1025
`fq_zech_ctx_order` (*C function*), 1025
`fq_zech_ctx_order_ui` (*C function*), 1025
`fq_zech_ctx_prime` (*C function*), 1025
`fq_zech_ctx_print` (*C function*), 1025
`fq_zech_ctx_struct` (*C type*), 1023
`fq_zech_ctx_t` (*C type*), 1023
`fq_zech_div` (*C function*), 1026
`fq_zech_embed_composition_matrix` (*C function*), 1061
`fq_zech_embed_composition_matrix_sub` (*C function*), 1061
`fq_zech_embed_dual_to_mono_matrix` (*C function*), 1061
`fq_zech_embed_gens` (*C function*), 1060
`fq_zech_embed_matrices` (*C function*), 1060
`fq_zech_embed_mono_to_dual_matrix` (*C function*), 1061
`fq_zech_embed_mul_matrix` (*C function*), 1061
`fq_zech_embed_trace_matrix` (*C function*), 1060
`fq_zech_equal` (*C function*), 1029
`fq_zech_fprint` (*C function*), 1027
`fq_zech_fprint_pretty` (*C function*), 1027
`fq_zech_frobenius` (*C function*), 1029
`fq_zech_gcdinv` (*C function*), 1026
`fq_zech_gen` (*C function*), 1028
`fq_zech_get_fmpz` (*C function*), 1028
`fq_zech_get_fq_nmod` (*C function*), 1028
`fq_zech_get_nmod_mat` (*C function*), 1028
`fq_zech_get_nmod_poly` (*C function*), 1028
`fq_zech_get_str` (*C function*), 1027
`fq_zech_get_str_pretty` (*C function*), 1027
`fq_zech_init` (*C function*), 1025
`fq_zech_init2` (*C function*), 1025
`fq_zech_inv` (*C function*), 1026
`fq_zech_is_invertible` (*C function*), 1029
`fq_zech_is_invertible_f` (*C function*), 1029
`fq_zech_is_one` (*C function*), 1029
`fq_zech_is_primitive` (*C function*), 1029
`fq_zech_is_square` (*C function*), 1027
`fq_zech_is_zero` (*C function*), 1029
`fq_zech_mat_add` (*C function*), 1035
`fq_zech_mat_can_solve` (*C function*), 1036
`fq_zech_mat_charpoly` (*C function*), 1037
`fq_zech_mat_clear` (*C function*), 1032
`fq_zech_mat_concat_horizontal` (*C function*), 1033
`fq_zech_mat_concat_vertical` (*C function*), 1033
`fq_zech_mat_entry` (*C function*), 1032
`fq_zech_mat_entry_set` (*C function*), 1032
`fq_zech_mat_equal` (*C function*), 1034
`fq_zech_mat_fprint` (*C function*), 1033
`fq_zech_mat_fprint_pretty` (*C function*), 1033
`fq_zech_mat_init` (*C function*), 1032
`fq_zech_mat_init_set` (*C function*), 1032
`fq_zech_mat_is_empty` (*C function*), 1034
`fq_zech_mat_is_one` (*C function*), 1034
`fq_zech_mat_is_square` (*C function*), 1034
`fq_zech_mat_is_zero` (*C function*), 1034
`fq_zech_mat_lu` (*C function*), 1036
`fq_zech_mat_minpoly` (*C function*), 1037
`fq_zech_mat_mul` (*C function*), 1035
`fq_zech_mat_mul_classical` (*C function*), 1035
`fq_zech_mat_mul_KS` (*C function*), 1035
`fq_zech_mat_mul_vec` (*C function*), 1035
`fq_zech_mat_mul_vec_ptr` (*C function*), 1035
`fq_zech_mat_ncols` (*C function*), 1032
`fq_zech_mat_neg` (*C function*), 1035
`fq_zech_mat_nrows` (*C function*), 1032
`fq_zech_mat_one` (*C function*), 1032
`fq_zech_mat_print` (*C function*), 1033
`fq_zech_mat_print_pretty` (*C function*), 1033
`fq_zech_mat_randops` (*C function*), 1034
`fq_zech_mat_randpermdiag` (*C function*), 1033
`fq_zech_mat_randrank` (*C function*), 1033
`fq_zech_mat_randtest` (*C function*), 1033

- [fq_zech_mat_randtril \(C function\), 1034](#)
[fq_zech_mat_randtriu \(C function\), 1034](#)
[fq_zech_mat_rref \(C function\), 1036](#)
[fq_zech_mat_set \(C function\), 1032](#)
[fq_zech_mat_set_fmpz_mod_mat \(C function\), 1032](#)
[fq_zech_mat_set_nmod_mat \(C function\), 1032](#)
[fq_zech_mat_similarity \(C function\), 1037](#)
[fq_zech_mat_solve \(C function\), 1036](#)
[fq_zech_mat_solve_tril \(C function\), 1036](#)
[fq_zech_mat_solve_triu \(C function\), 1036](#)
[fq_zech_mat_struct \(C type\), 1031](#)
[fq_zech_mat_sub \(C function\), 1035](#)
[fq_zech_mat_submul \(C function\), 1035](#)
[fq_zech_mat_swap \(C function\), 1032](#)
[fq_zech_mat_swap_entrywise \(C function\), 1032](#)
[fq_zech_mat_t \(C type\), 1031](#)
[fq_zech_mat_transpose \(C function\), 1034](#)
[fq_zech_mat_vec_mul \(C function\), 1035](#)
[fq_zech_mat_vec_mul_ptr \(C function\), 1035](#)
[fq_zech_mat_window_clear \(C function\), 1033](#)
[fq_zech_mat_window_init \(C function\), 1033](#)
[fq_zech_mat_zero \(C function\), 1032](#)
[fq_zech_modulus_derivative_inv \(C function\), 1061](#)
[fq_zech_modulus_pow_series_inv \(C function\), 1061](#)
[fq_zech_mul \(C function\), 1026](#)
[fq_zech_mul_fmpz \(C function\), 1026](#)
[fq_zech_mul_si \(C function\), 1026](#)
[fq_zech_mul_ui \(C function\), 1026](#)
[fq_zech_multiplicative_order \(C function\), 1029](#)
[fq_zech_neg \(C function\), 1026](#)
[fq_zech_norm \(C function\), 1029](#)
[fq_zech_one \(C function\), 1028](#)
[fq_zech_poly_add \(C function\), 1041](#)
[fq_zech_poly_add_series \(C function\), 1041](#)
[fq_zech_poly_add_si \(C function\), 1041](#)
[fq_zech_poly_clear \(C function\), 1038](#)
[fq_zech_poly_compose \(C function\), 1053](#)
[fq_zech_poly_compose_mod \(C function\), 1054](#)
[fq_zech_poly_compose_mod_brent_kung \(C function\), 1053](#)
[fq_zech_poly_compose_mod_brent_kung_precomp \(C function\), 1055](#)
[fq_zech_poly_compose_mod_brent_kung_preinv \(C function\), 1054](#)
[fq_zech_poly_compose_mod_horner \(C function\), 1053](#)
[fq_zech_poly_compose_mod_horner_preinv \(C function\), 1053](#)
[fq_zech_poly_compose_mod_preinv \(C function\), 1054](#)
[fq_zech_poly_deflate \(C function\), 1056](#)
[fq_zech_poly_deflation \(C function\), 1056](#)
[fq_zech_poly_degree \(C function\), 1038](#)
[fq_zech_poly_derivative \(C function\), 1052](#)
[fq_zech_poly_div \(C function\), 1048](#)
[fq_zech_poly_div_newton_n_preinv \(C function\), 1049](#)
[fq_zech_poly_div_series \(C function\), 1050](#)
[fq_zech_poly_divides \(C function\), 1051](#)
[fq_zech_poly_divrem \(C function\), 1048](#)
[fq_zech_poly_divrem_f \(C function\), 1048](#)
[fq_zech_poly_divrem_newton_n_preinv \(C function\), 1049](#)
[fq_zech_poly_equal \(C function\), 1040](#)
[fq_zech_poly_equal_fq_zech \(C function\), 1040](#)
[fq_zech_poly_equal_trunc \(C function\), 1040](#)
[fq_zech_poly_evaluate_fq_zech \(C function\), 1052](#)
[fq_zech_poly_factor \(C function\), 1058](#)
[fq_zech_poly_factor_berlekamp \(C function\), 1059](#)
[fq_zech_poly_factor_cantor_zassenhaus \(C function\), 1059](#)
[fq_zech_poly_factor_clear \(C function\), 1057](#)
[fq_zech_poly_factor_concat \(C function\), 1057](#)
[fq_zech_poly_factor_distinct_deg \(C function\), 1058](#)
[fq_zech_poly_factor_equal_deg \(C function\), 1058](#)
[fq_zech_poly_factor_equal_deg_prob \(C function\), 1058](#)
[fq_zech_poly_factor_fit_length \(C function\), 1057](#)
[fq_zech_poly_factor_init \(C function\), 1057](#)
[fq_zech_poly_factor_insert \(C function\), 1057](#)
[fq_zech_poly_factor_kaltofen_shoup \(C function\), 1059](#)
[fq_zech_poly_factor_pow \(C function\), 1057](#)
[fq_zech_poly_factor_print \(C function\), 1057](#)
[fq_zech_poly_factor_print_pretty \(C function\), 1057](#)
[fq_zech_poly_factor_realloc \(C function\), 1057](#)
[fq_zech_poly_factor_set \(C function\), 1057](#)
[fq_zech_poly_factor_split_single \(C function\), 1058](#)
[fq_zech_poly_factor_squarefree \(C function\), 1058](#)
[fq_zech_poly_factor_struct \(C type\), 1057](#)
[fq_zech_poly_factor_t \(C type\), 1057](#)
[fq_zech_poly_factor_with_berlekamp \(C function\), 1059](#)
[fq_zech_poly_factor_with_cantor_zassenhaus \(C function\), 1059](#)
[fq_zech_poly_factor_with_kaltofen_shoup \(C function\), 1059](#)
[fq_zech_poly_fit_length \(C function\), 1038](#)
[fq_zech_poly_fprint \(C function\), 1056](#)
[fq_zech_poly_fprint_pretty \(C function\), 1055](#)
[fq_zech_poly_gcd \(C function\), 1050](#)
[fq_zech_poly_gcd_euclidean_f \(C function\), 1050](#)

fq_zech_poly_gen (*C function*), 1039
 fq_zech_poly_get_coeff (*C function*), 1040
 fq_zech_poly_get_str (*C function*), 1056
 fq_zech_poly_get_str_pretty (*C function*), 1056
 fq_zech_poly_hamming_weight (*C function*), 1048
 fq_zech_poly_inflate (*C function*), 1056
 fq_zech_poly_init (*C function*), 1037
 fq_zech_poly_init2 (*C function*), 1037
 fq_zech_poly_inv_series (*C function*), 1049
 fq_zech_poly_inv_series_newton (*C function*), 1049
 fq_zech_poly_invsqrt_series (*C function*), 1052
 fq_zech_poly_is_gen (*C function*), 1040
 fq_zech_poly_is_irreducible (*C function*), 1058
 fq_zech_poly_is_irreducible_ben_or (*C function*), 1058
 fq_zech_poly_is_irreducible_ddf (*C function*), 1058
 fq_zech_poly_is_one (*C function*), 1040
 fq_zech_poly_is_squarefree (*C function*), 1058
 fq_zech_poly_is_unit (*C function*), 1040
 fq_zech_poly_is_zero (*C function*), 1040
 fq_zech_poly_iterated_frobenius_preinv (*C function*), 1059
 fq_zech_poly_lead (*C function*), 1038
 fq_zech_poly_length (*C function*), 1038
 fq_zech_poly_make_monic (*C function*), 1039
 fq_zech_poly_mul (*C function*), 1043
 fq_zech_poly_mul_classical (*C function*), 1042
 fq_zech_poly_mul_KS (*C function*), 1043
 fq_zech_poly_mul_reorder (*C function*), 1042
 fq_zech_poly_mulhigh (*C function*), 1044
 fq_zech_poly_mulhigh_classical (*C function*), 1044
 fq_zech_poly_mullo (*C function*), 1043
 fq_zech_poly_mullo_classical (*C function*), 1043
 fq_zech_poly_mullo_KS (*C function*), 1043
 fq_zech_poly_mulmod (*C function*), 1044
 fq_zech_poly_mulmod_preinv (*C function*), 1044
 fq_zech_poly_neg (*C function*), 1041
 fq_zech_poly_one (*C function*), 1039
 fq_zech_poly_pow (*C function*), 1045
 fq_zech_poly_pow_trunc (*C function*), 1047
 fq_zech_poly_pow_trunc_binexp (*C function*), 1047
 fq_zech_poly_powmod_fmpz_binexp (*C function*), 1046
 fq_zech_poly_powmod_fmpz_binexp_preinv (*C function*), 1046
 fq_zech_poly_powmod_fmpz_sliding_preinv (*C function*), 1046
 fq_zech_poly_powmod_ui_binexp (*C function*), 1045
 fq_zech_poly_powmod_ui_binexp_preinv (*C function*), 1045
 fq_zech_poly_precompute_matrix (*C function*), 1055
 fq_zech_poly_print (*C function*), 1056
 fq_zech_poly_print_pretty (*C function*), 1055
 fq_zech_poly_randtest (*C function*), 1039
 fq_zech_poly_randtest_irreducible (*C function*), 1039
 fq_zech_poly_randtest_monic (*C function*), 1039
 fq_zech_poly_randtest_not_zero (*C function*), 1039
 fq_zech_poly_realloc (*C function*), 1037
 fq_zech_poly_rem (*C function*), 1048
 fq_zech_poly_remove (*C function*), 1057
 fq_zech_poly_reverse (*C function*), 1038
 fq_zech_poly_roots (*C function*), 1060
 fq_zech_poly_scalar_addmul_fq_zech (*C function*), 1041
 fq_zech_poly_scalar_div_fq_zech (*C function*), 1042
 fq_zech_poly_scalar_mul_fq_zech (*C function*), 1041
 fq_zech_poly_scalar_submul_fq_zech (*C function*), 1042
 fq_zech_poly_set (*C function*), 1039
 fq_zech_poly_set_coeff (*C function*), 1040
 fq_zech_poly_set_coeff_fmpz (*C function*), 1040
 fq_zech_poly_set_fmpz_mod_poly (*C function*), 1039
 fq_zech_poly_set_fq_zech (*C function*), 1039
 fq_zech_poly_set_nmod_poly (*C function*), 1039
 fq_zech_poly_set_trunc (*C function*), 1038
 fq_zech_poly_shift_left (*C function*), 1047
 fq_zech_poly_shift_right (*C function*), 1047
 fq_zech_poly_sqr (*C function*), 1045
 fq_zech_poly_sqr_classical (*C function*), 1044
 fq_zech_poly_sqr_KS (*C function*), 1045
 fq_zech_poly_sqrt (*C function*), 1052
 fq_zech_poly_sqrt_series (*C function*), 1052
 fq_zech_poly_struct (*C type*), 1037
 fq_zech_poly_sub (*C function*), 1041
 fq_zech_poly_sub_series (*C function*), 1041
 fq_zech_poly_swap (*C function*), 1039
 fq_zech_poly_t (*C type*), 1037
 fq_zech_poly_truncate (*C function*), 1038
 fq_zech_poly_xgcd (*C function*), 1050
 fq_zech_poly_xgcd_euclidean_f (*C function*), 1051
 fq_zech_poly_zero (*C function*), 1039
 fq_zech_pow (*C function*), 1026
 fq_zech_pow_ui (*C function*), 1026
 fq_zech_print (*C function*), 1027
 fq_zech_print_pretty (*C function*), 1027

fq_zech_pth_root (*C function*), 1027
 fq_zech_rand (*C function*), 1027
 fq_zech_rand_not_zero (*C function*), 1028
 fq_zech_randtest (*C function*), 1027
 fq_zech_randtest_dense (*C function*), 1027
 fq_zech_randtest_not_zero (*C function*), 1027
 fq_zech_reduce (*C function*), 1025
 fq_zech_set (*C function*), 1028
 fq_zech_set_fmpz (*C function*), 1028
 fq_zech_set_fq_nmod (*C function*), 1028
 fq_zech_set_nmod_mat (*C function*), 1028
 fq_zech_set_nmod_poly (*C function*), 1028
 fq_zech_set_si (*C function*), 1028
 fq_zech_set_ui (*C function*), 1028
 fq_zech_sqr (*C function*), 1026
 fq_zech_sqrt (*C function*), 1027
 fq_zech_struct (*C type*), 1023
 fq_zech_sub (*C function*), 1026
 fq_zech_sub_one (*C function*), 1026
 fq_zech_swap (*C function*), 1028
 fq_zech_t (*C type*), 1023
 fq_zech_trace (*C function*), 1029
 fq_zech_zero (*C function*), 1028
 fq_zero (*C function*), 919
 FresnelC (*C macro*), 910
 FresnelS (*C macro*), 910
 Fun (*C macro*), 897

G

Gamma (*C macro*), 909
 GaussLegendreWeight (*C macro*), 909
 GaussSum (*C macro*), 912
 GCD (*C macro*), 907
 GegenbauerC (*C macro*), 909
 GeneralizedBernoulliB (*C macro*), 912
 GeneralizedRiemannHypothesis (*C macro*), 912
 GeneralLinearGroup (*C macro*), 905
 get_clock (*C function*), 23
 get_default_mpn_ctx (*C function*), 306
 GlaisherConstant (*C macro*), 900
 GoldenRatio (*C macro*), 899
 gr_abs (*C function*), 48
 gr_abs_ge (*C function*), 49
 gr_abs_gt (*C function*), 49
 gr_abs_le (*C function*), 49
 gr_abs_lt (*C function*), 49
 gr_acos (*C function*), 67
 gr_acos_pi (*C function*), 67
 gr_acosh (*C function*), 67
 gr_acot (*C function*), 67
 gr_acot_pi (*C function*), 67
 gr_acoth (*C function*), 67
 gr_acsc (*C function*), 67
 gr_acsc_pi (*C function*), 67
 gr_acsch (*C function*), 67
 gr_add (*C function*), 44
 gr_add_fmpz (*C function*), 44
 gr_add_fmpz (*C function*), 44
 gr_add_other (*C function*), 44
 gr_add_si (*C function*), 44
 gr_add_ui (*C function*), 44
 gr_addmul (*C function*), 44
 gr_addmul_fmpz (*C function*), 44
 gr_addmul_fmpz (*C function*), 44
 gr_addmul_other (*C function*), 44
 gr_addmul_si (*C function*), 44
 gr_addmul_ui (*C function*), 44
 gr_agm (*C function*), 71
 gr_agm1 (*C function*), 71
 gr_airy (*C function*), 70
 gr_airy_ai (*C function*), 70
 gr_airy_ai_prime (*C function*), 70
 gr_airy_ai_prime_zero (*C function*), 70
 gr_airy_ai_zero (*C function*), 70
 gr_airy_bi (*C function*), 70
 gr_airy_bi_prime (*C function*), 70
 gr_airy_bi_prime_zero (*C function*), 70
 gr_airy_bi_zero (*C function*), 70
 gr_arg (*C function*), 48
 gr_asec (*C function*), 67
 gr_asec_pi (*C function*), 67
 gr_asech (*C function*), 67
 gr_asin (*C function*), 67
 gr_asin_pi (*C function*), 67
 gr_asinh (*C function*), 67
 gr_atan (*C function*), 67
 gr_atan2 (*C function*), 67
 gr_atan_pi (*C function*), 67
 gr_atanh (*C function*), 67
 gr_barnes_g (*C function*), 68
 gr_bellnum_fmpz (*C function*), 69
 gr_bellnum_ui (*C function*), 69
 gr_bellnum_vec (*C function*), 69
 gr_bernoulli_fmpz (*C function*), 68
 gr_bernoulli_ui (*C function*), 68
 gr_bernoulli_vec (*C function*), 68
 gr_bessel_i (*C function*), 70
 gr_bessel_i_scaled (*C function*), 70
 gr_bessel_j (*C function*), 70
 gr_bessel_j_y (*C function*), 70
 gr_bessel_k (*C function*), 70
 gr_bessel_k_scaled (*C function*), 70
 gr_bessel_y (*C function*), 70
 gr_beta (*C function*), 68
 gr_beta_lower (*C function*), 69
 gr_bin (*C function*), 67
 gr_bin_ui (*C function*), 67
 gr_bin_ui_vec (*C function*), 67
 gr_bin_uiui (*C function*), 67
 gr_bin_vec (*C function*), 67
 gr_canonical_associate (*C function*), 47
 gr_carlson_rc (*C function*), 71
 gr_carlson_rd (*C function*), 71
 gr_carlson_rf (*C function*), 71
 gr_carlson_rg (*C function*), 71
 gr_carlson_rj (*C function*), 71

gr_catalan (*C function*), 66
 gr_ceil (*C function*), 48
 gr_chebyshev_t (*C function*), 69
 gr_chebyshev_t_fmpz (*C function*), 69
 gr_chebyshev_u (*C function*), 69
 gr_chebyshev_u_fmpz (*C function*), 69
 gr_check (*C function*), 40
 gr_clear (*C function*), 41
 gr_cmp (*C function*), 48
 gr_cmp_other (*C function*), 48
 gr_cmpabs (*C function*), 48
 gr_cmpabs_other (*C function*), 48
 gr_conj (*C function*), 48
 gr_cos (*C function*), 66
 gr_cos_integral (*C function*), 69
 gr_cos_pi (*C function*), 66
 gr_cosh (*C function*), 66
 gr_cosh_integral (*C function*), 69
 gr_cot (*C function*), 66
 gr_cot_pi (*C function*), 66
 gr_coth (*C function*), 66
 gr_coulomb (*C function*), 70
 gr_coulomb_f (*C function*), 70
 gr_coulomb_g (*C function*), 70
 gr_coulomb_hneg (*C function*), 70
 gr_coulomb_hpos (*C function*), 70
 gr_csc (*C function*), 66
 gr_csc_pi (*C function*), 66
 gr_csch (*C function*), 66
 gr_csgn (*C function*), 48
 gr_ctx_arb_get_prec (*C function*), 55
 gr_ctx_arb_set_prec (*C function*), 55
 gr_ctx_ca_get_option (*C function*), 56
 gr_ctx_ca_set_option (*C function*), 56
 gr_ctx_clear (*C function*), 40
 gr_ctx_cmp_coercion (*C function*), 53
 gr_ctx_fq_degree (*C function*), 49
 gr_ctx_fq_order (*C function*), 49
 gr_ctx_fq_prime (*C function*), 49
 gr_ctx_gen_name (*C function*), 40
 gr_ctx_get_real_prec (*C function*), 53
 gr_ctx_get_str (*C function*), 40
 gr_ctx_has_real_prec (*C function*), 53
 gr_ctx_init_complex_acb (*C function*), 55
 gr_ctx_init_complex_algebraic_ca (*C function*), 56
 gr_ctx_init_complex_ca (*C function*), 56
 gr_ctx_init_complex_extended_ca (*C function*), 56
 gr_ctx_init_complex_float_acf (*C function*), 56
 gr_ctx_init_complex_qqbar (*C function*), 55
 gr_ctx_init_debug (*C function*), 54
 gr_ctx_init_dirichlet_group (*C function*), 54
 gr_ctx_init_fexpr (*C function*), 58
 gr_ctx_init_fmpz (*C function*), 54
 gr_ctx_init_fmpz_poly (*C function*), 57
 gr_ctx_init_fmpz (*C function*), 54
 gr_ctx_init_fmpz_mod (*C function*), 55
 gr_ctx_init_fmpz_mod_mpoly_q (*C function*), 58
 gr_ctx_init_fmpz_mpoly (*C function*), 57
 gr_ctx_init_fmpz_poly_q (*C function*), 58
 gr_ctx_init_fmpz_poly (*C function*), 57
 gr_ctx_init_fmpzpi (*C function*), 54
 gr_ctx_init_fq (*C function*), 55
 gr_ctx_init_fq_nmod (*C function*), 55
 gr_ctx_init_fq_zech (*C function*), 55
 gr_ctx_init_gr_complex (*C function*), 56
 gr_ctx_init_gr_fraction (*C function*), 58
 gr_ctx_init_gr_fraction.GR_FRACTION_NO_REDUCTION
 (*C macro*), 58
 gr_ctx_init_gr_fraction.GR_FRACTION_STRONGLY_CANONICAL
 (*C macro*), 58
 gr_ctx_init_gr_mpoly (*C function*), 57
 gr_ctx_init_gr_poly (*C function*), 57
 gr_ctx_init_matrix_domain (*C function*), 57
 gr_ctx_init_matrix_ring (*C function*), 57
 gr_ctx_init_matrix_space (*C function*), 57
 gr_ctx_init_mpn_mod (*C function*), 455
 gr_ctx_init_mpn_mod_randtest (*C function*),
 455
 gr_ctx_init_nf (*C function*), 55
 gr_ctx_init_nf_fmpz_poly (*C function*), 55
 gr_ctx_init_nmod (*C function*), 381
 gr_ctx_init_nmod32 (*C function*), 381
 gr_ctx_init_nmod8 (*C function*), 381
 gr_ctx_init_nmod_redc (*C function*), 381
 gr_ctx_init_nmod_redc_fast (*C function*), 381
 gr_ctx_init_perm (*C function*), 54
 gr_ctx_init_psl2z (*C function*), 54
 gr_ctx_init_radix_integer (*C function*), 313
 gr_ctx_init_random (*C function*), 54
 gr_ctx_init_random_commutative_ring (*C function*), 54
 gr_ctx_init_random_field (*C function*), 54
 gr_ctx_init_real_algebraic_ca (*C function*),
 56
 gr_ctx_init_real_arb (*C function*), 55
 gr_ctx_init_real_ca (*C function*), 56
 gr_ctx_init_real_float_arf (*C function*), 56
 gr_ctx_init_real_qqbar (*C function*), 55
 gr_ctx_init_vector_gr_vec (*C function*), 57
 gr_ctx_init_vector_space_gr_vec (*C function*), 57
 gr_ctx_is_algebraically_closed (*C function*),
 53
 gr_ctx_is_approx_commutative_ring (*C function*), 53
 gr_ctx_is_canonical (*C function*), 53
 gr_ctx_is_commutative_ring (*C function*), 53
 gr_ctx_is_complex_vector_space (*C function*),
 53
 gr_ctx_is_exact (*C function*), 53
 gr_ctx_is_field (*C function*), 53
 gr_ctx_is_finite (*C function*), 53

`gr_ctx_is_finite_characteristic` (*C function*), 53
`gr_ctx_is_integral_domain` (*C function*), 53
`gr_ctx_is_multiplicative_group` (*C function*), 53
`gr_ctx_is_ordered_ring` (*C function*), 53
`gr_ctx_is_rational_vector_space` (*C function*), 53
`gr_ctx_is_real_vector_space` (*C function*), 53
`gr_ctx_is_ring` (*C function*), 53
`gr_ctx_is_unique_factorization_domain` (*C function*), 53
`gr_ctx_is_zero_ring` (*C function*), 53
`gr_ctx_ngens` (*C function*), 40
`gr_ctx_print` (*C function*), 40
`gr_ctx_println` (*C function*), 40
`gr_ctx_ptr` (*C type*), 38
`gr_ctx_set_gen_name` (*C function*), 40
`gr_ctx_set_gen_names` (*C function*), 40
`gr_ctx_set_is_field` (*C function*), 55
`gr_ctx_set_real_prec` (*C function*), 53
`gr_ctx_sizeof_elem` (*C function*), 40
`gr_ctx_struct` (*C type*), 38
`gr_ctx_t` (*C type*), 38
`gr_ctx_uninitialized` (*C function*), 38
`gr_ctx_write` (*C function*), 40
`gr_dedekind_eta` (*C function*), 72
`gr_dedekind_eta_q` (*C function*), 72
`gr_denominator` (*C function*), 48
`gr_derivative_gen` (*C function*), 47
`gr_digamma` (*C function*), 68
`gr_dilog` (*C function*), 69
`gr_dirichlet_chi_fmpz` (*C function*), 71
`gr_dirichlet_chi_vec` (*C function*), 71
`gr_dirichlet_eta` (*C function*), 71
`gr_dirichlet_hardy_theta` (*C function*), 71
`gr_dirichlet_hardy_z` (*C function*), 71
`gr_dirichlet_l` (*C function*), 71
`gr_dirichlet_l_all` (*C function*), 71
`gr_div` (*C function*), 45
`gr_div_fmpq` (*C function*), 45
`gr_div_fmpz` (*C function*), 45
`gr_div_nonunique` (*C function*), 46
`gr_div_other` (*C function*), 45
`gr_div_si` (*C function*), 45
`gr_div_ui` (*C function*), 45
`gr_divexact` (*C function*), 46
`gr_divexact_fmpz` (*C function*), 46
`gr_divexact_other` (*C function*), 46
`gr_divexact_si` (*C function*), 46
`gr_divexact_ui` (*C function*), 46
`gr_divides` (*C function*), 46
`GR_DOMAIN` (*C macro*), 39
`gr_doublefac` (*C function*), 68
`gr_doublefac_ui` (*C function*), 68
`gr_eisenstein_e` (*C function*), 72
`gr_eisenstein_g` (*C function*), 72
`gr_eisenstein_g_vec` (*C function*), 72
`gr_elliptic_e` (*C function*), 71
`gr_elliptic_e_inc` (*C function*), 71
`gr_elliptic_f` (*C function*), 71
`gr_elliptic_invariants` (*C function*), 72
`gr_elliptic_k` (*C function*), 71
`gr_elliptic_pi` (*C function*), 71
`gr_elliptic_pi_inc` (*C function*), 71
`gr_elliptic_roots` (*C function*), 72
`GR_ENTRY` (*C macro*), 73
`gr_equal` (*C function*), 44
`gr_erf` (*C function*), 69
`gr_erfc` (*C function*), 69
`gr_erfcinv` (*C function*), 69
`gr_erfcx` (*C function*), 69
`gr_erfi` (*C function*), 69
`gr_erfinv` (*C function*), 69
`gr_euclidean_div` (*C function*), 46
`gr_euclidean_divrem` (*C function*), 46
`gr_euclidean_rem` (*C function*), 46
`gr_euler` (*C function*), 66
`gr_eulernum_fmpz` (*C function*), 68
`gr_eulernum_ui` (*C function*), 68
`gr_eulernum_vec` (*C function*), 68
`gr_exp` (*C function*), 66
`gr_exp10` (*C function*), 66
`gr_exp2` (*C function*), 66
`gr_exp_integral` (*C function*), 69
`gr_exp_integral_ei` (*C function*), 69
`gr_exp_pi_i` (*C function*), 66
`gr_expm1` (*C function*), 66
`gr_fac` (*C function*), 67
`gr_fac_fmpz` (*C function*), 67
`gr_fac_ui` (*C function*), 67
`gr_fac_vec` (*C function*), 67
`gr_factor` (*C function*), 47
`gr_falling` (*C function*), 67
`gr_falling_ui` (*C function*), 67
`gr_fib_fmpz` (*C function*), 68
`gr_fib_ui` (*C function*), 68
`gr_fib_vec` (*C function*), 68
`gr_floor` (*C function*), 48
`gr_fmpz_mpoly_evaluate` (*C function*), 61
`gr_fmpz_mpoly_evaluate_horner` (*C function*), 61
`gr_fmpz_mpoly_evaluate_iter` (*C function*), 61
`gr_fmpz_poly_evaluate` (*C function*), 61
`gr_fmpz_poly_evaluate_horner` (*C function*), 61
`gr_fmpz_poly_evaluate_rectangular` (*C function*), 61
`gr_fq_frobenius` (*C function*), 49
`gr_fq_is_primitive` (*C function*), 49
`gr_fq_multiplicative_order` (*C function*), 49
`gr_fq_norm` (*C function*), 49
`gr_fq_pth_root` (*C function*), 49
`gr_fq_trace` (*C function*), 49
`gr_fresnel` (*C function*), 69
`gr_fresnel_c` (*C function*), 69
`gr_fresnel_s` (*C function*), 69

gr_funcptr (*C type*), 51
 gr_gamma (*C function*), 68
 gr_gamma_fmpq (*C function*), 68
 gr_gamma_fmpz (*C function*), 68
 gr_gamma_lower (*C function*), 69
 gr_gamma_upper (*C function*), 69
 gr_gcd (*C function*), 47
 gr_ge (*C function*), 49
 gr_gegenbauer_c (*C function*), 69
 gr_gen (*C function*), 43
 gr_generic_add (*C function*), 59
 gr_generic_add_fmpq (*C function*), 60
 gr_generic_add_fmpz (*C function*), 60
 gr_generic_add_other (*C function*), 60
 gr_generic_add_si (*C function*), 60
 gr_generic_add_ui (*C function*), 60
 gr_generic_addmul (*C function*), 60
 gr_generic_addmul_fmpq (*C function*), 60
 gr_generic_addmul_fmpz (*C function*), 60
 gr_generic_addmul_other (*C function*), 60
 gr_generic_addmul_si (*C function*), 60
 gr_generic_addmul_ui (*C function*), 60
 gr_generic_bernoulli_fmpz (*C function*), 62
 gr_generic_bernoulli_ui (*C function*), 62
 gr_generic_bernoulli_vec (*C function*), 62
 gr_generic_clear (*C function*), 59
 gr_generic_cmp (*C function*), 62
 gr_generic_cmp_other (*C function*), 62
 gr_generic_cmpabs (*C function*), 62
 gr_generic_cmpabs_other (*C function*), 62
 gr_generic_ctx_clear (*C function*), 59
 gr_generic_ctx_predicate (*C function*), 51
 gr_generic_ctx_predicate_false (*C function*), 51
 gr_generic_ctx_predicate_true (*C function*), 51
 gr_generic_denominator (*C function*), 61
 gr_generic_div_fmpq (*C function*), 61
 gr_generic_div_fmpz (*C function*), 61
 gr_generic_div_other (*C function*), 61
 gr_generic_div_si (*C function*), 61
 gr_generic_div_ui (*C function*), 61
 gr_generic_divexact (*C function*), 61
 gr_generic_equal (*C function*), 59
 gr_generic_eulernum_fmpz (*C function*), 62
 gr_generic_eulernum_ui (*C function*), 62
 gr_generic_eulernum_vec (*C function*), 62
 gr_generic_get_fmpz_2exp_fmpz (*C function*), 60
 gr_generic_init (*C function*), 59
 gr_generic_inv (*C function*), 61
 gr_generic_is_invertible (*C function*), 61
 gr_generic_is_neg_one (*C function*), 59
 gr_generic_is_one (*C function*), 59
 gr_generic_is_square (*C function*), 61
 gr_generic_is_zero (*C function*), 59
 gr_generic_mul (*C function*), 59
 gr_generic_mul_2exp_fmpz (*C function*), 60
 gr_generic_mul_2exp_si (*C function*), 60
 gr_generic_mul_fmpq (*C function*), 60
 gr_generic_mul_fmpz (*C function*), 60
 gr_generic_mul_other (*C function*), 60
 gr_generic_mul_si (*C function*), 60
 gr_generic_mul_two (*C function*), 60
 gr_generic_mul_ui (*C function*), 60
 gr_generic_neg (*C function*), 59
 gr_generic_neg_one (*C function*), 59
 gr_generic_numerator (*C function*), 61
 gr_generic_one (*C function*), 59
 gr_generic_other_add (*C function*), 60
 gr_generic_other_add_vec (*C function*), 63
 gr_generic_other_div (*C function*), 61
 gr_generic_other_div_vec (*C function*), 63
 gr_generic_other_divexact_vec (*C function*), 63
 gr_generic_other_mul (*C function*), 60
 gr_generic_other_mul_vec (*C function*), 63
 gr_generic_other_pow (*C function*), 61
 gr_generic_other_pow_vec (*C function*), 63
 gr_generic_other_sub (*C function*), 60
 gr_generic_other_sub_vec (*C function*), 63
 gr_generic_poly_factor_roots (*C function*), 65
 gr_generic_pow_fmpq (*C function*), 61
 gr_generic_pow_fmpz (*C function*), 61
 gr_generic_pow_fmpz_binexp (*C function*), 61
 gr_generic_pow_fmpz_sliding (*C function*), 61
 gr_generic_pow_other (*C function*), 61
 gr_generic_pow_si (*C function*), 61
 gr_generic_pow_ui (*C function*), 61
 gr_generic_pow_ui_binexp (*C function*), 61
 gr_generic_pow_ui_sliding (*C function*), 61
 gr_generic_randtest (*C function*), 59
 gr_generic_randtest_not_zero (*C function*), 59
 gr_generic_randtest_small (*C function*), 59
 gr_generic_rsqr (C function), 61
 gr_generic_scalar_add_vec (*C function*), 63
 gr_generic_scalar_div_vec (*C function*), 63
 gr_generic_scalar_divexact_vec (*C function*), 63
 gr_generic_scalar_mul_vec (*C function*), 63
 gr_generic_scalar_other_add_vec (*C function*), 63
 gr_generic_scalar_other_div_vec (*C function*), 63
 gr_generic_scalar_other_divexact_vec (*C function*), 63
 gr_generic_scalar_other_mul_vec (*C function*), 63
 gr_generic_scalar_other_pow_vec (*C function*), 63
 gr_generic_scalar_other_sub_vec (*C function*), 63
 gr_generic_scalar_pow_vec (*C function*), 63
 gr_generic_scalar_sub_vec (*C function*), 63
 gr_generic_set (*C function*), 59
 gr_generic_set_fmpq (*C function*), 59

[gr_generic_set_fmpz \(C function\), 59](#)
[gr_generic_set_fmpz_2exp_fmpz \(C function\), 60](#)
[gr_generic_set_other \(C function\), 59](#)
[gr_generic_set_shallow \(C function\), 59](#)
[gr_generic_set_si \(C function\), 59](#)
[gr_generic_set_str \(C function\), 59](#)
[gr_generic_set_str_balance_additions \(C function\), 59](#)
[gr_generic_set_str_expr \(C function\), 59](#)
[gr_generic_set_str_ring_exponents \(C function\), 59](#)
[gr_generic_set_ui \(C function\), 59](#)
[gr_generic_sqr \(C function\), 60](#)
[gr_generic_sqrt \(C function\), 61](#)
[gr_generic_stirling_s1_ui_vec \(C function\), 62](#)
[gr_generic_stirling_s1_uiui \(C function\), 62](#)
[gr_generic_stirling_s1u_ui_vec \(C function\), 62](#)
[gr_generic_stirling_s1u_uiui \(C function\), 62](#)
[gr_generic_stirling_s2_ui_vec \(C function\), 62](#)
[gr_generic_stirling_s2_uiui \(C function\), 62](#)
[gr_generic_sub \(C function\), 59](#)
[gr_generic_sub_fmpz \(C function\), 60](#)
[gr_generic_sub_fmpz \(C function\), 60](#)
[gr_generic_sub_other \(C function\), 60](#)
[gr_generic_sub_si \(C function\), 60](#)
[gr_generic_sub_ui \(C function\), 60](#)
[gr_generic_submul \(C function\), 60](#)
[gr_generic_submul_fmpz \(C function\), 60](#)
[gr_generic_submul_fmpz \(C function\), 60](#)
[gr_generic_submul_other \(C function\), 60](#)
[gr_generic_submul_si \(C function\), 60](#)
[gr_generic_submul_ui \(C function\), 60](#)
[gr_generic_swap \(C function\), 59](#)
[gr_generic_vec_add \(C function\), 63](#)
[gr_generic_vec_add_other \(C function\), 63](#)
[gr_generic_vec_add_scalar \(C function\), 63](#)
[gr_generic_vec_add_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_add_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_add_scalar_other \(C function\), 63](#)
[gr_generic_vec_add_scalar_si \(C function\), 63](#)
[gr_generic_vec_add_scalar_ui \(C function\), 63](#)
[gr_generic_vec_clear \(C function\), 62](#)
[gr_generic_vec_div \(C function\), 63](#)
[gr_generic_vec_div_other \(C function\), 63](#)
[gr_generic_vec_div_scalar \(C function\), 63](#)
[gr_generic_vec_div_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_div_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_div_scalar_other \(C function\), 63](#)
[gr_generic_vec_div_scalar_si \(C function\), 63](#)
[gr_generic_vec_div_scalar_ui \(C function\), 63](#)
[gr_generic_vec_divexact \(C function\), 63](#)
[gr_generic_vec_divexact_other \(C function\), 63](#)
[gr_generic_vec_divexact_scalar \(C function\), 63](#)
[gr_generic_vec_divexact_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_divexact_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_divexact_scalar_other \(C function\), 63](#)
[gr_generic_vec_divexact_scalar_si \(C function\), 63](#)
[gr_generic_vec_divexact_scalar_ui \(C function\), 63](#)
[gr_generic_vec_dot \(C function\), 63](#)
[gr_generic_vec_dot_fmpz \(C function\), 63](#)
[gr_generic_vec_dot_rev \(C function\), 63](#)
[gr_generic_vec_dot_si \(C function\), 63](#)
[gr_generic_vec_dot_ui \(C function\), 63](#)
[gr_generic_vec_equal \(C function\), 62](#)
[gr_generic_vec_init \(C function\), 62](#)
[gr_generic_vec_is_zero \(C function\), 63](#)
[gr_generic_vec_mul \(C function\), 63](#)
[gr_generic_vec_mul_other \(C function\), 63](#)
[gr_generic_vec_mul_scalar \(C function\), 63](#)
[gr_generic_vec_mul_scalar_2exp_si \(C function\), 62](#)
[gr_generic_vec_mul_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_mul_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_mul_scalar_other \(C function\), 63](#)
[gr_generic_vec_mul_scalar_si \(C function\), 63](#)
[gr_generic_vec_mul_scalar_ui \(C function\), 63](#)
[gr_generic_vec_neg \(C function\), 62](#)
[gr_generic_vec_normalise \(C function\), 62](#)
[gr_generic_vec_normalise_weak \(C function\), 62](#)
[gr_generic_vec_pow \(C function\), 63](#)
[gr_generic_vec_pow_other \(C function\), 63](#)
[gr_generic_vec_pow_scalar \(C function\), 63](#)
[gr_generic_vec_pow_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_pow_scalar_fmpz \(C function\), 63](#)
[gr_generic_vec_pow_scalar_other \(C function\), 63](#)
[gr_generic_vec_pow_scalar_si \(C function\), 63](#)
[gr_generic_vec_pow_scalar_ui \(C function\), 63](#)
[gr_generic_vec_reciprocals \(C function\), 63](#)
[gr_generic_vec_scalar_addmul \(C function\), 62](#)
[gr_generic_vec_scalar_addmul_si \(C function\), 62](#)
[gr_generic_vec_scalar_submul \(C function\), 62](#)

`gr_generic_vec_scalar_submul_si` (*C function*), 62
`gr_generic_vec_set` (*C function*), 62
`gr_generic_vec_set_powers` (*C function*), 63
`gr_generic_vec_sub` (*C function*), 63
`gr_generic_vec_sub_other` (*C function*), 63
`gr_generic_vec_sub_scalar` (*C function*), 63
`gr_generic_vec_sub_scalar_fmpq` (*C function*), 63
`gr_generic_vec_sub_scalar_fmpz` (*C function*), 63
`gr_generic_vec_sub_scalar_other` (*C function*), 63
`gr_generic_vec_sub_scalar_si` (*C function*), 63
`gr_generic_vec_sub_scalar_ui` (*C function*), 63
`gr_generic_vec_swap` (*C function*), 62
`gr_generic_vec_zero` (*C function*), 62
`gr_generic_write` (*C function*), 59
`gr_generic_write_n` (*C function*), 59
`gr_generic_zero` (*C function*), 59
`gr_gens` (*C function*), 43
`gr_gens_recursive` (*C function*), 43
`gr_get_d` (*C function*), 43
`gr_get_d_2exp_si` (*C function*), 43
`gr_get_fexpr` (*C function*), 43
`gr_get_fexpr_serialize` (*C function*), 43
`gr_get_fmpq` (*C function*), 43
`gr_get_fmpz` (*C function*), 43
`gr_get_fmpz_2exp_fmpz` (*C function*), 43
`gr_get_si` (*C function*), 43
`gr_get_str` (*C function*), 42
`gr_get_str_n` (*C function*), 42
`gr_get_ui` (*C function*), 43
`gr_glaisher` (*C function*), 66
`gr_gt` (*C function*), 49
`gr_harmonic` (*C function*), 68
`gr_harmonic_ui` (*C function*), 68
`gr_heap_clear` (*C function*), 41
`gr_heap_clear_vec` (*C function*), 41
`gr_heap_init` (*C function*), 41
`gr_heap_init_vec` (*C function*), 41
`gr_hermite_h` (*C function*), 69
`gr_hilbert_class_poly` (*C function*), 72
`gr_hurwitz_zeta` (*C function*), 71
`gr_hypgeom_of1` (*C function*), 70
`gr_hypgeom_1f1` (*C function*), 70
`gr_hypgeom_2f1` (*C function*), 70
`gr_hypgeom_pfq` (*C function*), 70
`gr_hypgeom_u` (*C function*), 70
`gr_i` (*C function*), 48
`gr_im` (*C function*), 48
`gr_in_domain` (*C function*), 40
`gr_init` (*C function*), 41
`gr_inv` (*C function*), 45
`gr_is_integer` (*C function*), 44
`gr_is_invertible` (*C function*), 45
`gr_is_neg_one` (*C function*), 44
`gr_is_one` (*C function*), 44
`gr_is_rational` (*C function*), 44
`gr_is_square` (*C function*), 47
`gr_is_zero` (*C function*), 44
`gr_jacobi_p` (*C function*), 69
`gr_jacobi_theta` (*C function*), 72
`gr_jacobi_theta_1` (*C function*), 72
`gr_jacobi_theta_2` (*C function*), 72
`gr_jacobi_theta_3` (*C function*), 72
`gr_jacobi_theta_4` (*C function*), 72
`gr_khinchin` (*C function*), 66
`gr_laguerre_l` (*C function*), 69
`gr_lambertw` (*C function*), 67
`gr_lambertw_fmpz` (*C function*), 67
`gr_lcm` (*C function*), 47
`gr_le` (*C function*), 49
`gr_legendre_p` (*C function*), 69
`gr_legendre_p_root_ui` (*C function*), 69
`gr_legendre_q` (*C function*), 69
`gr_lerch_phi` (*C function*), 71
`gr_lgamma` (*C function*), 68
`gr_log` (*C function*), 66
`gr_log10` (*C function*), 66
`gr_log1p` (*C function*), 66
`gr_log2` (*C function*), 66
`gr_log_barnes_g` (*C function*), 68
`gr_log_integral` (*C function*), 69
`gr_log_pi_i` (*C function*), 66
`gr_lt` (*C function*), 49
`gr_mat_add` (*C function*), 82
`gr_mat_add_fmpq` (*C function*), 82
`gr_mat_add_fmpz` (*C function*), 82
`gr_mat_add_scalar` (*C function*), 82
`gr_mat_add_scalar_other` (*C function*), 82
`gr_mat_add_si` (*C function*), 82
`gr_mat_add_ui` (*C function*), 82
`gr_mat_addmul_scalar` (*C function*), 83
`gr_mat_adjugate` (*C function*), 88
`gr_mat_adjugate_charpoly` (*C function*), 88
`gr_mat_adjugate_cofactor` (*C function*), 88
`gr_mat_apply_row_similarity` (*C function*), 90
`gr_mat_charpoly` (*C function*), 88
`gr_mat_charpoly_berkowitz` (*C function*), 88
`gr_mat_charpoly_danilevsky` (*C function*), 89
`gr_mat_charpoly_faddeev` (*C function*), 89
`gr_mat_charpoly_faddeev_bsgs` (*C function*), 89
`gr_mat_charpoly_from_hessenberg` (*C function*), 89
`gr_mat_charpoly_gauss` (*C function*), 89
`gr_mat_charpoly_generic` (*C function*), 88
`gr_mat_charpoly_householder` (*C function*), 89
`gr_mat_clear` (*C function*), 79
`gr_mat_companion` (*C function*), 90
`gr_mat_companion_fraction` (*C function*), 90
`gr_mat_concat_horizontal` (*C function*), 80
`gr_mat_concat_vertical` (*C function*), 80
`gr_mat_det` (*C function*), 86
`gr_mat_det_berkowitz` (*C function*), 86
`gr_mat_det_cofactor` (*C function*), 86

gr_mat_det_fflu (*C function*), 86
 gr_mat_det_generic (*C function*), 86
 gr_mat_det_generic_field (*C function*), 86
 gr_mat_det_generic_integral_domain (*C function*), 86
 gr_mat_det_lu (*C function*), 86
 gr_mat_diag_mul (*C function*), 84
 gr_mat_diagonalization (*C function*), 90
 gr_mat_diagonalization_generic (*C function*), 90
 gr_mat_diagonalization_precomp (*C function*), 90
 gr_mat_div_fmpq (*C function*), 83
 gr_mat_div_fmpz (*C function*), 83
 gr_mat_div_scalar (*C function*), 83
 gr_mat_div_scalar_other (*C function*), 83
 gr_mat_div_si (*C function*), 83
 gr_mat_div_ui (*C function*), 83
 gr_mat_eigenvalues (*C function*), 90
 gr_mat_eigenvalues_other (*C function*), 90
 GR_MAT_ENTRY (*C macro*), 78
 gr_mat_entry_ptr (*C function*), 78
 gr_mat_entrywise_binary_op (*C function*), 81
 gr_mat_entrywise_binary_op_scalar (*C function*), 81
 gr_mat_entrywise_binary_predicate_all (*C function*), 81
 gr_mat_entrywise_unary_op (*C function*), 81
 gr_mat_entrywise_unary_predicate_all (*C function*), 81
 gr_mat_entrywise_unary_predicate_any (*C function*), 81
 gr_mat_equal (*C function*), 80
 gr_mat_exp (*C function*), 91
 gr_mat_exp_jordan (*C function*), 91
 gr_mat_fflu (*C function*), 85
 gr_mat_find_nonzero_pivot (*C function*), 84
 gr_mat_find_nonzero_pivot_generic (*C function*), 84
 gr_mat_find_nonzero_pivot_large_abs (*C function*), 84
 gr_mat_func_jordan (*C function*), 91
 gr_mat_func_param_jordan (*C function*), 91
 gr_mat_generic (*C function*), 87
 gr_mat_gr_poly_evaluate (*C function*), 84
 gr_mat_gr_poly_solve_lode_newton (*C function*), 95
 gr_mat_hadamard (*C function*), 94
 gr_mat_hessenberg (*C function*), 92
 gr_mat_hessenberg_gauss (*C function*), 92
 gr_mat_hessenberg_householder (*C function*), 92
 gr_mat_hilbert (*C function*), 94
 gr_mat_init (*C function*), 79
 gr_mat_init_set (*C function*), 79
 gr_mat_inv (*C function*), 88
 gr_mat_invert_cols (*C function*), 81
 gr_mat_invert_rows (*C function*), 80
 gr_mat_is_col_orthogonal (*C function*), 92
 gr_mat_is_col_orthonormal (*C function*), 92
 gr_mat_is_diagonal (*C function*), 84
 gr_mat_is_empty (*C function*), 81
 gr_mat_is_hessenberg (*C function*), 92
 gr_mat_is_lower_triangular (*C function*), 84
 gr_mat_is_neg_one (*C function*), 80
 gr_mat_is_one (*C function*), 80
 gr_mat_is_orthogonal (*C function*), 92
 gr_mat_is_row_lll_reduced_naive (*C function*), 95
 gr_mat_is_row_lll_reduced_with_removal_naive (*C function*), 95
 gr_mat_is_row_orthogonal (*C function*), 92
 gr_mat_is_row_orthonormal (*C function*), 92
 gr_mat_is_scalar (*C function*), 80
 gr_mat_is_square (*C function*), 81
 gr_mat_is_upper_triangular (*C function*), 84
 gr_mat_is_zero (*C function*), 80
 gr_mat_jordan_blocks (*C function*), 91
 gr_mat_jordan_form (*C function*), 91
 gr_mat_jordan_transformation (*C function*), 91
 gr_mat_log (*C function*), 91
 gr_mat_log_jordan (*C function*), 91
 gr_mat_lq (*C function*), 93
 gr_mat_lq_generic (*C function*), 93
 gr_mat_lq_gso (*C function*), 93
 gr_mat_lq_recursive (*C function*), 93
 gr_mat_lu (*C function*), 84
 gr_mat_lu_classical (*C function*), 84
 gr_mat_lu_generic (*C function*), 84
 gr_mat_lu_recursive (*C function*), 84
 gr_mat_minpoly_field (*C function*), 89
 gr_mat_move_row (*C function*), 81
 gr_mat_mul (*C function*), 83
 gr_mat_mul_classical (*C function*), 83
 gr_mat_mul_diag (*C function*), 84
 gr_mat_mul_fmpq (*C function*), 82
 gr_mat_mul_fmpz (*C function*), 82
 gr_mat_mul_generic (*C function*), 83
 gr_mat_mul_rosowski (*C function*), 83
 gr_mat_mul_scalar (*C function*), 82
 gr_mat_mul_scalar_other (*C function*), 82
 gr_mat_mul_si (*C function*), 82
 gr_mat_mul_strassen (*C function*), 83
 gr_mat_mul_ui (*C function*), 82
 gr_mat_mul_waksman (*C function*), 83
 gr_mat_ncols (*C macro*), 78
 gr_mat_neg (*C function*), 82
 gr_mat_nonsingular_solve (*C function*), 86
 gr_mat_nonsingular_solve_den (*C function*), 86
 gr_mat_nonsingular_solve_den_fflu (*C function*), 86
 gr_mat_nonsingular_solve_fflu (*C function*), 86
 gr_mat_nonsingular_solve_fflu_precomp (*C function*), 86
 gr_mat_nonsingular_solve_lu (*C function*), 86

`gr_mat_nonsingular_solve_lu_precomp` (*C function*), 86
`gr_mat_nonsingular_solve_tril` (*C function*), 85
`gr_mat_nonsingular_solve_tril_classical` (*C function*), 85
`gr_mat_nonsingular_solve_tril_generic` (*C function*), 85
`gr_mat_nonsingular_solve_tril_recursive` (*C function*), 85
`gr_mat_nonsingular_solve_triu` (*C function*), 85
`gr_mat_nonsingular_solve_triu_classical` (*C function*), 85
`gr_mat_nonsingular_solve_triu_generic` (*C function*), 85
`gr_mat_nonsingular_solve_triu_recursive` (*C function*), 85
`gr_mat_norm_1` (*C function*), 82
`gr_mat_norm_frobenius` (*C function*), 82
`gr_mat_norm_inf` (*C function*), 82
`gr_mat_norm_max` (*C function*), 82
`gr_mat_nrows` (*C macro*), 78
`gr_mat_nullspace` (*C function*), 88
`gr_mat_nullspace_from_rref` (*C function*), 88
`gr_mat_nullspace_no_resize` (*C function*), 88
`gr_mat_one` (*C function*), 80
`gr_mat_ones` (*C function*), 94
`gr_mat_pascal` (*C function*), 94
`gr_mat_permanent` (*C function*), 87
`gr_mat_permanent_cofactor` (*C function*), 87
`gr_mat_permanent_glynn` (*C function*), 87
`gr_mat_permanent_glynn_threaded` (*C function*), 87
`gr_mat_permanent_ryser` (*C function*), 87
`gr_mat_pow_fmpq` (*C function*), 91
`gr_mat_pow_fmpq_jordan` (*C function*), 91
`gr_mat_pow_fmpz` (*C function*), 84
`gr_mat_pow_scalar` (*C function*), 91
`gr_mat_pow_scalar_jordan` (*C function*), 91
`gr_mat_pow_si` (*C function*), 84
`gr_mat_pow_ui` (*C function*), 84
`gr_mat_print` (*C function*), 79
`gr_mat_qr` (*C function*), 93
`gr_mat_randops` (*C function*), 92
`gr_mat_randpermdiag` (*C function*), 92
`gr_mat_randrank` (*C function*), 92
`gr_mat_randsimilar` (*C function*), 92
`gr_mat_randtest` (*C function*), 92
`gr_mat_randtest_orthogonal` (*C function*), 93
`gr_mat_rank` (*C function*), 87
`gr_mat_rank_fflu` (*C function*), 87
`gr_mat_rank_lu` (*C function*), 87
`gr_mat_reduce_row` (*C function*), 94
`gr_mat_reduce_row_generic` (*C function*), 94
`gr_mat_rref` (*C function*), 87
`gr_mat_rref_den` (*C function*), 87
`gr_mat_rref_den_fflu` (*C function*), 87
`gr_mat_rref_fflu` (*C function*), 87
`gr_mat_rref_lu` (*C function*), 87
`gr_mat_rsqr` (*C function*), 91
`gr_mat_scalar_add` (*C function*), 82
`gr_mat_scalar_mul` (*C function*), 82
`gr_mat_scalar_other_add` (*C function*), 82
`gr_mat_scalar_other_mul` (*C function*), 82
`gr_mat_scalar_other_sub` (*C function*), 82
`gr_mat_scalar_sub` (*C function*), 82
`gr_mat_set` (*C function*), 80
`gr_mat_set_fmpq` (*C function*), 80
`gr_mat_set_fmpq_mat` (*C function*), 80
`gr_mat_set_fmpz` (*C function*), 80
`gr_mat_set_fmpz_mat` (*C function*), 80
`gr_mat_set_gr_mat_other` (*C function*), 80
`gr_mat_set_jordan_blocks` (*C function*), 91
`gr_mat_set_scalar` (*C function*), 80
`gr_mat_set_si` (*C function*), 80
`gr_mat_set_ui` (*C function*), 80
`GR_MAT_SHALLOW_TRANSPOSE` (*C macro*), 79
`gr_mat_solve_field` (*C function*), 86
`gr_mat_sqr` (*C function*), 84
`gr_mat_sqrt` (*C function*), 91
`gr_mat_stirling` (*C function*), 94
`gr_mat_struct` (*C type*), 78
`gr_mat_sub` (*C function*), 82
`gr_mat_sub_fmpq` (*C function*), 82
`gr_mat_sub_fmpz` (*C function*), 82
`gr_mat_sub_scalar` (*C function*), 82
`gr_mat_sub_scalar_other` (*C function*), 82
`gr_mat_sub_si` (*C function*), 82
`gr_mat_sub_ui` (*C function*), 82
`gr_mat_submul_scalar` (*C function*), 83
`gr_mat_swap` (*C function*), 79
`gr_mat_swap_cols` (*C function*), 80
`gr_mat_swap_entrywise` (*C function*), 79
`gr_mat_swap_rows` (*C function*), 80
`gr_mat_t` (*C type*), 78
`gr_mat_test_approx_mul_max_norm` (*C function*), 96
`gr_mat_test_approx_mul_pos_entrywise_accurate` (*C function*), 96
`gr_mat_test_charpoly` (*C function*), 96
`gr_mat_test_det` (*C function*), 96
`gr_mat_test_lu` (*C function*), 96
`gr_mat_test_mul` (*C function*), 96
`gr_mat_test_nonsingular_solve_tril` (*C function*), 96
`gr_mat_test_nonsingular_solve_triu` (*C function*), 96
`GR_MAT_TMP_CLEAR_SHALLOW_TRANSPOSE` (*C macro*), 79
`GR_MAT_TMP_INIT_SHALLOW_TRANSPOSE` (*C macro*), 79
`gr_mat_trace` (*C function*), 86
`gr_mat_transpose` (*C function*), 80
`gr_mat_window_clear` (*C function*), 79
`gr_mat_window_init` (*C function*), 79

`gr_mat_write` (*C function*), 79
`gr_mat_zero` (*C function*), 80
`gr_max` (*C function*), 49
`gr_method` (*C type*), 51
`gr_method_tab_init` (*C function*), 51
`gr_method_tab_input` (*C type*), 51
`gr_min` (*C function*), 49
`gr_modular_delta` (*C function*), 72
`gr_modular_j` (*C function*), 72
`gr_modular_lambda` (*C function*), 72
`gr_mpoly_add` (*C function*), 125
`gr_mpoly_assert_canonical` (*C function*), 127
`gr_mpoly_canonical_associate` (*C function*), 126
`GR_MPOLY_CCTX` (*C macro*), 122
`gr_mpoly_clear` (*C function*), 123
`gr_mpoly_combine_like_terms` (*C function*), 127
`gr_mpoly_ctx_clear` (*C function*), 123
`gr_mpoly_ctx_init` (*C function*), 123
`gr_mpoly_ctx_init_rand` (*C function*), 123
`gr_mpoly_ctx_is_commutative_ring` (*C function*), 123
`gr_mpoly_ctx_is_field` (*C function*), 123
`gr_mpoly_ctx_is_integral_domain` (*C function*), 123
`gr_mpoly_ctx_is_ring` (*C function*), 123
`gr_mpoly_ctx_is_threadsafe` (*C function*), 123
`gr_mpoly_ctx_is_zero_ring` (*C function*), 123
`gr_mpoly_ctx_set_gen_names` (*C function*), 123
`gr_mpoly_ctx_struct` (*C type*), 122
`gr_mpoly_ctx_t` (*C type*), 122
`gr_mpoly_ctx_write` (*C function*), 123
`gr_mpoly_derivative` (*C function*), 126
`gr_mpoly_equal` (*C function*), 124
`gr_mpoly_fit_bits` (*C function*), 126
`gr_mpoly_fit_length` (*C function*), 126
`gr_mpoly_fit_length_fit_bits` (*C function*), 126
`gr_mpoly_fit_length_reset_bits` (*C function*), 126
`gr_mpoly_gen` (*C function*), 124
`gr_mpoly_gens` (*C function*), 124
`gr_mpoly_gens_recursive` (*C function*), 124
`gr_mpoly_get_coeff_scalar_fmpz` (*C function*), 125
`gr_mpoly_get_coeff_scalar_ui` (*C function*), 125
`gr_mpoly_init` (*C function*), 123
`gr_mpoly_init2` (*C function*), 123
`gr_mpoly_init3` (*C function*), 123
`gr_mpoly_integral` (*C function*), 126
`gr_mpoly_inv` (*C function*), 126
`gr_mpoly_is_canonical` (*C function*), 127
`gr_mpoly_is_gen` (*C function*), 124
`gr_mpoly_is_zero` (*C function*), 124
`gr_mpoly_length` (*C function*), 124
`GR_MPOLY_MCTX` (*C macro*), 122
`gr_mpoly_mul` (*C function*), 126
`gr_mpoly_mul_fmpq` (*C function*), 126
`gr_mpoly_mul_fmpz` (*C function*), 126
`gr_mpoly_mul_johnson` (*C function*), 126
`gr_mpoly_mul_monomial` (*C function*), 126
`gr_mpoly_mul_scalar` (*C function*), 126
`gr_mpoly_mul_si` (*C function*), 126
`gr_mpoly_mul_ui` (*C function*), 126
`gr_mpoly_neg` (*C function*), 125
`GR_MPOLY_NVARS` (*C macro*), 122
`gr_mpoly_print_pretty` (*C function*), 125
`gr_mpoly_push_term_scalar_fmpz` (*C function*), 127
`gr_mpoly_push_term_scalar_ui` (*C function*), 126
`gr_mpoly_randtest_bits` (*C function*), 125
`gr_mpoly_set` (*C function*), 123
`gr_mpoly_set_coeff_fmpq_fmpz` (*C function*), 125
`gr_mpoly_set_coeff_fmpq_ui` (*C function*), 125
`gr_mpoly_set_coeff_fmpz_fmpz` (*C function*), 125
`gr_mpoly_set_coeff_fmpz_ui` (*C function*), 125
`gr_mpoly_set_coeff_scalar_fmpz` (*C function*), 125
`gr_mpoly_set_coeff_scalar_ui` (*C function*), 125
`gr_mpoly_set_coeff_si_fmpz` (*C function*), 125
`gr_mpoly_set_coeff_si_ui` (*C function*), 125
`gr_mpoly_set_coeff_ui_fmpz` (*C function*), 125
`gr_mpoly_set_coeff_ui_ui` (*C function*), 125
`gr_mpoly_set_fmpq` (*C function*), 124
`gr_mpoly_set_fmpz` (*C function*), 124
`gr_mpoly_set_other` (*C function*), 124
`gr_mpoly_set_scalar` (*C function*), 124
`gr_mpoly_set_shallow` (*C function*), 123
`gr_mpoly_set_si` (*C function*), 124
`gr_mpoly_set_ui` (*C function*), 124
`gr_mpoly_sort_terms` (*C function*), 127
`gr_mpoly_struct` (*C type*), 122
`gr_mpoly_sub` (*C function*), 125
`gr_mpoly_swap` (*C function*), 123
`gr_mpoly_t` (*C type*), 122
`GR_MPOLY_VARS` (*C macro*), 122
`gr_mpoly_write_pretty` (*C function*), 125
`gr_mpoly_zero` (*C function*), 123
`gr_mul` (*C function*), 44
`gr_mul_2exp_fmpz` (*C function*), 45
`gr_mul_2exp_si` (*C function*), 45
`gr_mul_fmpq` (*C function*), 44
`gr_mul_fmpz` (*C function*), 44
`gr_mul_other` (*C function*), 44
`gr_mul_si` (*C function*), 44
`gr_mul_two` (*C function*), 45
`gr_mul_ui` (*C function*), 44
`GR_MUST_SUCCEED` (*C macro*), 39
`gr_neg` (*C function*), 44
`gr_neg_inf` (*C function*), 48
`gr_neg_one` (*C function*), 43

`gr_nint` (*C function*), 48
`gr_not_implemented` (*C function*), 51
`gr_not_in_domain` (*C function*), 51
`gr_numerator` (*C function*), 48
`gr_one` (*C function*), 43
`gr_other_add` (*C function*), 44
`gr_other_div` (*C function*), 45
`gr_other_divexact` (*C function*), 46
`gr_other_mul` (*C function*), 44
`gr_other_pow` (*C function*), 46
`gr_other_sub` (*C function*), 44
`GR_PARSE_BALANCE_ADDITIONS` (*C macro*), 59
`GR_PARSE_RING_EXPONENTS` (*C macro*), 59
`gr_partitions_fmpz` (*C function*), 69
`gr_partitions_ui` (*C function*), 69
`gr_partitions_vec` (*C function*), 69
`gr_pi` (*C function*), 66
`gr_poly_acos_series` (*C function*), 116
`gr_poly_acosh_series` (*C function*), 116
`gr_poly_add` (*C function*), 99
`gr_poly_add_fmpq` (*C function*), 101
`gr_poly_add_fmpz` (*C function*), 101
`gr_poly_add_scalar` (*C function*), 101
`gr_poly_add_si` (*C function*), 101
`gr_poly_add_ui` (*C function*), 101
`gr_poly_addmul_scalar` (*C function*), 102
`gr_poly_asin_series` (*C function*), 116
`gr_poly_asinh_series` (*C function*), 116
`gr_poly_atan_series` (*C function*), 116
`gr_poly_atanh_series` (*C function*), 116
`gr_poly_canonical_associate` (*C function*), 112
`gr_poly_clear` (*C function*), 98
`gr_poly_coeff_ptr` (*C function*), 98
`gr_poly_coeff_srcptr` (*C function*), 98
`gr_poly_compose` (*C function*), 110
`gr_poly_compose_divconquer` (*C function*), 110
`gr_poly_compose_horner` (*C function*), 110
`gr_poly_compose_mod` (*C function*), 118
`gr_poly_compose_mod_brent_kung` (*C function*), 118
`gr_poly_compose_mod_brent_kung_precomp_preinv` (*C function*), 120
`gr_poly_compose_mod_brent_kung_preinv` (*C function*), 119
`gr_poly_compose_mod_horner` (*C function*), 118
`gr_poly_compose_mod_horner_preinv` (*C function*), 119
`gr_poly_compose_mod_preinv` (*C function*), 119
`gr_poly_compose_series` (*C function*), 111
`gr_poly_compose_series_brent_kung` (*C function*), 111
`gr_poly_compose_series_divconquer` (*C function*), 111
`gr_poly_compose_series_horner` (*C function*), 111
`gr_poly_derivative` (*C function*), 112
`gr_poly_dispersion` (*C function*), 114
`gr_poly_dispersion_factor` (*C function*), 114
`gr_poly_dispersion_from_factors` (*C function*), 115
`gr_poly_dispersion_resultant` (*C function*), 114
`gr_poly_div` (*C function*), 104
`gr_poly_div_basecase` (*C function*), 104
`gr_poly_div_divconquer` (*C function*), 104
`gr_poly_div_newton` (*C function*), 104
`gr_poly_div_newton_n_preinv` (*C function*), 104
`gr_poly_div_scalar` (*C function*), 103
`gr_poly_div_series` (*C function*), 105
`gr_poly_div_series_basecase` (*C function*), 105
`gr_poly_div_series_divconquer` (*C function*), 105
`gr_poly_div_series_invmul` (*C function*), 105
`gr_poly_div_series_newton` (*C function*), 105
`gr_poly_divexact_basecase` (*C function*), 106
`gr_poly_divexact_basecase_bidirectional` (*C function*), 106
`gr_poly_divexact_bidirectional` (*C function*), 106
`gr_poly_divexact_series_basecase` (*C function*), 106
`gr_poly_divrem` (*C function*), 103
`gr_poly_divrem_basecase` (*C function*), 103
`gr_poly_divrem_divconquer` (*C function*), 103
`gr_poly_divrem_newton` (*C function*), 103
`gr_poly_divrem_newton_n_preinv` (*C function*), 105
`gr_poly_entry_ptr` (*C function*), 98
`gr_poly_entry_srcptr` (*C function*), 98
`gr_poly_equal` (*C function*), 99
`gr_poly_evaluate` (*C function*), 107
`gr_poly_evaluate_horner` (*C function*), 107
`gr_poly_evaluate_modular` (*C function*), 107
`gr_poly_evaluate_other` (*C function*), 107
`gr_poly_evaluate_other_horner` (*C function*), 107
`gr_poly_evaluate_other_rectangular` (*C function*), 107
`gr_poly_evaluate_rectangular` (*C function*), 107
`gr_poly_evaluate_vec_fast` (*C function*), 109
`gr_poly_evaluate_vec_iter` (*C function*), 109
`gr_poly_exp_series` (*C function*), 117
`gr_poly_exp_series_basecase` (*C function*), 117
`gr_poly_exp_series_basecase_mul` (*C function*), 117
`gr_poly_exp_series_newton` (*C function*), 117
`gr_poly_factor_squarefree` (*C function*), 114
`gr_poly_fit_length` (*C function*), 98
`gr_poly_gcd` (*C function*), 112
`gr_poly_gcd_euclidean` (*C function*), 112
`gr_poly_gcd_hgcd` (*C function*), 112
`gr_poly_gcd_subresultant` (*C function*), 112
`gr_poly_gen` (*C function*), 98
`gr_poly_get_coeff_scalar` (*C function*), 99
`gr_poly_get_fmpz_poly` (*C function*), 98

gr_poly_get_str (*C function*), 98
 gr_poly_init (*C function*), 98
 gr_poly_init2 (*C function*), 98
 gr_poly_integral (*C function*), 112
 gr_poly_interpolate (*C function*), 109
 gr_poly_interpolate_exact (*C function*), 109
 gr_poly_interpolate_fast (*C function*), 110
 gr_poly_inv_series (*C function*), 105
 gr_poly_inv_series_basecase (*C function*), 105
 gr_poly_inv_series_newton (*C function*), 105
 gr_poly_is_gen (*C function*), 99
 gr_poly_is_monic (*C function*), 112
 gr_poly_is_one (*C function*), 99
 gr_poly_is_scalar (*C function*), 99
 gr_poly_is_zero (*C function*), 99
 gr_poly_leading_taylor_shift (*C function*), 114
 gr_poly_length (*C function*), 98
 gr_poly_log1p_series (*C function*), 117
 gr_poly_log_series (*C function*), 117
 gr_poly_make_monic (*C function*), 112
 gr_poly_mul (*C function*), 99
 gr_poly_mul_fmpq (*C function*), 101
 gr_poly_mul_fmpz (*C function*), 101
 gr_poly_mul_karatsuba (*C function*), 100
 gr_poly_mul_scalar (*C function*), 101
 gr_poly_mul_si (*C function*), 101
 gr_poly_mul_toom33 (*C function*), 100
 gr_poly_mul_ui (*C function*), 101
 gr_poly_mullov (*C function*), 99
 gr_poly_mullov_bivariate_KS (*C function*), 100
 gr_poly_mullov_classical (*C function*), 100
 gr_poly_mullov_complex_reorder (*C function*), 100
 gr_poly_mullov_toom_serial (*C function*), 101
 gr_poly_mulmid (*C function*), 99
 gr_poly_mulmid_bivariate_KS (*C function*), 100
 gr_poly_mulmid_classical (*C function*), 100
 gr_poly_mulmid_complex_reorder (*C function*), 100
 gr_poly_mulmod (*C function*), 118
 gr_poly_mulmod_preinv (*C function*), 118
 gr_poly_neg (*C function*), 99
 gr_poly_neg_one (*C function*), 98
 gr_poly_newton_basis_evaluate (*C function*), 108
 gr_poly_newton_basis_from_monomial (*C function*), 108
 gr_poly_newton_basis_interpolate (*C function*), 108
 gr_poly_newton_basis_interpolate_exact (*C function*), 108
 gr_poly_newton_basis_to_monomial (*C function*), 108
 gr_poly_nth_derivative (*C function*), 112
 gr_poly_one (*C function*), 98
 gr_poly_pow_fmpz (*C function*), 102
 gr_poly_pow_series_fmpq_recurrence (*C function*), 102
 gr_poly_pow_series_ui (*C function*), 102
 gr_poly_pow_series_ui_binexp (*C function*), 102
 gr_poly_pow_ui (*C function*), 102
 gr_poly_pow_ui_binexp (*C function*), 102
 gr_poly_powmod_fmpz_binexp (*C function*), 118
 gr_poly_powmod_fmpz_binexp_preinv (*C function*), 118
 gr_poly_powmod_fmpz_sliding_preinv (*C function*), 118
 gr_poly_powmod_ui_binexp (*C function*), 118
 gr_poly_powmod_ui_binexp_preinv (*C function*), 118
 gr_poly_powmod_x_fmpz_preinv (*C function*), 118
 gr_poly_precompute_matrix (*C function*), 119
 gr_poly_print (*C function*), 98
 gr_poly_product_roots (*C function*), 109
 gr_poly_randtest (*C function*), 98
 gr_poly_rem (*C function*), 104
 gr_poly_resultant (*C function*), 113
 gr_poly_resultant_euclidean (*C function*), 113
 gr_poly_resultant_hgcd (*C function*), 113
 gr_poly_resultant_small (*C function*), 113
 gr_poly_resultant_sylvester (*C function*), 113
 gr_poly_reverse (*C function*), 98
 gr_poly_revert_series (*C function*), 111
 gr_poly_revert_series_lagrange (*C function*), 111
 gr_poly_revert_series_lagrange_fast (*C function*), 111
 gr_poly_revert_series_newton (*C function*), 111
 gr_poly_roots (*C function*), 116
 gr_poly_roots_other (*C function*), 116
 gr_poly_rsqrts_series (*C function*), 107
 gr_poly_rsqrts_series_basecase (*C function*), 107
 gr_poly_rsqrts_series_miller (*C function*), 107
 gr_poly_rsqrts_series_newton (*C function*), 107
 gr_poly_scalar_mul (*C function*), 101
 gr_poly_set (*C function*), 98
 gr_poly_set_coeff_fmpq (*C function*), 99
 gr_poly_set_coeff_fmpz (*C function*), 99
 gr_poly_set_coeff_scalar (*C function*), 99
 gr_poly_set_coeff_si (*C function*), 99
 gr_poly_set_coeff_ui (*C function*), 99
 gr_poly_set_fmpq (*C function*), 99
 gr_poly_set_fmpq_poly (*C function*), 98
 gr_poly_set_fmpz (*C function*), 99
 gr_poly_set_gr_poly_other (*C function*), 98
 gr_poly_set_scalar (*C function*), 99
 gr_poly_set_si (*C function*), 99
 gr_poly_set_str (*C function*), 98
 gr_poly_set_ui (*C function*), 99
 gr_poly_shift_equivalent (*C function*), 114

`gr_poly_shift_left` (*C function*), 102
`gr_poly_shift_right` (*C function*), 102
`gr_poly_shiftless_decomposition` (*C function*), 115
`gr_poly_shiftless_decomposition_factor` (*C function*), 115
`gr_poly_shiftless_decomposition_from_factors` (*C function*), 115
`gr_poly_sin_cos_series_basecase` (*C function*), 117
`gr_poly_sin_cos_series_tangent` (*C function*), 117
`gr_poly_sqrt_series` (*C function*), 107
`gr_poly_sqrt_series_basecase` (*C function*), 107
`gr_poly_sqrt_series_miller` (*C function*), 107
`gr_poly_sqrt_series_newton` (*C function*), 107
`gr_poly_squarefree_part` (*C function*), 114
`gr_poly_struct` (*C type*), 97
`gr_poly_sub` (*C function*), 99
`gr_poly_sub_fmpq` (*C function*), 101
`gr_poly_sub_fmpz` (*C function*), 101
`gr_poly_sub_scalar` (*C function*), 101
`gr_poly_sub_si` (*C function*), 101
`gr_poly_sub_ui` (*C function*), 101
`gr_poly_submul_scalar` (*C function*), 102
`gr_poly_swap` (*C function*), 98
`gr_poly_t` (*C type*), 97
`gr_poly_tan_series` (*C function*), 117
`gr_poly_tan_series_basecase` (*C function*), 117
`gr_poly_tan_series_newton` (*C function*), 117
`gr_poly_taylor_shift` (*C function*), 110
`gr_poly_taylor_shift_convolution` (*C function*), 110
`gr_poly_taylor_shift_divconquer` (*C function*), 110
`gr_poly_taylor_shift_horner` (*C function*), 110
`gr_poly_truncate` (*C function*), 98
`gr_poly_write` (*C function*), 98
`gr_poly_xgcd` (*C function*), 113
`gr_poly_xgcd_euclidean` (*C function*), 113
`gr_poly_xgcd_hgcd` (*C function*), 113
`gr_poly_zero` (*C function*), 98
`gr_polygamma` (*C function*), 71
`gr_polylog` (*C function*), 71
`gr_pos_inf` (*C function*), 48
`gr_pow` (*C function*), 46
`gr_pow_fmpq` (*C function*), 46
`gr_pow_fmpz` (*C function*), 46
`gr_pow_other` (*C function*), 46
`gr_pow_si` (*C function*), 46
`gr_pow_ui` (*C function*), 46
`gr_print` (*C function*), 42
`gr_println` (*C function*), 42
`gr_ptr` (*C type*), 38
`gr_randtest` (*C function*), 42
`gr_randtest_invertible` (*C function*), 42
`gr_randtest_not_zero` (*C function*), 42
`gr_randtest_small` (*C function*), 42
`gr_re` (*C function*), 48
`gr_rfac` (*C function*), 67
`gr_rfac_fmpz` (*C function*), 67
`gr_rfac_ui` (*C function*), 67
`gr_rfac_vec` (*C function*), 67
`gr_rgamma` (*C function*), 68
`gr_riemann_xi` (*C function*), 71
`gr_rising` (*C function*), 67
`gr_rising_ui` (*C function*), 67
`gr_rsqr` (*C function*), 47
`gr_sec` (*C function*), 66
`gr_sec_pi` (*C function*), 66
`gr_sech` (*C function*), 66
`gr_series_acos` (*C function*), 131
`gr_series_acosh` (*C function*), 131
`gr_series_add` (*C function*), 131
`gr_series_agm1` (*C function*), 131
`gr_series_airy` (*C function*), 131
`gr_series_airy_ai` (*C function*), 131
`gr_series_airy_ai_prime` (*C function*), 131
`gr_series_airy_bi` (*C function*), 131
`gr_series_airy_bi_prime` (*C function*), 131
`gr_series_asin` (*C function*), 131
`gr_series_asinh` (*C function*), 131
`gr_series_atan` (*C function*), 131
`gr_series_atanh` (*C function*), 131
`gr_series_beta_lower` (*C function*), 131
`gr_series_clear` (*C function*), 131
`gr_series_coeff_is_zero` (*C function*), 131
`gr_series_cos_integral` (*C function*), 131
`gr_series_cosh_integral` (*C function*), 131
`gr_series_ctx_clear` (*C function*), 131
`gr_series_ctx_init` (*C function*), 128
`gr_series_ctx_is_commutative_ring` (*C function*), 131
`gr_series_ctx_is_complex_vector_space` (*C function*), 131
`gr_series_ctx_is_integral_domain` (*C function*), 131
`gr_series_ctx_is_rational_vector_space` (*C function*), 131
`gr_series_ctx_is_real_vector_space` (*C function*), 131
`gr_series_ctx_is_ring` (*C function*), 131
`gr_series_ctx_set_gen_name` (*C function*), 131
`gr_series_ctx_set_gen_names` (*C function*), 131
`gr_series_ctx_write` (*C function*), 131
`gr_series_digamma` (*C function*), 131
`gr_series_dirichlet_hardy_theta` (*C function*), 131
`gr_series_dirichlet_hardy_z` (*C function*), 131
`gr_series_dirichlet_l` (*C function*), 131
`gr_series_div` (*C function*), 131
`gr_series_divexact` (*C function*), 131
`GR_SERIES_ELEM_CTX` (*C macro*), 130
`gr_series_elliptic_k` (*C function*), 131
`gr_series_equal` (*C function*), 131

gr_series_erf (*C function*), 131
 gr_series_erfc (*C function*), 131
 gr_series_erfi (*C function*), 131
 GR_SERIES_ERR_EXACT (*C macro*), 130
 GR_SERIES_ERR_MAX (*C macro*), 130
 GR_SERIES_ERROR (*C macro*), 130
 gr_series_exp (*C function*), 131
 gr_series_exp_integral_ei (*C function*), 131
 gr_series_fresnel (*C function*), 131
 gr_series_fresnel_c (*C function*), 131
 gr_series_fresnel_s (*C function*), 131
 gr_series_gamma (*C function*), 131
 gr_series_gamma_lower (*C function*), 131
 gr_series_gamma_upper (*C function*), 131
 gr_series_gen (*C function*), 131
 gr_series_gens_recursive (*C function*), 131
 gr_series_hurwitz_zeta (*C function*), 131
 gr_series_hypgeom_pfq (*C function*), 131
 gr_series_init (*C function*), 131
 gr_series_inv (*C function*), 131
 gr_series_is_one (*C function*), 131
 gr_series_is_zero (*C function*), 131
 gr_series_jacobi_theta (*C function*), 131
 gr_series_jacobi_theta_1 (*C function*), 131
 gr_series_jacobi_theta_2 (*C function*), 131
 gr_series_jacobi_theta_3 (*C function*), 131
 gr_series_jacobi_theta_4 (*C function*), 131
 gr_series_lgamma (*C function*), 131
 gr_series_log (*C function*), 131
 gr_series_log_integral (*C function*), 131
 gr_series_mod_acos (*C function*), 129
 gr_series_mod_acosh (*C function*), 129
 gr_series_mod_add (*C function*), 129
 gr_series_mod_asin (*C function*), 129
 gr_series_mod_asinh (*C function*), 129
 gr_series_mod_atan (*C function*), 129
 gr_series_mod_atanh (*C function*), 129
 gr_series_mod_clear (*C function*), 129
 gr_series_mod_ctx_clear (*C function*), 129
 gr_series_mod_ctx_init (*C function*), 128
 gr_series_mod_ctx_is_commutative_ring (*C function*), 129
 gr_series_mod_ctx_is_complex_vector_space (*C function*), 129
 gr_series_mod_ctx_is_field (*C function*), 129
 gr_series_mod_ctx_is_integral_domain (*C function*), 129
 gr_series_mod_ctx_is_rational_vector_space (*C function*), 129
 gr_series_mod_ctx_is_real_vector_space (*C function*), 129
 gr_series_mod_ctx_is_ring (*C function*), 129
 gr_series_mod_ctx_set_gen_name (*C function*), 129
 gr_series_mod_ctx_set_gen_names (*C function*), 129
 gr_series_mod_ctx_write (*C function*), 129
 gr_series_mod_div (*C function*), 129
 GR_SERIES_MOD_ELEM_CTX (*C macro*), 128
 gr_series_mod_equal (*C function*), 129
 gr_series_mod_exp (*C function*), 129
 gr_series_mod_gen (*C function*), 129
 gr_series_mod_gens_recursive (*C function*), 129
 gr_series_mod_init (*C function*), 129
 gr_series_mod_inv (*C function*), 129
 gr_series_mod_is_one (*C function*), 129
 gr_series_mod_is_zero (*C function*), 129
 gr_series_mod_log (*C function*), 129
 gr_series_mod_mul (*C function*), 129
 GR_SERIES_MOD_N (*C macro*), 128
 gr_series_mod_neg (*C function*), 129
 gr_series_mod_one (*C function*), 129
 gr_series_mod_randtest (*C function*), 129
 gr_series_mod_rsqr (*C function*), 129
 gr_series_mod_set (*C function*), 129
 gr_series_mod_set_fmpq (*C function*), 129
 gr_series_mod_set_fmpz (*C function*), 129
 gr_series_mod_set_other (*C function*), 129
 gr_series_mod_set_shallow (*C function*), 129
 gr_series_mod_set_si (*C function*), 129
 gr_series_mod_set_ui (*C function*), 129
 gr_series_mod_sqrt (*C function*), 129
 gr_series_mod_sub (*C function*), 129
 gr_series_mod_swap (*C function*), 129
 gr_series_mod_tan (*C function*), 129
 gr_series_mod_write (*C function*), 129
 gr_series_mod_zero (*C function*), 129
 gr_series_mul (*C function*), 131
 gr_series_neg (*C function*), 131
 gr_series_one (*C function*), 131
 GR_SERIES_POLY (*C macro*), 130
 gr_series_polylog (*C function*), 131
 GR_SERIES_PREC (*C macro*), 130
 gr_series_randtest (*C function*), 131
 gr_series_rgamma (*C function*), 131
 gr_series_rsqr (*C function*), 131
 gr_series_set (*C function*), 131
 gr_series_set_fmpq (*C function*), 131
 gr_series_set_fmpz (*C function*), 131
 gr_series_set_gr_poly (*C function*), 131
 gr_series_set_other (*C function*), 131
 gr_series_set_scalar (*C function*), 131
 gr_series_set_shallow (*C function*), 131
 gr_series_set_si (*C function*), 131
 gr_series_set_ui (*C function*), 131
 gr_series_sin_integral (*C function*), 131
 gr_series_sinh_integral (*C function*), 131
 gr_series_sqrt (*C function*), 131
 gr_series_struct (*C type*), 130
 gr_series_sub (*C function*), 131
 gr_series_swap (*C function*), 131
 gr_series_t (*C type*), 130
 gr_series_tan (*C function*), 131
 gr_series_vec_struct (*C type*), 130
 gr_series_vec_t (*C type*), 130

`gr_series_weierstrass_p` (*C function*), 131
`gr_series_write` (*C function*), 131
`gr_series_zero` (*C function*), 131
`gr_set` (*C function*), 42
`gr_set_d` (*C function*), 42
`gr_set_fexpr` (*C function*), 43
`gr_set_fmpq` (*C function*), 42
`gr_set_fmpz` (*C function*), 42
`gr_set_fmpz_10exp_fmpz` (*C function*), 43
`gr_set_fmpz_2exp_fmpz` (*C function*), 43
`gr_set_interval_mid_rad` (*C function*), 49
`gr_set_other` (*C function*), 42
`gr_set_shallow` (*C function*), 41
`gr_set_si` (*C function*), 42
`gr_set_str` (*C function*), 42
`gr_set_ui` (*C function*), 42
`gr_sgn` (*C function*), 48
`gr_sin` (*C function*), 66
`gr_sin_cos` (*C function*), 66
`gr_sin_cos_pi` (*C function*), 66
`gr_sin_integral` (*C function*), 69
`gr_sin_pi` (*C function*), 66
`gr_sinc` (*C function*), 66
`gr_sinc_pi` (*C function*), 66
`gr_sinh` (*C function*), 66
`gr_sinh_cosh` (*C function*), 66
`gr_sinh_integral` (*C function*), 69
`gr_spherical_y_si` (*C function*), 69
`gr_sqr` (*C function*), 45
`gr_sqrt` (*C function*), 47
`gr_srcptr` (*C type*), 38
`gr_static_method_table` (*C type*), 51
`gr_stieltjes` (*C function*), 71
`gr_stirling_s1_ui_vec` (*C function*), 68
`gr_stirling_s1_uiui` (*C function*), 68
`gr_stirling_s1u_ui_vec` (*C function*), 68
`gr_stirling_s1u_uiui` (*C function*), 68
`gr_stirling_s2_ui_vec` (*C function*), 68
`gr_stirling_s2_uiui` (*C function*), 68
`gr_sub` (*C function*), 44
`gr_sub_fmpq` (*C function*), 44
`gr_sub_fmpz` (*C function*), 44
`gr_sub_other` (*C function*), 44
`gr_sub_si` (*C function*), 44
`gr_sub_ui` (*C function*), 44
`gr_submul` (*C function*), 45
`gr_submul_fmpq` (*C function*), 45
`gr_submul_fmpz` (*C function*), 45
`gr_submul_other` (*C function*), 45
`gr_submul_si` (*C function*), 45
`gr_submul_ui` (*C function*), 45
`GR_SUCCESS` (*C macro*), 38
`gr_swap` (*C function*), 41
`gr_tan` (*C function*), 66
`gr_tan_pi` (*C function*), 66
`gr_tanh` (*C function*), 66
`GR_TEST_FAIL` (*C macro*), 39
`gr_test_ring` (*C function*), 52
`GR_TMP_CLEAR` (*C macro*), 41
`GR_TMP_CLEAR2` (*C macro*), 41
`GR_TMP_CLEAR3` (*C macro*), 41
`GR_TMP_CLEAR4` (*C macro*), 41
`GR_TMP_CLEAR5` (*C macro*), 41
`GR_TMP_CLEAR_VEC` (*C macro*), 41
`GR_TMP_INIT` (*C macro*), 41
`GR_TMP_INIT2` (*C macro*), 41
`GR_TMP_INIT3` (*C macro*), 41
`GR_TMP_INIT4` (*C macro*), 41
`GR_TMP_INIT5` (*C macro*), 41
`GR_TMP_INIT_VEC` (*C macro*), 41
`gr_trunc` (*C function*), 48
`gr_uinf` (*C function*), 48
`GR_UNABLE` (*C macro*), 39
`gr_undefined` (*C function*), 48
`gr_unknown` (*C function*), 48
`gr_vec_append` (*C function*), 73
`gr_vec_clear` (*C function*), 73
`gr_vec_contains` (*C function*), 77
`GR_VEC_ENTRY` (*C macro*), 73
`gr_vec_entry_ptr` (*C function*), 73
`gr_vec_entry_srcptr` (*C function*), 73
`gr_vec_fit_length` (*C function*), 73
`gr_vec_init` (*C function*), 73
`gr_vec_length` (*C function*), 73
`gr_vec_permute` (*C function*), 77
`gr_vec_print` (*C function*), 73
`gr_vec_set` (*C function*), 73
`gr_vec_set_length` (*C function*), 73
`gr_vec_sort` (*C function*), 77
`gr_vec_struct` (*C type*), 73
`gr_vec_t` (*C type*), 73
`gr_vec_write` (*C function*), 73
`gr_weierstrass_p` (*C function*), 72
`gr_weierstrass_p_inv` (*C function*), 72
`gr_weierstrass_p_prime` (*C function*), 72
`gr_weierstrass_sigma` (*C function*), 72
`gr_weierstrass_zeta` (*C function*), 72
`gr_write` (*C function*), 42
`gr_write_n` (*C function*), 42
`gr_zero` (*C function*), 43
`gr_zeta` (*C function*), 71
`gr_zeta_nzeros` (*C function*), 71
`gr_zeta_ui` (*C function*), 71
`gr_zeta_zero` (*C function*), 71
`gr_zeta_zero_vec` (*C function*), 71
`Greater` (*C macro*), 901
`GreaterEqual` (*C macro*), 901
`Guess` (*C macro*), 900

H

`HankelH1` (*C macro*), 910
`HankelH2` (*C macro*), 910
`HarmonicNumber` (*C macro*), 909
`HermiteH` (*C macro*), 909
`HilbertClassPolynomial` (*C macro*), 913
`HilbertMatrix` (*C macro*), 905

HurwitzZeta (*C macro*), 911
 Hypergeometric0F1 (*C macro*), 911
 Hypergeometric0F1Regularized (*C macro*), 911
 Hypergeometric1F1 (*C macro*), 911
 Hypergeometric1F1Regularized (*C macro*), 911
 Hypergeometric1F2 (*C macro*), 911
 Hypergeometric1F2Regularized (*C macro*), 911
 Hypergeometric2F0 (*C macro*), 911
 Hypergeometric2F1 (*C macro*), 911
 Hypergeometric2F1Regularized (*C macro*), 911
 Hypergeometric2F2 (*C macro*), 911
 Hypergeometric2F2Regularized (*C macro*), 911
 Hypergeometric3F2 (*C macro*), 911
 Hypergeometric3F2Regularized (*C macro*), 911
 HypergeometricU (*C macro*), 911
 HypergeometricUStar (*C macro*), 911
 HypergeometricUStarRemainder (*C macro*), 911
 hypgeom_bound (*C function*), 797
 hypgeom_clear (*C function*), 797
 hypgeom_estimate_terms (*C function*), 797
 hypgeom_init (*C function*), 797
 hypgeom_precompute (*C function*), 797
 hypgeom_struct (*C type*), 797
 hypgeom_t (*C type*), 797

I

IdentityMatrix (*C macro*), 905
 ifft_butterfly (*C function*), 299
 ifft_butterfly_sqrt2 (*C function*), 301
 ifft_butterfly_twiddle (*C function*), 302
 ifft_mfa_truncate_sqrt2 (*C function*), 303
 ifft_mfa_truncate_sqrt2_outer (*C function*),
 304
 ifft_negacyclic (*C function*), 304
 ifft_radix2 (*C function*), 300
 ifft_radix2_twiddle (*C function*), 302
 ifft_truncate (*C function*), 301
 ifft_truncate1 (*C function*), 301
 ifft_truncate1_twiddle (*C function*), 302
 ifft_truncate_sqrt2 (*C function*), 302
 Im (*C macro*), 906
 Implies (*C macro*), 898
 IncompleteBeta (*C macro*), 910
 IncompleteBetaRegularized (*C macro*), 910
 IncompleteEllipticE (*C macro*), 912
 IncompleteEllipticF (*C macro*), 912
 IncompleteEllipticPi (*C macro*), 912
 IndefiniteIntegralEqual (*C macro*), 906
 Infimum (*C macro*), 903
 Infinity (*C macro*), 902
 init_all_clocks (*C function*), 23
 init_clock (*C function*), 23
 IntegersGreaterEqual (*C macro*), 901
 IntegersLessEqual (*C macro*), 901
 Integral (*C macro*), 904
 Intersection (*C macro*), 899
 Interval (*C macro*), 901
 IsEven (*C macro*), 906

IsHolomorphicOn (*C macro*), 904
 IsMeromorphicOn (*C macro*), 904
 IsOdd (*C macro*), 906
 IsPrime (*C macro*), 907
 Item (*C macro*), 898

J

JacobiP (*C macro*), 909
 JacobiSymbol (*C macro*), 907
 JacobiTheta (*C macro*), 912
 JacobiThetaEpsilon (*C macro*), 913
 JacobiThetaPermutation (*C macro*), 913
 JacobiThetaQ (*C macro*), 912

K

KeiperLiLambda (*C macro*), 912
 KhinchinConstant (*C macro*), 900
 KroneckerDelta (*C macro*), 906
 KroneckerSymbol (*C macro*), 907

L

LaguerreL (*C macro*), 909
 LambertW (*C macro*), 908
 LandauG (*C macro*), 908
 Lattice (*C macro*), 902
 LCM (*C macro*), 907
 LeftLimit (*C macro*), 904
 LegendreP (*C macro*), 909
 LegendrePolynomialZero (*C macro*), 909
 LegendreSymbol (*C macro*), 907
 Length (*C macro*), 899
 LerchPhi (*C macro*), 911
 Less (*C macro*), 901
 LessEqual (*C macro*), 901
 Limit (*C macro*), 904
 LiouvilleLambda (*C macro*), 907
 List (*C macro*), 898
 Log (*C macro*), 907
 LogBarnesG (*C macro*), 909
 LogBarnesGRemainder (*C macro*), 909
 LogGamma (*C macro*), 909
 Logic (*C macro*), 913
 LogIntegral (*C macro*), 910
 LowerGamma (*C macro*), 910

M

mag_add (*C function*), 587
 mag_add_2exp_fmpz (*C function*), 587
 mag_add_lower (*C function*), 587
 mag_add_ui (*C function*), 587
 mag_add_ui_2exp_si (*C function*), 587
 mag_add_ui_lower (*C function*), 587
 mag_addmul (*C function*), 587
 mag_allocated_bytes (*C function*), 584
 mag_atan (*C function*), 589
 mag_atan_lower (*C function*), 589
 mag_bernoulli_div_fac_ui (*C function*), 590
 mag_bin_uiui (*C function*), 590

`mag_binpow_uiui` (*C function*), 589
`mag_clear` (*C function*), 584
`mag_cmp` (*C function*), 586
`mag_cmp_2exp_si` (*C function*), 586
`mag_const_pi` (*C function*), 589
`mag_const_pi_lower` (*C function*), 589
`mag_cosh` (*C function*), 589
`mag_cosh_lower` (*C function*), 590
`mag_div` (*C function*), 587
`mag_div_fmpz` (*C function*), 587
`mag_div_lower` (*C function*), 587
`mag_div_ui` (*C function*), 587
`mag_dump_file` (*C function*), 586
`mag_dump_str` (*C function*), 586
`mag_equal` (*C function*), 586
`mag_exp` (*C function*), 589
`mag_exp_lower` (*C function*), 589
`mag_exp_tail` (*C function*), 589
`mag_expinv` (*C function*), 589
`mag_expinv_lower` (*C function*), 589
`mag_expm1` (*C function*), 589
`mag_fac_ui` (*C function*), 590
`mag_fast_add_2exp_si` (*C function*), 588
`mag_fast_addmul` (*C function*), 588
`mag_fast_init_set` (*C function*), 588
`mag_fast_init_set_arf` (*C function*), 606
`mag_fast_is_zero` (*C function*), 588
`mag_fast_mul` (*C function*), 588
`mag_fast_mul_2exp_si` (*C function*), 588
`mag_fast_zero` (*C function*), 588
`mag_fprint` (*C function*), 586
`mag_geom_series` (*C function*), 589
`mag_get_d` (*C function*), 585
`mag_get_d_log2_approx` (*C function*), 585
`mag_get_fmpq` (*C function*), 586
`mag_get_fmpz` (*C function*), 586
`mag_get_fmpz_lower` (*C function*), 586
`mag_hurwitz_zeta_uiui` (*C function*), 590
`mag_hypot` (*C function*), 588
`mag_inf` (*C function*), 585
`mag_init` (*C function*), 584
`mag_init_set` (*C function*), 585
`mag_init_set_arf` (*C function*), 606
`mag_inv` (*C function*), 588
`mag_inv_lower` (*C function*), 588
`mag_is_finite` (*C function*), 585
`mag_is_inf` (*C function*), 585
`mag_is_special` (*C function*), 585
`mag_is_zero` (*C function*), 585
`mag_load_file` (*C function*), 586
`mag_load_str` (*C function*), 586
`mag_log` (*C function*), 589
`mag_log1p` (*C function*), 589
`mag_log_lower` (*C function*), 589
`mag_log_ui` (*C function*), 589
`mag_max` (*C function*), 586
`mag_min` (*C function*), 586
`mag_mul` (*C function*), 587
`mag_mul_2exp_fmpz` (*C function*), 587
`mag_mul_2exp_si` (*C function*), 587
`mag_mul_fmpz` (*C function*), 587
`mag_mul_fmpz_lower` (*C function*), 587
`mag_mul_lower` (*C function*), 587
`mag_mul_ui` (*C function*), 587
`mag_mul_ui_lower` (*C function*), 587
`mag_neg_log` (*C function*), 589
`mag_neg_log_lower` (*C function*), 589
`mag_one` (*C function*), 585
`mag_polylog_tail` (*C function*), 590
`mag_pow_fmpz` (*C function*), 588
`mag_pow_fmpz_lower` (*C function*), 588
`mag_pow_ui` (*C function*), 588
`mag_pow_ui_lower` (*C function*), 588
`mag_print` (*C function*), 586
`mag_randtest` (*C function*), 587
`mag_randtest_special` (*C function*), 587
`mag_rfac_ui` (*C function*), 590
`mag_root` (*C function*), 589
`mag_rsqrt` (*C function*), 588
`mag_rsqrt_lower` (*C function*), 588
`mag_set` (*C function*), 585
`mag_set_d` (*C function*), 585
`mag_set_d_2exp_fmpz` (*C function*), 585
`mag_set_d_2exp_fmpz_lower` (*C function*), 585
`mag_set_d_lower` (*C function*), 585
`mag_set_fmpz` (*C function*), 585
`mag_set_fmpz_2exp_fmpz` (*C function*), 585
`mag_set_fmpz_2exp_fmpz_lower` (*C function*), 585
`mag_set_fmpz_lower` (*C function*), 585
`mag_set_ui` (*C function*), 585
`mag_set_ui_2exp_si` (*C function*), 585
`mag_set_ui_lower` (*C function*), 585
`mag_sinh` (*C function*), 590
`mag_sinh_lower` (*C function*), 590
`mag_sqrt` (*C function*), 588
`mag_sqrt_lower` (*C function*), 588
`mag_struct` (*C type*), 584
`mag_sub` (*C function*), 587
`mag_sub_lower` (*C function*), 587
`mag_swap` (*C function*), 584
`mag_t` (*C type*), 584
`mag_zero` (*C function*), 585
`Matrices` (*C macro*), 905
`Matrix` (*C macro*), 905
`Matrix2x2` (*C macro*), 905
`Max` (*C macro*), 906
`Maximum` (*C macro*), 903
`MeromorphicDerivative` (*C macro*), 904
`MeromorphicLimit` (*C macro*), 904
`Min` (*C macro*), 906
`Minimum` (*C macro*), 903
`Mod` (*C macro*), 906
`ModularGroupAction` (*C macro*), 913
`ModularGroupFundamentalDomain` (*C macro*), 913
`ModularJ` (*C macro*), 912

-
- ModularLambda (*C macro*), 912
 - ModularLambdaFundamentalDomain (*C macro*), 913
 - MoebiusMu (*C macro*), 907
 - mpn_addmod_2expp1_1 (*C function*), 298
 - mpn_ctx_clear (*C function*), 306
 - mpn_ctx_init (*C function*), 306
 - mpn_ctx_mpn_mul (*C function*), 306
 - mpn_ctx_struct (*C type*), 306
 - mpn_ctx_t (*C type*), 306
 - mpn_div_2expmod_2expp1 (*C function*), 299
 - mpn_mod_add (*C function*), 456
 - mpn_mod_add_fmpz (*C function*), 456
 - mpn_mod_add_si (*C function*), 456
 - mpn_mod_add_ui (*C function*), 456
 - mpn_mod_addmul (*C function*), 456
 - mpn_mod_addmul_fmpz (*C function*), 456
 - mpn_mod_addmul_si (*C function*), 456
 - mpn_mod_addmul_ui (*C function*), 456
 - mpn_mod_clear (*C function*), 456
 - mpn_mod_ctx_clear (*C function*), 456
 - mpn_mod_ctx_is_field (*C function*), 456
 - MPN_MOD_CTX_IS_PRIME (*C macro*), 455
 - MPN_MOD_CTX_MODULUS (*C macro*), 455
 - MPN_MOD_CTX_MODULUS_BITS (*C macro*), 455
 - MPN_MOD_CTX_MODULUS_NORMED (*C macro*), 455
 - MPN_MOD_CTX_MODULUS_PREINV (*C macro*), 455
 - MPN_MOD_CTX_NLIMBS (*C macro*), 455
 - MPN_MOD_CTX_NORM (*C macro*), 455
 - mpn_mod_ctx_set_is_field (*C function*), 455
 - mpn_mod_ctx_write (*C function*), 456
 - mpn_mod_div (*C function*), 456
 - mpn_mod_equal (*C function*), 456
 - mpn_mod_fmma (*C function*), 456
 - mpn_mod_get_fmpz (*C function*), 456
 - mpn_mod_init (*C function*), 456
 - mpn_mod_inv (*C function*), 456
 - mpn_mod_is_neg_one (*C function*), 456
 - mpn_mod_is_one (*C function*), 456
 - mpn_mod_is_zero (*C function*), 456
 - mpn_mod_mat_det (*C function*), 458
 - mpn_mod_mat_lu (*C function*), 458
 - mpn_mod_mat_lu_classical_delayed (*C function*), 458
 - mpn_mod_mat_mul (*C function*), 457
 - mpn_mod_mat_mul_multi_mod (*C function*), 457
 - mpn_mod_mat_mul_waksman (*C function*), 457
 - mpn_mod_mat_nonsingular_solve_tril (*C function*), 457
 - mpn_mod_mat_nonsingular_solve_triu (*C function*), 457
 - MPN_MOD_MAX_LIMBS (*C macro*), 455
 - MPN_MOD_MIN_LIMBS (*C macro*), 455
 - mpn_mod_mul (*C function*), 456
 - mpn_mod_mul_fmpz (*C function*), 456
 - mpn_mod_mul_si (*C function*), 456
 - mpn_mod_mul_ui (*C function*), 456
 - mpn_mod_neg (*C function*), 456
 - mpn_mod_neg_one (*C function*), 456
 - mpn_mod_one (*C function*), 456
 - mpn_mod_randtest (*C function*), 456
 - mpn_mod_set (*C function*), 456
 - mpn_mod_set_fmpz (*C function*), 456
 - mpn_mod_set_mpn (*C function*), 456
 - mpn_mod_set_other (*C function*), 456
 - mpn_mod_set_si (*C function*), 456
 - mpn_mod_set_ui (*C function*), 456
 - mpn_mod_sqr (*C function*), 456
 - mpn_mod_sub (*C function*), 456
 - mpn_mod_sub_fmpz (*C function*), 456
 - mpn_mod_sub_si (*C function*), 456
 - mpn_mod_sub_ui (*C function*), 456
 - mpn_mod_submul (*C function*), 456
 - mpn_mod_submul_fmpz (*C function*), 456
 - mpn_mod_submul_si (*C function*), 456
 - mpn_mod_submul_ui (*C function*), 456
 - mpn_mod_swap (*C function*), 456
 - mpn_mod_write (*C function*), 456
 - mpn_mod_zero (*C function*), 456
 - mpn_mul_2expmod_2expp1 (*C function*), 298
 - mpn_mul_default_mpn_ctx (*C function*), 306
 - mpn_negmod_2expp1 (*C function*), 298
 - MPN_NORM (*C macro*), 279
 - mpn_normmod_2expp1 (*C function*), 298
 - MPN_SWAP (*C macro*), 279
 - mpoly_ctx_clear (*C function*), 26
 - mpoly_ctx_init (*C function*), 26
 - mpoly_ctx_init_rand (*C function*), 26
 - mpoly_ctx_struct (*C type*), 26
 - mpoly_ctx_t (*C type*), 26
 - mpoly_exp_bits_required_ffmpz (*C function*), 28
 - mpoly_exp_bits_required_pfmmpz (*C function*), 28
 - mpoly_exp_bits_required_ui (*C function*), 28
 - mpoly_get_cmpmask (*C function*), 27
 - mpoly_get_monomial_ffmpz (*C function*), 29
 - mpoly_get_monomial_pfmmpz (*C function*), 29
 - mpoly_get_monomial_ui (*C function*), 29
 - mpoly_main_variable_terms1 (*C function*), 31
 - mpoly_max_degrees_tight (*C function*), 28
 - mpoly_max_fields_fmpz (*C function*), 28
 - mpoly_max_fields_ui_sp (*C function*), 28
 - mpoly_monomial_add (*C function*), 27
 - mpoly_monomial_add_mp (*C function*), 27
 - mpoly_monomial_cmp (*C function*), 27
 - mpoly_monomial_divides (*C function*), 28
 - mpoly_monomial_divides1 (*C function*), 28
 - mpoly_monomial_divides_mp (*C function*), 28
 - mpoly_monomial_divides_tight (*C function*), 28
 - mpoly_monomial_equal (*C function*), 27
 - mpoly_monomial_exists (*C function*), 28
 - mpoly_monomial_gt (*C function*), 27
 - mpoly_monomial_is_zero (*C function*), 27
 - mpoly_monomial_lt (*C function*), 27
 - mpoly_monomial_mul_ui (*C function*), 27

`mpoly_monomial_overflows` (*C function*), 27
`mpoly_monomial_overflows1` (*C function*), 27
`mpoly_monomial_overflows_mp` (*C function*), 27
`mpoly_monomial_set` (*C function*), 27
`mpoly_monomial_sub` (*C function*), 27
`mpoly_monomial_sub_mp` (*C function*), 27
`mpoly_monomial_swap` (*C function*), 27
`mpoly_ordering_isdeg` (*C function*), 26
`mpoly_ordering_isrev` (*C function*), 26
`mpoly_ordering_print` (*C function*), 26
`mpoly_ordering_randtest` (*C function*), 26
`mpoly_pack_monomials_tight` (*C function*), 30
`mpoly_pack_vec_fmpz` (*C function*), 30
`mpoly_pack_vec_ui` (*C function*), 30
`mpoly_repack_monomials` (*C function*), 30
`mpoly_search_monomials` (*C function*), 29
`mpoly_set_monomial_ffmpz` (*C function*), 29
`mpoly_set_monomial_pfmpz` (*C function*), 29
`mpoly_set_monomial_ui` (*C function*), 29
`mpoly_term_exp_fits_si` (*C function*), 29
`mpoly_term_exp_fits_ui` (*C function*), 29
`mpoly_unpack_monomials_tight` (*C function*), 30
`mpoly_unpack_vec_fmpz` (*C function*), 30
`mpoly_unpack_vec_ui` (*C function*), 30
`MPZ_MIN_ALLOC` (*C macro*), 156
`Mul` (*C macro*), 900
`mul_mfa_truncate_sqrt2` (*C function*), 305
`mul_precomp_struct` (*C type*), 307
`mul_truncate_sqrt2` (*C function*), 305
`MultiZetaValue` (*C macro*), 911

N

`n_addmod` (*C function*), 141
`n_binvert` (*C function*), 141
`n_cbrt` (*C function*), 149
`n_cbrt_binary_search` (*C function*), 149
`n_cbrt_chebyshev_approx` (*C function*), 149
`n_cbrtrem` (*C function*), 149
`n_cleanup_primes` (*C function*), 145
`n_clog` (*C function*), 137
`n_clog_2exp` (*C function*), 137
`n_compute_primes` (*C function*), 145
`n_CRT` (*C function*), 148
`n_discrete_log_bsgs` (*C function*), 153
`n_div2_preinv` (*C function*), 139
`n_divides` (*C function*), 144
`n_divrem2_precomp` (*C function*), 139
`n_divrem2_preinv` (*C function*), 138
`n_divrem_norm` (*C function*), 138
`n_divrem_preinv` (*C function*), 138
`n_divrem_preinv_unnorm` (*C function*), 138
`n_euler_phi` (*C function*), 153
`n_factor` (*C function*), 151
`n_factor_ecm` (*C function*), 154
`n_factor_ecm_add` (*C function*), 154
`n_factor_ecm_double` (*C function*), 154
`n_factor_ecm_mul_montgomery_ladder` (*C function*), 154
`n_factor_ecm_select_curve` (*C function*), 154
`n_factor_ecm_stage_I` (*C function*), 154
`n_factor_ecm_stage_II` (*C function*), 154
`n_factor_evaluate` (*C function*), 149
`n_factor_init` (*C function*), 149
`n_factor_insert` (*C function*), 150
`n_factor_lehman` (*C function*), 150
`n_factor_one_line` (*C function*), 150
`n_factor_partial` (*C function*), 151
`n_factor_pollard_brent` (*C function*), 152
`n_factor_pollard_brent_single` (*C function*), 152
`n_factor_power235` (*C function*), 150
`n_factor_pp1` (*C function*), 152
`n_factor_pp1_wrapper` (*C function*), 152
`n_factor_SQUFOF` (*C function*), 151
`n_factor_t` (*C type*), 149
`n_factor_t.exp` (*C member*), 149
`n_factor_t.num` (*C member*), 149
`n_factor_t.p` (*C member*), 149
`n_factor_trial` (*C function*), 150
`n_factor_trial_partial` (*C function*), 151
`n_factor_trial_range` (*C function*), 150
`n_factorial_fast_mod2_preinv` (*C function*), 153
`n_factorial_mod2_preinv` (*C function*), 153
`n_flog` (*C function*), 137
`n_gcd` (*C function*), 140
`n_gcdinv` (*C function*), 140
`n_invmod` (*C function*), 141
`n_is_perfect_power` (*C function*), 149
`n_is_perfect_power235` (*C function*), 148
`n_is_prime` (*C function*), 146
`n_is_prime_odd_no_trial` (*C function*), 146
`n_is_prime_pocklington` (*C function*), 146
`n_is_prime_pseudosquare` (*C function*), 146
`n_is_probabprime` (*C function*), 148
`n_is_probabprime_BPSW` (*C function*), 147
`n_is_probabprime_fermat` (*C function*), 147
`n_is_probabprime_fibonacci` (*C function*), 147
`n_is_probabprime_lucas` (*C function*), 147
`n_is_square` (*C function*), 148
`n_is_squarefree` (*C function*), 153
`n_is_strong_probabprime2_preinv` (*C function*), 147
`n_is_strong_probabprime_precomp` (*C function*), 147
`n_jacobi` (*C function*), 141
`n_jacobi_unsigned` (*C function*), 141
`n_ll_mod_preinv` (*C function*), 139
`n_lll_mod_preinv` (*C function*), 139
`n_mod2_precomp` (*C function*), 138
`n_mod2_preinv` (*C function*), 139
`n_mod_precomp` (*C function*), 138
`n_moebius_mu` (*C function*), 153
`n_moebius_mu_vec` (*C function*), 153
`n_mulmod2` (*C function*), 140
`n_mulmod2_preinv` (*C function*), 139

n_mulmod_and_precomp_shoup (*C function*), 143
 n_mulmod_precomp (*C function*), 139
 n_mulmod_precomp_shoup (*C function*), 142
 n_mulmod_precomp_shoup_quo_rem (*C function*), 143
 n_mulmod_precomp_shoup_rem_from_quo (*C function*), 143
 n_mulmod_preinv (*C function*), 140
 n_mulmod_shoup (*C function*), 143
 n_nextprime (*C function*), 145
 n_nth_prime (*C function*), 145
 n_nth_prime_bounds (*C function*), 145
 n_pow (*C function*), 137
 n_powmod (*C function*), 141
 n_powmod2 (*C function*), 141
 n_powmod2_fmpz_preinv (*C function*), 142
 n_powmod2_preinv (*C function*), 141
 n_powmod2_ui_preinv (*C function*), 142
 n_powmod_precomp (*C function*), 141
 n_powmod_ui_precomp (*C function*), 141
 n_precompute_inverse (*C function*), 138
 n_preinvert_limb (*C function*), 137
 n_preinvert_limb_prenorm (*C function*), 137
 n_prime_inverses_arr_readonly (*C function*), 145
 n_prime_pi (*C function*), 145
 n_prime_pi_bounds (*C function*), 145
 n_primes_arr_readonly (*C function*), 145
 n_primes_clear (*C function*), 144
 n_primes_extend_small (*C function*), 144
 n_primes_init (*C function*), 144
 n_primes_jump_after (*C function*), 144
 n_primes_next (*C function*), 144
 n_primes_sieve_range (*C function*), 144
 n_primitive_root_prime (*C function*), 153
 n_primitive_root_prime_prefactor (*C function*), 153
 n_randbits (*C function*), 136
 n_randint (*C function*), 136
 n_randlimb (*C function*), 136
 n_randprime (*C function*), 136
 n_randtest (*C function*), 136
 n_randtest_bits (*C function*), 136
 n_randtest_not_zero (*C function*), 136
 n_randtest_prime (*C function*), 136
 n_remove (*C function*), 149
 n_remove2_precomp (*C function*), 150
 n_revbin (*C function*), 137
 n_root (*C function*), 149
 n_rootrem (*C function*), 149
 n_sizeinbase (*C function*), 137
 n_sqrt (*C function*), 148
 n_sqrtmod (*C function*), 142
 n_sqrtmod_2pow (*C function*), 142
 n_sqrtmod_primepow (*C function*), 142
 n_sqrtmodn (*C function*), 142
 n_sqrtrem (*C function*), 148
 n_submod (*C function*), 141
 n_urandint (*C function*), 136
 n_xgcd (*C function*), 140
 Neg (*C macro*), 900
 nf_clear (*C function*), 537
 nf_elem_add (*C function*), 539
 nf_elem_canonicalise (*C function*), 537
 nf_elem_clear (*C function*), 537
 nf_elem_coprime_den (*C function*), 541
 nf_elem_coprime_den_signed (*C function*), 541
 nf_elem_div (*C function*), 540
 nf_elem_equal (*C function*), 539
 nf_elem_get_den (*C function*), 539
 nf_elem_get_fmpq_poly (*C function*), 538
 nf_elem_get_fmpz_mat_row (*C function*), 538
 nf_elem_get_fmpz_mod_poly (*C function*), 538
 nf_elem_get_fmpz_mod_poly_den (*C function*), 538
 nf_elem_get_nmod_poly (*C function*), 538
 nf_elem_get_nmod_poly_den (*C function*), 538
 nf_elem_init (*C function*), 537
 nf_elem_inv (*C function*), 540
 nf_elem_is_one (*C function*), 539
 nf_elem_is_zero (*C function*), 539
 nf_elem_mod_fmpz (*C function*), 541
 nf_elem_mod_fmpz_den (*C function*), 541
 nf_elem_mul (*C function*), 540
 nf_elem_mul_gen (*C function*), 539
 nf_elem_mul_red (*C function*), 540
 nf_elem_neg (*C function*), 539
 nf_elem_norm (*C function*), 540
 nf_elem_norm_div (*C function*), 540
 nf_elem_one (*C function*), 539
 nf_elem_pow (*C function*), 540
 nf_elem_print_pretty (*C function*), 539
 nf_elem_randtest (*C function*), 537
 nf_elem_reduce (*C function*), 538
 nf_elem_rep_mat (*C function*), 541
 nf_elem_rep_mat_fmpz_mat_den (*C function*), 541
 nf_elem_set (*C function*), 539
 nf_elem_set_den (*C function*), 539
 nf_elem_set_fmpq_poly (*C function*), 538
 nf_elem_set_fmpz_mat_row (*C function*), 538
 nf_elem_smod_fmpz (*C function*), 541
 nf_elem_smod_fmpz_den (*C function*), 541
 nf_elem_struct (*C type*), 537
 nf_elem_sub (*C function*), 540
 nf_elem_swap (*C function*), 539
 nf_elem_t (*C type*), 537
 nf_elem_trace (*C function*), 540
 nf_elem_zero (*C function*), 539
 nf_init (*C function*), 537
 nf_struct (*C type*), 537
 nf_t (*C type*), 537
 nfloat1024_struct (*C type*), 591
 nfloat1024_t (*C type*), 591
 nfloat128_struct (*C type*), 591
 nfloat128_t (*C type*), 591

`nfloa192_struct` (*C type*), 591
`nfloa192_t` (*C type*), 591
`nfloa2048_struct` (*C type*), 591
`nfloa2048_t` (*C type*), 591
`nfloa256_struct` (*C type*), 591
`nfloa256_t` (*C type*), 591
`nfloa384_struct` (*C type*), 591
`nfloa384_t` (*C type*), 591
`nfloa4096_struct` (*C type*), 591
`nfloa4096_t` (*C type*), 591
`nfloa512_struct` (*C type*), 591
`nfloa512_t` (*C type*), 591
`nfloa64_struct` (*C type*), 591
`nfloa64_t` (*C type*), 591
`nfloa_abs` (*C function*), 594
`nfloa_add` (*C function*), 594
`nfloa_addmul` (*C function*), 594
`NFLOAT_ALLOW_INF` (*C macro*), 592
`NFLOAT_ALLOW_NAN` (*C macro*), 593
`NFLOAT_ALLOW_UNDERFLOW` (*C macro*), 592
`nfloa_atan` (*C function*), 595
`nfloa_ceil` (*C function*), 594
`nfloa_clear` (*C function*), 593
`nfloa_cmp` (*C function*), 594
`nfloa_cmpabs` (*C function*), 594
`nfloa_complex_abs` (*C function*), 596
`nfloa_complex_add` (*C function*), 596
`nfloa_complex_clear` (*C function*), 596
`nfloa_complex_cmp` (*C function*), 596
`nfloa_complex_cmpabs` (*C function*), 596
`nfloa_complex_conj` (*C function*), 596
`NFLOAT_COMPLEX_CTX_DATA_NLIMBS` (*C macro*), 596
`nfloa_complex_ctx_init` (*C function*), 596
`nfloa_complex_div` (*C function*), 596
`nfloa_complex_equal` (*C function*), 596
`nfloa_complex_get_acb` (*C function*), 596
`nfloa_complex_get_acf` (*C function*), 596
`nfloa_complex_i` (*C function*), 596
`nfloa_complex_im` (*C function*), 596
`NFLOAT_COMPLEX_IM` (*C macro*), 596
`nfloa_complex_init` (*C function*), 596
`nfloa_complex_inv` (*C function*), 596
`nfloa_complex_is_neg_one` (*C function*), 596
`nfloa_complex_is_one` (*C function*), 596
`NFLOAT_COMPLEX_IS_SPECIAL` (*C macro*), 596
`nfloa_complex_is_zero` (*C function*), 596
`NFLOAT_COMPLEX_IS_ZERO` (*C macro*), 596
`nfloa_complex_mat_lu` (*C function*), 596
`nfloa_complex_mat_mul` (*C function*), 596
`nfloa_complex_mat_mul_block` (*C function*), 596
`nfloa_complex_mat_mul_fixed` (*C function*), 596
`nfloa_complex_mat_mul_reorder` (*C function*), 596
`nfloa_complex_mat_nonsingular_solve_tril` (*C function*), 596
`nfloa_complex_mat_nonsingular_solve_triu` (*C function*), 596
`nfloa_complex_mul` (*C function*), 596
`nfloa_complex_mul_2exp_si` (*C function*), 596
`nfloa_complex_neg` (*C function*), 596
`nfloa_complex_neg_one` (*C function*), 596
`nfloa_complex_one` (*C function*), 596
`nfloa_complex_pi` (*C function*), 596
`nfloa_complex_ptr` (*C type*), 596
`nfloa_complex_randtest` (*C function*), 596
`nfloa_complex_re` (*C function*), 596
`NFLOAT_COMPLEX_RE` (*C macro*), 596
`nfloa_complex_set` (*C function*), 596
`nfloa_complex_set_acb` (*C function*), 596
`nfloa_complex_set_acf` (*C function*), 596
`nfloa_complex_set_d` (*C function*), 596
`nfloa_complex_set_fmpq` (*C function*), 596
`nfloa_complex_set_fmpz` (*C function*), 596
`nfloa_complex_set_other` (*C function*), 596
`nfloa_complex_set_si` (*C function*), 596
`nfloa_complex_set_ui` (*C function*), 596
`nfloa_complex_sqr` (*C function*), 596
`nfloa_complex_srcptr` (*C type*), 596
`nfloa_complex_sub` (*C function*), 596
`nfloa_complex_swap` (*C function*), 596
`nfloa_complex_write` (*C function*), 596
`nfloa_complex_zero` (*C function*), 596
`nfloa_cos` (*C function*), 595
`nfloa_cosh` (*C function*), 595
`nfloa_ctx_get_real_prec` (*C function*), 593
`nfloa_ctx_init` (*C function*), 592
`nfloa_ctx_set_real_prec` (*C function*), 593
`nfloa_ctx_write` (*C function*), 593
`NFLOAT_D` (*C macro*), 592
`NFLOAT_DATA` (*C macro*), 592
`nfloa_div` (*C function*), 594
`nfloa_div_si` (*C function*), 594
`nfloa_div_ui` (*C function*), 594
`nfloa_equal` (*C function*), 593
`nfloa_exp` (*C function*), 595
`NFLOAT_EXP` (*C macro*), 592
`NFLOAT_EXP_NAN` (*C macro*), 592
`NFLOAT_EXP_NEG_INF` (*C macro*), 592
`NFLOAT_EXP_POS_INF` (*C macro*), 592
`NFLOAT_EXP_ZERO` (*C macro*), 592
`nfloa_expm1` (*C function*), 595
`nfloa_floor` (*C function*), 594
`nfloa_gamma` (*C function*), 595
`nfloa_get_arf` (*C function*), 594
`nfloa_get_d_2exp_si` (*C function*), 594
`nfloa_get_fmpz` (*C function*), 594
`NFLOAT_HEADER_LIMBS` (*C macro*), 592
`nfloa_im` (*C function*), 594
`nfloa_init` (*C function*), 593
`nfloa_inv` (*C function*), 594
`NFLOAT_IS_INF` (*C macro*), 592
`NFLOAT_IS_NAN` (*C macro*), 592
`NFLOAT_IS_NEG_INF` (*C macro*), 592

- `nfloat_is_neg_one` (*C function*), 593
- `nfloat_is_one` (*C function*), 593
- `NFLOAT_IS_POS_INF` (*C macro*), 592
- `NFLOAT_IS_SPECIAL` (*C macro*), 592
- `nfloat_is_zero` (*C function*), 593
- `NFLOAT_IS_ZERO` (*C macro*), 592
- `nfloat_log` (*C function*), 595
- `nfloat_log1p` (*C function*), 595
- `nfloat_mat_lq` (*C function*), 595
- `nfloat_mat_lu` (*C function*), 595
- `nfloat_mat_mul` (*C function*), 595
- `nfloat_mat_mul_block` (*C function*), 595
- `nfloat_mat_mul_fixed` (*C function*), 595
- `nfloat_mat_nonsingular_solve_tril` (*C function*), 595
- `nfloat_mat_nonsingular_solve_triu` (*C function*), 595
- `NFLOAT_MAX_ALLOC` (*C macro*), 592
- `NFLOAT_MAX_EXP` (*C macro*), 592
- `NFLOAT_MAX_LIMBS` (*C macro*), 591
- `NFLOAT_MIN_EXP` (*C macro*), 592
- `NFLOAT_MIN_LIMBS` (*C macro*), 591
- `nfloat_mul` (*C function*), 594
- `nfloat_mul_2exp_si` (*C function*), 594
- `nfloat_nan` (*C function*), 593
- `nfloat_neg` (*C function*), 594
- `nfloat_neg_inf` (*C function*), 593
- `nfloat_neg_one` (*C function*), 593
- `nfloat_nint` (*C function*), 594
- `nfloat_one` (*C function*), 593
- `nfloat_pi` (*C function*), 595
- `nfloat_pos_inf` (*C function*), 593
- `nfloat_pow` (*C function*), 594
- `nfloat_ptr` (*C type*), 591
- `nfloat_randtest` (*C function*), 594
- `NFLOAT_RND_CEIL` (*C macro*), 593
- `NFLOAT_RND_FLOOR` (*C macro*), 593
- `nfloat_rsqrt` (*C function*), 594
- `nfloat_set` (*C function*), 593
- `nfloat_set_arf` (*C function*), 594
- `nfloat_set_d` (*C function*), 594
- `nfloat_set_fmpq` (*C function*), 594
- `nfloat_set_fmpz` (*C function*), 593
- `nfloat_set_mpn_2exp` (*C function*), 594
- `nfloat_set_other` (*C function*), 594
- `nfloat_set_si` (*C function*), 593
- `nfloat_set_str` (*C function*), 594
- `nfloat_set_ui` (*C function*), 593
- `nfloat_sgn` (*C function*), 594
- `NFLOAT_SGNBIT` (*C macro*), 592
- `nfloat_sin` (*C function*), 595
- `nfloat_sinh` (*C function*), 595
- `nfloat_sqr` (*C function*), 594
- `nfloat_sqrt` (*C function*), 594
- `nfloat_srcptr` (*C type*), 591
- `nfloat_sub` (*C function*), 594
- `nfloat_submul` (*C function*), 594
- `nfloat_swap` (*C function*), 593
- `nfloat_tan` (*C function*), 595
- `nfloat_tanh` (*C function*), 595
- `nfloat_trunc` (*C function*), 594
- `nfloat_write` (*C function*), 594
- `nfloat_zero` (*C function*), 593
- `nfloat_zeta` (*C function*), 595
- `NMOD2_RED2` (*C macro*), 382
- `nmod_2_pow_ui` (*C function*), 383
- `nmod_add` (*C function*), 382
- `NMOD_ADDMUL` (*C macro*), 382
- `nmod_berlekamp_massey_add_point` (*C function*), 433
- `nmod_berlekamp_massey_add_points` (*C function*), 433
- `nmod_berlekamp_massey_add_zeros` (*C function*), 433
- `nmod_berlekamp_massey_clear` (*C function*), 432
- `nmod_berlekamp_massey_init` (*C function*), 432
- `nmod_berlekamp_massey_point_count` (*C function*), 433
- `nmod_berlekamp_massey_points` (*C function*), 433
- `nmod_berlekamp_massey_R_poly` (*C function*), 433
- `nmod_berlekamp_massey_reduce` (*C function*), 433
- `nmod_berlekamp_massey_set_prime` (*C function*), 433
- `nmod_berlekamp_massey_start_over` (*C function*), 432
- `nmod_berlekamp_massey_V_poly` (*C function*), 433
- `NMOD_BITS` (*C macro*), 382
- `NMOD_CAN_USE_SHOUP` (*C macro*), 382
- `nmod_discrete_log_pohlig_hellman_clear` (*C function*), 385
- `nmod_discrete_log_pohlig_hellman_init` (*C function*), 385
- `nmod_discrete_log_pohlig_hellman_precompute_prime` (*C function*), 385
- `nmod_discrete_log_pohlig_hellman_primitive_root` (*C function*), 385
- `nmod_discrete_log_pohlig_hellman_run` (*C function*), 385
- `nmod_div` (*C function*), 383
- `nmod_divides` (*C function*), 383
- `nmod_geometric_progression_clear` (*C function*), 431
- `nmod_geometric_progression_init` (*C function*), 431
- `nmod_init` (*C function*), 382
- `nmod_inv` (*C function*), 383
- `nmod_mat_add` (*C function*), 392
- `nmod_mat_addmul` (*C function*), 393
- `nmod_mat_can_solve` (*C function*), 395
- `nmod_mat_can_solve_inner` (*C function*), 395
- `nmod_mat_charpoly` (*C function*), 397
- `nmod_mat_charpoly_berkowitz` (*C function*), 397

`nmod_mat_charpoly_danilevsky` (*C function*), 397

`nmod_mat_clear` (*C function*), 389

`nmod_mat_concat_horizontal` (*C function*), 390

`nmod_mat_concat_vertical` (*C function*), 390

`nmod_mat_det` (*C function*), 394

`nmod_mat_det_howell` (*C function*), 394

`nmod_mat_entry` (*C macro*), 390

`nmod_mat_entry_ptr` (*C function*), 390

`nmod_mat_equal` (*C function*), 392

`nmod_mat_fprint` (*C function*), 391

`nmod_mat_fprint_pretty` (*C function*), 391

`nmod_mat_get_entry` (*C function*), 390

`nmod_mat_howell_form` (*C function*), 397

`nmod_mat_init` (*C function*), 389

`nmod_mat_init_set` (*C function*), 389

`nmod_mat_inv` (*C function*), 394

`nmod_mat_invert_cols` (*C function*), 392

`nmod_mat_invert_rows` (*C function*), 392

`nmod_mat_is_zero` (*C function*), 390

`nmod_mat_is_zero_row` (*C function*), 392

`nmod_mat_lu` (*C function*), 396

`nmod_mat_lu_classical` (*C function*), 396

`nmod_mat_lu_classical_delayed` (*C function*), 396

`nmod_mat_lu_recursive` (*C function*), 396

`nmod_mat_minpoly` (*C function*), 397

`nmod_mat_mod` (*C function*), 389

`nmod_mat_mul` (*C function*), 393

`nmod_mat_mul_blas` (*C function*), 393

`nmod_mat_mul_classical` (*C function*), 393

`nmod_mat_mul_classical_threaded` (*C function*), 393

`nmod_mat_mul_nmod_vec` (*C function*), 393

`nmod_mat_mul_nmod_vec_ptr` (*C function*), 393

`nmod_mat_mul_strassen` (*C function*), 393

`nmod_mat_ncols` (*C function*), 390

`nmod_mat_neg` (*C function*), 392

`nmod_mat_nmod_vec_mul` (*C function*), 393

`nmod_mat_nmod_vec_mul_ptr` (*C function*), 393

`nmod_mat_nrows` (*C function*), 390

`nmod_mat_nullspace` (*C function*), 397

`nmod_mat_permute_rows` (*C function*), 392

`nmod_mat_pow` (*C function*), 394

`nmod_mat_print` (*C function*), 391

`nmod_mat_print_pretty` (*C function*), 391

`nmod_mat_rand` (*C function*), 391

`nmod_mat_randfull` (*C function*), 391

`nmod_mat_randops` (*C function*), 391

`nmod_mat_randpermdiag` (*C function*), 391

`nmod_mat_randrank` (*C function*), 391

`nmod_mat_randtest` (*C function*), 391

`nmod_mat_randtril` (*C function*), 391

`nmod_mat_randtriu` (*C function*), 391

`nmod_mat_rank` (*C function*), 394

`nmod_mat_reduce_row` (*C function*), 396

`nmod_mat_rref` (*C function*), 396

`nmod_mat_scalar_addmul_ui` (*C function*), 392

`nmod_mat_scalar_mul` (*C function*), 392

`nmod_mat_scalar_mul_fmpz` (*C function*), 392

`nmod_mat_set` (*C function*), 389

`nmod_mat_set_entry` (*C function*), 390

`nmod_mat_set_mod` (*C function*), 389

`nmod_mat_similarity` (*C function*), 397

`nmod_mat_solve` (*C function*), 395

`nmod_mat_solve_tril` (*C function*), 394

`nmod_mat_solve_tril_classical` (*C function*), 394

`nmod_mat_solve_tril_recursive` (*C function*), 395

`nmod_mat_solve_triu` (*C function*), 395

`nmod_mat_solve_triu_classical` (*C function*), 395

`nmod_mat_solve_triu_recursive` (*C function*), 395

`nmod_mat_solve_vec` (*C function*), 396

`nmod_mat_strong_echelon_form` (*C function*), 397

`nmod_mat_struct` (*C type*), 389

`nmod_mat_sub` (*C function*), 392

`nmod_mat_submul` (*C function*), 393

`nmod_mat_swap` (*C function*), 389

`nmod_mat_swap_cols` (*C function*), 392

`nmod_mat_swap_entrywise` (*C function*), 389

`nmod_mat_swap_rows` (*C function*), 392

`nmod_mat_t` (*C type*), 389

`nmod_mat_trace` (*C function*), 394

`nmod_mat_transpose` (*C function*), 392

`nmod_mat_window_clear` (*C function*), 390

`nmod_mat_window_init` (*C function*), 390

`nmod_mat_zero` (*C function*), 390

`nmod_mpoly_add` (*C function*), 448

`nmod_mpoly_add_ui` (*C function*), 448

`nmod_mpoly_clear` (*C function*), 444

`nmod_mpoly_cmp` (*C function*), 446

`nmod_mpoly_combine_like_terms` (*C function*), 447

`nmod_mpoly_compose_nmod_mpoly` (*C function*), 449

`nmod_mpoly_compose_nmod_mpoly_gen` (*C function*), 449

`nmod_mpoly_compose_nmod_mpoly_geobucket` (*C function*), 449

`nmod_mpoly_compose_nmod_mpoly_horner` (*C function*), 449

`nmod_mpoly_compose_nmod_poly` (*C function*), 449

`nmod_mpoly_content_vars` (*C function*), 451

`nmod_mpoly_ctx_clear` (*C function*), 443

`nmod_mpoly_ctx_init` (*C function*), 443

`nmod_mpoly_ctx_modulus` (*C function*), 443

`nmod_mpoly_ctx_nvars` (*C function*), 443

`nmod_mpoly_ctx_ord` (*C function*), 443

`nmod_mpoly_ctx_struct` (*C type*), 443

`nmod_mpoly_ctx_t` (*C type*), 443

`nmod_mpoly_degree_fmpz` (*C function*), 445

`nmod_mpoly_degree_si` (*C function*), 445
`nmod_mpoly_degrees_fit_si` (*C function*), 445
`nmod_mpoly_degrees_fmpz` (*C function*), 445
`nmod_mpoly_degrees_si` (*C function*), 445
`nmod_mpoly_derivative` (*C function*), 449
`nmod_mpoly_discriminant` (*C function*), 452
`nmod_mpoly_div` (*C function*), 450
`nmod_mpoly_div_monagan_pearce` (*C function*), 453
`nmod_mpoly_divides` (*C function*), 450
`nmod_mpoly_divides_dense` (*C function*), 451
`nmod_mpoly_divides_heap_threaded` (*C function*), 451
`nmod_mpoly_divides_monagan_pearce` (*C function*), 451
`nmod_mpoly_divrem` (*C function*), 450
`nmod_mpoly_divrem_ideal` (*C function*), 450
`nmod_mpoly_divrem_ideal_monagan_pearce` (*C function*), 453
`nmod_mpoly_divrem_monagan_pearce` (*C function*), 453
`nmod_mpoly_equal` (*C function*), 444
`nmod_mpoly_equal_ui` (*C function*), 445
`nmod_mpoly_evaluate_all_ui` (*C function*), 449
`nmod_mpoly_evaluate_one_ui` (*C function*), 449
`nmod_mpoly_factor` (*C function*), 454
`nmod_mpoly_factor_clear` (*C function*), 454
`nmod_mpoly_factor_get_base` (*C function*), 454
`nmod_mpoly_factor_get_constant_ui` (*C function*), 454
`nmod_mpoly_factor_get_exp_si` (*C function*), 454
`nmod_mpoly_factor_init` (*C function*), 454
`nmod_mpoly_factor_length` (*C function*), 454
`nmod_mpoly_factor_sort` (*C function*), 454
`nmod_mpoly_factor_squarefree` (*C function*), 454
`nmod_mpoly_factor_struct` (*C type*), 453
`nmod_mpoly_factor_swap` (*C function*), 454
`nmod_mpoly_factor_swap_base` (*C function*), 454
`nmod_mpoly_factor_t` (*C type*), 453
`nmod_mpoly_fit_length` (*C function*), 443
`nmod_mpoly_fprint_pretty` (*C function*), 444
`nmod_mpoly_from_univar` (*C function*), 452
`nmod_mpoly_gcd` (*C function*), 451
`nmod_mpoly_gcd_brown` (*C function*), 451
`nmod_mpoly_gcd_cofactors` (*C function*), 451
`nmod_mpoly_gcd_hensel` (*C function*), 451
`nmod_mpoly_gcd_zippel` (*C function*), 451
`nmod_mpoly_gen` (*C function*), 444
`nmod_mpoly_get_coeff_ui_fmpz` (*C function*), 446
`nmod_mpoly_get_coeff_ui_monomial` (*C function*), 446
`nmod_mpoly_get_coeff_ui_ui` (*C function*), 446
`nmod_mpoly_get_coeff_vars_ui` (*C function*), 446
`nmod_mpoly_get_str_pretty` (*C function*), 444
`nmod_mpoly_get_term` (*C function*), 447
`nmod_mpoly_get_term_coeff_ui` (*C function*), 446
`nmod_mpoly_get_term_exp_fmpz` (*C function*), 447
`nmod_mpoly_get_term_exp_si` (*C function*), 447
`nmod_mpoly_get_term_exp_ui` (*C function*), 447
`nmod_mpoly_get_term_monomial` (*C function*), 447
`nmod_mpoly_get_term_var_exp_si` (*C function*), 447
`nmod_mpoly_get_term_var_exp_ui` (*C function*), 447
`nmod_mpoly_get_ui` (*C function*), 445
`nmod_mpoly_init` (*C function*), 443
`nmod_mpoly_init2` (*C function*), 443
`nmod_mpoly_init3` (*C function*), 443
`nmod_mpoly_is_canonical` (*C function*), 446
`nmod_mpoly_is_gen` (*C function*), 444
`nmod_mpoly_is_one` (*C function*), 445
`nmod_mpoly_is_square` (*C function*), 452
`nmod_mpoly_is_ui` (*C function*), 445
`nmod_mpoly_is_zero` (*C function*), 445
`nmod_mpoly_length` (*C function*), 446
`nmod_mpoly_make_monic` (*C function*), 448
`nmod_mpoly_mul` (*C function*), 450
`nmod_mpoly_mul_array` (*C function*), 450
`nmod_mpoly_mul_array_threaded` (*C function*), 450
`nmod_mpoly_mul_dense` (*C function*), 450
`nmod_mpoly_mul_heap_threaded` (*C function*), 450
`nmod_mpoly_mul_johnson` (*C function*), 450
`nmod_mpoly_neg` (*C function*), 448
`nmod_mpoly_one` (*C function*), 445
`nmod_mpoly_pow_fmpz` (*C function*), 450
`nmod_mpoly_pow_rmul` (*C function*), 453
`nmod_mpoly_pow_ui` (*C function*), 450
`nmod_mpoly_print_pretty` (*C function*), 444
`nmod_mpoly_push_term_ui_ffmpz` (*C function*), 447
`nmod_mpoly_push_term_ui_fmpz` (*C function*), 447
`nmod_mpoly_push_term_ui_ui` (*C function*), 447
`nmod_mpoly_quadratic_root` (*C function*), 452
`nmod_mpoly_randtest_bits` (*C function*), 448
`nmod_mpoly_randtest_bound` (*C function*), 448
`nmod_mpoly_randtest_bounds` (*C function*), 448
`nmod_mpoly_realloc` (*C function*), 443
`nmod_mpoly_resize` (*C function*), 446
`nmod_mpoly_resultant` (*C function*), 452
`nmod_mpoly_reverse` (*C function*), 448
`nmod_mpoly_scalar_mul_ui` (*C function*), 448
`nmod_mpoly_set` (*C function*), 444
`nmod_mpoly_set_coeff_ui_fmpz` (*C function*), 446
`nmod_mpoly_set_coeff_ui_monomial` (*C function*), 446

`nmod_mpoly_set_coeff_ui_ui` (*C function*), 446
`nmod_mpoly_set_str_pretty` (*C function*), 444
`nmod_mpoly_set_term_coeff_ui` (*C function*), 447
`nmod_mpoly_set_term_exp_fmpz` (*C function*), 447
`nmod_mpoly_set_term_exp_ui` (*C function*), 447
`nmod_mpoly_set_ui` (*C function*), 445
`nmod_mpoly_sort_terms` (*C function*), 447
`nmod_mpoly_sqrt` (*C function*), 452
`nmod_mpoly_struct` (*C type*), 443
`nmod_mpoly_sub` (*C function*), 448
`nmod_mpoly_sub_ui` (*C function*), 448
`nmod_mpoly_swap` (*C function*), 444
`nmod_mpoly_t` (*C type*), 443
`nmod_mpoly_term_coeff_ref` (*C function*), 446
`nmod_mpoly_term_content` (*C function*), 451
`nmod_mpoly_term_exp_fits_si` (*C function*), 447
`nmod_mpoly_term_exp_fits_ui` (*C function*), 447
`nmod_mpoly_to_univar` (*C function*), 452
`nmod_mpoly_total_degree_fits_si` (*C function*), 445
`nmod_mpoly_total_degree_fmpz` (*C function*), 445
`nmod_mpoly_total_degree_si` (*C function*), 445
`nmod_mpoly_univar_clear` (*C function*), 452
`nmod_mpoly_univar_degree_fits_si` (*C function*), 452
`nmod_mpoly_univar_get_term_coeff` (*C function*), 453
`nmod_mpoly_univar_get_term_exp_si` (*C function*), 453
`nmod_mpoly_univar_init` (*C function*), 452
`nmod_mpoly_univar_length` (*C function*), 452
`nmod_mpoly_univar_swap` (*C function*), 452
`nmod_mpoly_univar_swap_term_coeff` (*C function*), 453
`nmod_mpoly_used_vars` (*C function*), 445
`nmod_mpoly_zero` (*C function*), 445
`nmod_mul` (*C function*), 383
`NMOD_MUL_FULLWORD` (*C macro*), 382
`NMOD_MUL_PRENORM` (*C macro*), 382
`nmod_neg` (*C function*), 382
`nmod_poly_add` (*C function*), 403
`nmod_poly_add_series` (*C function*), 403
`nmod_poly_asin_series` (*C function*), 429
`nmod_poly_asinh_series` (*C function*), 429
`nmod_poly_atan_series` (*C function*), 428
`nmod_poly_atanh_series` (*C function*), 428
`nmod_poly_bit_pack` (*C function*), 404
`nmod_poly_bit_unpack` (*C function*), 404
`nmod_poly_clear` (*C function*), 399
`nmod_poly_compose` (*C function*), 418
`nmod_poly_compose_divconquer` (*C function*), 418
`nmod_poly_compose_horner` (*C function*), 418
`nmod_poly_compose_mod` (*C function*), 422
`nmod_poly_compose_mod_brent_kung` (*C function*), 419
`nmod_poly_compose_mod_brent_kung_precomp_preinv` (*C function*), 420
`nmod_poly_compose_mod_brent_kung_preinv` (*C function*), 420
`nmod_poly_compose_mod_brent_kung_vec_preinv` (*C function*), 421
`nmod_poly_compose_mod_brent_kung_vec_preinv_threaded` (*C function*), 421
`nmod_poly_compose_mod_brent_kung_vec_preinv_threaded_pool` (*C function*), 421
`nmod_poly_compose_mod_horner` (*C function*), 419
`nmod_poly_compose_series` (*C function*), 426
`nmod_poly_cos_series` (*C function*), 429
`nmod_poly_cosh_series` (*C function*), 429
`nmod_poly_deflate` (*C function*), 431
`nmod_poly_deflation` (*C function*), 431
`nmod_poly_degree` (*C function*), 399
`nmod_poly_derivative` (*C function*), 414
`nmod_poly_discriminant` (*C function*), 426
`nmod_poly_div` (*C function*), 411
`nmod_poly_div_newton_n_preinv` (*C function*), 413
`nmod_poly_div_root` (*C function*), 413
`nmod_poly_div_series` (*C function*), 413
`nmod_poly_div_series_basecase` (*C function*), 412
`nmod_poly_divexact` (*C function*), 412
`nmod_poly_divides` (*C function*), 414
`nmod_poly_divides_classical` (*C function*), 414
`nmod_poly_divrem` (*C function*), 411
`nmod_poly_divrem_basecase` (*C function*), 411
`nmod_poly_divrem_newton_n_preinv` (*C function*), 413
`nmod_poly_divrem_precomp_struct` (*C type*), 307
`nmod_poly_equal` (*C function*), 402
`nmod_poly_equal_nmod` (*C function*), 402
`nmod_poly_equal_trunc` (*C function*), 402
`nmod_poly_equal_ui` (*C function*), 402
`nmod_poly_evaluate_geometric_nmod_vec_fast` (*C function*), 416
`nmod_poly_evaluate_geometric_nmod_vec_iter` (*C function*), 416
`nmod_poly_evaluate_mat` (*C function*), 415
`nmod_poly_evaluate_mat_horner` (*C function*), 415
`nmod_poly_evaluate_mat_paterson_stockmeyer` (*C function*), 415
`nmod_poly_evaluate_nmod` (*C function*), 415
`nmod_poly_evaluate_nmod_vec` (*C function*), 416
`nmod_poly_evaluate_nmod_vec_fast` (*C function*), 415
`nmod_poly_evaluate_nmod_vec_iter` (*C function*), 415
`nmod_poly_exp_series` (*C function*), 428
`nmod_poly_factor` (*C function*), 442

`nmod_poly_factor_berlekamp` (*C function*), 442
`nmod_poly_factor_cantor_zassenhaus` (*C function*), 441
`nmod_poly_factor_clear` (*C function*), 440
`nmod_poly_factor_concat` (*C function*), 441
`nmod_poly_factor_distinct_deg` (*C function*), 441
`nmod_poly_factor_distinct_deg_threaded` (*C function*), 441
`nmod_poly_factor_equal_deg` (*C function*), 441
`nmod_poly_factor_equal_deg_prob` (*C function*), 441
`nmod_poly_factor_fit_length` (*C function*), 440
`nmod_poly_factor_init` (*C function*), 440
`nmod_poly_factor_insert` (*C function*), 440
`nmod_poly_factor_kaltofen_shoup` (*C function*), 442
`nmod_poly_factor_pow` (*C function*), 441
`nmod_poly_factor_print` (*C function*), 440
`nmod_poly_factor_realloc` (*C function*), 440
`nmod_poly_factor_set` (*C function*), 440
`nmod_poly_factor_squarefree` (*C function*), 441
`nmod_poly_factor_struct` (*C type*), 440
`nmod_poly_factor_t` (*C type*), 440
`nmod_poly_factor_with_berlekamp` (*C function*), 442
`nmod_poly_factor_with_cantor_zassenhaus` (*C function*), 442
`nmod_poly_factor_with_kaltofen_shoup` (*C function*), 442
`nmod_poly_find_distinct_nonzero_roots` (*C function*), 430
`nmod_poly_fit_length` (*C function*), 399
`nmod_poly_fprint` (*C function*), 402
`nmod_poly_fprint_pretty` (*C function*), 402
`nmod_poly_fread` (*C function*), 402
`nmod_poly_gcd` (*C function*), 423
`nmod_poly_gcd_euclidean` (*C function*), 422
`nmod_poly_gcd_euclidean_redc_half` (*C function*), 422
`nmod_poly_gcd_hgcd` (*C function*), 422
`nmod_poly_gcdinv` (*C function*), 425
`nmod_poly_get_coeff_ui` (*C function*), 401
`nmod_poly_get_str` (*C function*), 401
`nmod_poly_get_str_pretty` (*C function*), 401
`nmod_poly_inflate` (*C function*), 431
`nmod_poly_init` (*C function*), 399
`nmod_poly_init2` (*C function*), 399
`nmod_poly_init2_preinv` (*C function*), 399
`nmod_poly_init_mod` (*C function*), 399
`nmod_poly_init_preinv` (*C function*), 399
`nmod_poly_integral` (*C function*), 414
`nmod_poly_interpolate_geometric_nmod_vec_fast` (*C function*), 418
`nmod_poly_interpolate_geometric_nmod_vec_fast_hybrid` (*C function*), 417
`nmod_poly_interpolate_nmod_vec` (*C function*), 417
`nmod_poly_interpolate_nmod_vec_barycentric` (*C function*), 417
`nmod_poly_interpolate_nmod_vec_fast` (*C function*), 417
`nmod_poly_interpolate_nmod_vec_newton` (*C function*), 417
`nmod_poly_inv_series` (*C function*), 412
`nmod_poly_inv_series_basecase` (*C function*), 412
`nmod_poly_inv_series_newton` (*C function*), 412
`nmod_poly_invmod` (*C function*), 425
`nmod_poly_invsqrt_series` (*C function*), 426
`nmod_poly_is_gen` (*C function*), 403
`nmod_poly_is_irreducible` (*C function*), 441
`nmod_poly_is_irreducible_ddf` (*C function*), 441
`nmod_poly_is_irreducible_rabin` (*C function*), 441
`nmod_poly_is_monic` (*C function*), 399
`nmod_poly_is_one` (*C function*), 402
`nmod_poly_is_squarefree` (*C function*), 441
`nmod_poly_is_unit` (*C function*), 399
`nmod_poly_is_zero` (*C function*), 402
`nmod_poly_length` (*C function*), 399
`nmod_poly_log_series` (*C function*), 428
`nmod_poly_make_monic` (*C function*), 404
`nmod_poly_mat_add` (*C function*), 437
`nmod_poly_mat_clear` (*C function*), 434
`nmod_poly_mat_concat_horizontal` (*C function*), 435
`nmod_poly_mat_concat_vertical` (*C function*), 435
`nmod_poly_mat_degree` (*C function*), 437
`nmod_poly_mat_det` (*C function*), 439
`nmod_poly_mat_det_fflu` (*C function*), 439
`nmod_poly_mat_det_interpolate` (*C function*), 439
`nmod_poly_mat_entry` (*C function*), 434
`nmod_poly_mat_equal` (*C function*), 436
`nmod_poly_mat_equal_nmod_mat` (*C function*), 436
`nmod_poly_mat_evaluate_nmod` (*C function*), 437
`nmod_poly_mat_fflu` (*C function*), 438
`nmod_poly_mat_find_pivot_any` (*C function*), 438
`nmod_poly_mat_find_pivot_partial` (*C function*), 438
`nmod_poly_mat_get_coeff_mat` (*C function*), 436
`nmod_poly_mat_init` (*C function*), 434
`nmod_poly_mat_init_set` (*C function*), 434
`nmod_poly_mat_inv` (*C function*), 439
`nmod_poly_mat_is_empty` (*C function*), 436
`nmod_poly_mat_is_one` (*C function*), 436
`nmod_poly_mat_is_square` (*C function*), 436
`nmod_poly_mat_is_zero` (*C function*), 436
`nmod_poly_mat_max_length` (*C function*), 437
`nmod_poly_mat_modulus` (*C function*), 434
`nmod_poly_mat_mul` (*C function*), 437

`nmod_poly_mat_mul_classical` (*C function*), 437
`nmod_poly_mat_mul_interpolate` (*C function*), 437
`nmod_poly_mat_mul_KS` (*C function*), 437
`nmod_poly_mat_ncols` (*C function*), 434
`nmod_poly_mat_neg` (*C function*), 437
`nmod_poly_mat_nrows` (*C function*), 434
`nmod_poly_mat_nullspace` (*C function*), 439
`nmod_poly_mat_one` (*C function*), 436
`nmod_poly_mat_pow` (*C function*), 438
`nmod_poly_mat_print` (*C function*), 435
`nmod_poly_mat_randtest` (*C function*), 435
`nmod_poly_mat_randtest_sparse` (*C function*), 435
`nmod_poly_mat_rank` (*C function*), 439
`nmod_poly_mat_rref` (*C function*), 438
`nmod_poly_mat_scalar_mul_nmod` (*C function*), 437
`nmod_poly_mat_scalar_mul_nmod_poly` (*C function*), 437
`nmod_poly_mat_set` (*C function*), 434
`nmod_poly_mat_set_coeff_mat` (*C function*), 436
`nmod_poly_mat_set_nmod_mat` (*C function*), 434
`nmod_poly_mat_set_trunc` (*C function*), 434
`nmod_poly_mat_shift_left` (*C function*), 434
`nmod_poly_mat_shift_right` (*C function*), 434
`nmod_poly_mat_solve` (*C function*), 440
`nmod_poly_mat_solve_fflu` (*C function*), 440
`nmod_poly_mat_solve_fflu_precomp` (*C function*), 440
`nmod_poly_mat_sqr` (*C function*), 437
`nmod_poly_mat_sqr_classical` (*C function*), 438
`nmod_poly_mat_sqr_interpolate` (*C function*), 438
`nmod_poly_mat_sqr_KS` (*C function*), 438
`nmod_poly_mat_struct` (*C type*), 434
`nmod_poly_mat_sub` (*C function*), 437
`nmod_poly_mat_swap` (*C function*), 435
`nmod_poly_mat_swap_entrywise` (*C function*), 435
`nmod_poly_mat_t` (*C type*), 434
`nmod_poly_mat_trace` (*C function*), 439
`nmod_poly_mat_transpose` (*C function*), 436
`nmod_poly_mat_truncate` (*C function*), 434
`nmod_poly_mat_window_clear` (*C function*), 435
`nmod_poly_mat_window_init` (*C function*), 435
`nmod_poly_mat_zero` (*C function*), 436
`nmod_poly_max_bits` (*C function*), 399
`nmod_poly_minimal_irreducible` (*C function*), 400
`nmod_poly_modulus` (*C function*), 399
`nmod_poly_mul` (*C function*), 406
`nmod_poly_mul_classical` (*C function*), 405
`nmod_poly_mul_KS` (*C function*), 406
`nmod_poly_mul_KS2` (*C function*), 406
`nmod_poly_mul_KS4` (*C function*), 406
`nmod_poly_mulhigh` (*C function*), 407
`nmod_poly_mulhigh_classical` (*C function*), 405
`nmod_poly_mulmod` (*C function*), 406
`nmod_poly_mulmod_classical` (*C function*), 405
`nmod_poly_mulmod_KS` (*C function*), 406
`nmod_poly_mulmod_precond` (*C function*), 407
`nmod_poly_mulmod_precond_clear` (*C function*), 408
`nmod_poly_mulmod_precond_init_method` (*C function*), 408
`nmod_poly_mulmod_precond_init_num` (*C function*), 408
`nmod_poly_mulmod_precond_struct` (*C type*), 408
`nmod_poly_mulmod_precond_t` (*C type*), 408
`nmod_poly_mulmod_preinv` (*C function*), 407
`nmod_poly_multi_crt` (*C function*), 432
`nmod_poly_multi_crt_clear` (*C function*), 432
`nmod_poly_multi_crt_init` (*C function*), 431
`nmod_poly_multi_crt_precomp` (*C function*), 431
`nmod_poly_multi_crt_precomp_p` (*C function*), 431
`nmod_poly_multi_crt_precompute` (*C function*), 431
`nmod_poly_multi_crt_precompute_p` (*C function*), 431
`nmod_poly_neg` (*C function*), 403
`nmod_poly_pow` (*C function*), 409
`nmod_poly_pow_binexp` (*C function*), 409
`nmod_poly_pow_trunc` (*C function*), 409
`nmod_poly_pow_trunc_binexp` (*C function*), 409
`nmod_poly_power_sums` (*C function*), 427
`nmod_poly_power_sums_naive` (*C function*), 427
`nmod_poly_power_sums_schoenhage` (*C function*), 427
`nmod_poly_power_sums_to_poly` (*C function*), 427
`nmod_poly_power_sums_to_poly_naive` (*C function*), 427
`nmod_poly_power_sums_to_poly_schoenhage` (*C function*), 427
`nmod_poly_powers_mod_bsgs` (*C function*), 411
`nmod_poly_powers_mod_naive` (*C function*), 411
`nmod_poly_powmod_fmpz_binexp` (*C function*), 409
`nmod_poly_powmod_fmpz_binexp_preinv` (*C function*), 410
`nmod_poly_powmod_ui_binexp` (*C function*), 409
`nmod_poly_powmod_ui_binexp_preinv` (*C function*), 410
`nmod_poly_powmod_x_fmpz_preinv` (*C function*), 410
`nmod_poly_powmod_x_ui_preinv` (*C function*), 410
`nmod_poly_precompute_matrix` (*C function*), 420
`nmod_poly_print` (*C function*), 402
`nmod_poly_print_pretty` (*C function*), 402

`nmod_poly_product_roots_nmod_vec` (*C function*), 430
`nmod_poly_randtest` (*C function*), 400
`nmod_poly_randtest_irreducible` (*C function*), 401
`nmod_poly_randtest_monice` (*C function*), 400
`nmod_poly_randtest_monice_irreducible` (*C function*), 401
`nmod_poly_randtest_monice_primitive` (*C function*), 401
`nmod_poly_randtest_pentomial` (*C function*), 400
`nmod_poly_randtest_pentomial_irreducible` (*C function*), 401
`nmod_poly_randtest_sparse_irreducible` (*C function*), 401
`nmod_poly_randtest_trinomial` (*C function*), 400
`nmod_poly_randtest_trinomial_irreducible` (*C function*), 401
`nmod_poly_read` (*C function*), 402
`nmod_poly_realloc` (*C function*), 399
`nmod_poly_rem` (*C function*), 412
`nmod_poly_remove` (*C function*), 414
`nmod_poly_resultant` (*C function*), 425
`nmod_poly_resultant_euclidean` (*C function*), 424
`nmod_poly_resultant_hgcd` (*C function*), 424
`nmod_poly_reverse` (*C function*), 400
`nmod_poly_revert_series` (*C function*), 426
`nmod_poly_scalar_addmul_nmod` (*C function*), 404
`nmod_poly_scalar_mul_nmod` (*C function*), 404
`nmod_poly_set` (*C function*), 400
`nmod_poly_set_coeff_ui` (*C function*), 401
`nmod_poly_set_str` (*C function*), 401
`nmod_poly_set_trunc` (*C function*), 400
`nmod_poly_shift_left` (*C function*), 403
`nmod_poly_shift_right` (*C function*), 403
`nmod_poly_sin_series` (*C function*), 429
`nmod_poly_sinh_series` (*C function*), 429
`nmod_poly_sqrt` (*C function*), 427
`nmod_poly_sqrt_series` (*C function*), 426
`nmod_poly_struct` (*C type*), 399
`nmod_poly_sub` (*C function*), 403
`nmod_poly_sub_series` (*C function*), 403
`nmod_poly_swap` (*C function*), 400
`nmod_poly_t` (*C type*), 399
`nmod_poly_tan_series` (*C function*), 429
`nmod_poly_tanh_series` (*C function*), 429
`nmod_poly_taylor_shift` (*C function*), 419
`nmod_poly_taylor_shift_convolution` (*C function*), 419
`nmod_poly_taylor_shift_horner` (*C function*), 419
`nmod_poly_truncate` (*C function*), 400
`nmod_poly_xgcd` (*C function*), 424
`nmod_poly_xgcd_euclidean` (*C function*), 423
`nmod_poly_xgcd_hgcd` (*C function*), 423
`nmod_poly_zero` (*C function*), 400
`nmod_pow_fmpz` (*C function*), 383
`nmod_pow_ui` (*C function*), 383
`NMOD_RED` (*C macro*), 382
`NMOD_RED2` (*C macro*), 382
`NMOD_RED3` (*C macro*), 382
`nmod_redc_add` (*C function*), 384
`nmod_redc_can_use_fast` (*C function*), 384
`nmod_redc_ctx_init_nmod` (*C function*), 384
`nmod_redc_ctx_init_ui` (*C function*), 384
`nmod_redc_ctx_struct` (*C type*), 384
`nmod_redc_ctx_t` (*C type*), 384
`nmod_redc_fast_add` (*C function*), 384
`nmod_redc_fast_mul` (*C function*), 384
`nmod_redc_fast_mul_two` (*C function*), 384
`nmod_redc_fast_neg` (*C function*), 384
`nmod_redc_fast_normalise` (*C function*), 384
`nmod_redc_fast_sub` (*C function*), 384
`nmod_redc_get_nmod` (*C function*), 384
`nmod_redc_half_add` (*C function*), 385
`nmod_redc_half_can_use_fast` (*C function*), 385
`nmod_redc_half_ctx_init_nmod` (*C function*), 385
`nmod_redc_half_ctx_init_ui` (*C function*), 385
`nmod_redc_half_fast_add` (*C function*), 385
`nmod_redc_half_fast_mul` (*C function*), 385
`nmod_redc_half_fast_sub` (*C function*), 385
`nmod_redc_half_get_nmod` (*C function*), 385
`nmod_redc_half_mul` (*C function*), 385
`nmod_redc_half_set_nmod` (*C function*), 385
`nmod_redc_half_set_ui` (*C function*), 385
`nmod_redc_half_sub` (*C function*), 385
`nmod_redc_mul` (*C function*), 384
`nmod_redc_neg` (*C function*), 384
`nmod_redc_set_nmod` (*C function*), 384
`nmod_redc_set_ui` (*C function*), 384
`nmod_redc_sub` (*C function*), 384
`nmod_sub` (*C function*), 382
`nmod_ui_add_ui` (*C function*), 382
`nmod_ui_mul_ui` (*C function*), 383
`nmod_ui_pow_ui` (*C function*), 383
`NMOD_VEC_DOT` (*C macro*), 388
`NN` (*C macro*), 901
`nn_ptr` (*C type*), 16
`nn_srcptr` (*C type*), 16
`Not` (*C macro*), 897
`NotElement` (*C macro*), 898
`NotEqual` (*C macro*), 897
`NumberE` (*C macro*), 899
`NumberI` (*C macro*), 899

O

`One` (*C macro*), 905
`OpenClosedInterval` (*C macro*), 901
`OpenComplexDisk` (*C macro*), 902
`OpenInterval` (*C macro*), 901
`OpenRealBall` (*C macro*), 902

Or (*C macro*), 897
ordering_t (*C type*), 26
ordering_t.ORD_DEGLEX (*C macro*), 26
ordering_t.ORD_DEGREVLEX (*C macro*), 26
ordering_t.ORD_LEX (*C macro*), 26
Otherwise (*C macro*), 898

P

padic_add (*C function*), 1066
padic_clear (*C function*), 1064
padic_ctx_clear (*C function*), 1064
padic_ctx_init (*C function*), 1064
padic_debug (*C function*), 1070
padic_div (*C function*), 1066
padic_equal (*C function*), 1066
padic_exp (*C function*), 1067
padic_exp_balanced (*C function*), 1068
padic_exp_rectangular (*C function*), 1068
padic_fprint (*C function*), 1070
padic_get_fmpq (*C function*), 1065
padic_get_fmpz (*C function*), 1065
padic_get_mpq (*C function*), 1065
padic_get_mpz (*C function*), 1065
padic_get_prec (*C function*), 1063
padic_get_str (*C function*), 1070
padic_get_val (*C function*), 1063
padic_init (*C function*), 1064
padic_init2 (*C function*), 1064
padic_inv (*C function*), 1067
padic_is_one (*C function*), 1066
padic_is_zero (*C function*), 1066
padic_log (*C function*), 1068
padic_log_balanced (*C function*), 1069
padic_log_rectangular (*C function*), 1069
padic_log_satoh (*C function*), 1069
padic_mat (*C function*), 1077
padic_mat_add (*C function*), 1080
padic_mat_clear (*C function*), 1078
padic_mat_entry (*C function*), 1077
padic_mat_equal (*C function*), 1079
padic_mat_fprint (*C function*), 1079
padic_mat_fprint_pretty (*C function*), 1079
padic_mat_get_entry_padic (*C function*), 1079
padic_mat_get_fmpq_mat (*C function*), 1079
padic_mat_get_prec (*C function*), 1077
padic_mat_get_val (*C function*), 1077
padic_mat_init (*C function*), 1078
padic_mat_init2 (*C function*), 1078
padic_mat_is_canonical (*C function*), 1078
padic_mat_is_empty (*C function*), 1078
padic_mat_is_square (*C function*), 1078
padic_mat_is_zero (*C function*), 1079
padic_mat_mul (*C function*), 1081
padic_mat_ncols (*C function*), 1077
padic_mat_neg (*C function*), 1080
padic_mat_nrows (*C function*), 1077
padic_mat_one (*C function*), 1078
padic_mat_prec (*C function*), 1077

padic_mat_print (*C function*), 1079
padic_mat_print_pretty (*C function*), 1079
padic_mat_randtest (*C function*), 1080
padic_mat_reduce (*C function*), 1078
padic_mat_scalar_div_fmpz (*C function*), 1080
padic_mat_scalar_mul_fmpz (*C function*), 1080
padic_mat_scalar_mul_padic (*C function*), 1080
padic_mat_set (*C function*), 1078
padic_mat_set_entry_padic (*C function*), 1079
padic_mat_set_fmpq_mat (*C function*), 1079
padic_mat_sub (*C function*), 1080
padic_mat_swap (*C function*), 1078
padic_mat_swap_entrywise (*C function*), 1078
padic_mat_transpose (*C function*), 1080
padic_mat_val (*C function*), 1077
padic_mat_zero (*C function*), 1078
padic_mul (*C function*), 1066
padic_neg (*C function*), 1066
padic_one (*C function*), 1066
padic_poly_add (*C function*), 1073
padic_poly_canonicalise (*C function*), 1071
padic_poly_clear (*C function*), 1071
padic_poly_compose (*C function*), 1076
padic_poly_compose_pow (*C function*), 1076
padic_poly_debug (*C function*), 1076
padic_poly_degree (*C function*), 1071
padic_poly_derivative (*C function*), 1075
padic_poly_equal (*C function*), 1073
padic_poly_evaluate_padic (*C function*), 1075
padic_poly_fit_length (*C function*), 1070
padic_poly_fprint (*C function*), 1076
padic_poly_fprint_pretty (*C function*), 1076
padic_poly_get_coeff_padic (*C function*), 1073
padic_poly_get_fmpq_poly (*C function*), 1072
padic_poly_get_fmpz_poly (*C function*), 1072
padic_poly_init (*C function*), 1070
padic_poly_init2 (*C function*), 1070
padic_poly_inv_series (*C function*), 1075
padic_poly_is_canonical (*C function*), 1077
padic_poly_is_one (*C function*), 1073
padic_poly_is_reduced (*C function*), 1077
padic_poly_is_zero (*C function*), 1073
padic_poly_length (*C function*), 1071
padic_poly_mul (*C function*), 1074
padic_poly_neg (*C function*), 1073
padic_poly_one (*C function*), 1072
padic_poly_pow (*C function*), 1074
padic_poly_prec (*C function*), 1071
padic_poly_print (*C function*), 1076
padic_poly_print_pretty (*C function*), 1076
padic_poly_randtest (*C function*), 1071
padic_poly_randtest_not_zero (*C function*),
1071
padic_poly_randtest_val (*C function*), 1071
padic_poly_realloc (*C function*), 1070
padic_poly_reduce (*C function*), 1071
padic_poly_scalar_mul_padic (*C function*),
1074

- padic_poly_set (*C function*), 1072
 padic_poly_set_coeff_padic (*C function*), 1073
 padic_poly_set_fmpq (*C function*), 1072
 padic_poly_set_fmpq_poly (*C function*), 1072
 padic_poly_set_fmpz (*C function*), 1072
 padic_poly_set_fmpz_poly (*C function*), 1072
 padic_poly_set_padic (*C function*), 1072
 padic_poly_set_si (*C function*), 1072
 padic_poly_set_ui (*C function*), 1072
 padic_poly_shift_left (*C function*), 1075
 padic_poly_shift_right (*C function*), 1075
 padic_poly_sub (*C function*), 1073
 padic_poly_swap (*C function*), 1072
 padic_poly_truncate (*C function*), 1071
 padic_poly_val (*C function*), 1071
 padic_poly_zero (*C function*), 1072
 padic_pow_si (*C function*), 1067
 padic_prec (*C function*), 1063
 padic_print (*C function*), 1070
 padic_randtest (*C function*), 1065
 padic_randtest_int (*C function*), 1065
 padic_randtest_not_zero (*C function*), 1065
 padic_reduce (*C function*), 1064
 padic_set (*C function*), 1065
 padic_set_fmpq (*C function*), 1065
 padic_set_fmpz (*C function*), 1065
 padic_set_mpq (*C function*), 1065
 padic_set_mpz (*C function*), 1065
 padic_set_si (*C function*), 1065
 padic_set_ui (*C function*), 1065
 padic_shift (*C function*), 1066
 padic_sqrt (*C function*), 1067
 padic_sub (*C function*), 1066
 padic_swap (*C function*), 1065
 padic_teichmuller (*C function*), 1069
 padic_unit (*C function*), 1063
 padic_val (*C function*), 1063
 padic_val_fac (*C function*), 1069
 padic_val_fac_ui (*C function*), 1069
 padic_val_fac_ui_2 (*C function*), 1069
 padic_zero (*C function*), 1066
 Parentheses (*C macro*), 913
 partitions_fmpz_fmpz (*C function*), 798
 partitions_fmpz_ui (*C function*), 798
 partitions_fmpz_ui_using_doubles (*C function*), 798
 partitions_hrr_sum_arb (*C function*), 798
 partitions_leading_fmpz (*C function*), 799
 partitions_rademacher_bound (*C function*), 798
 PartitionsP (*C macro*), 908
 Path (*C macro*), 904
 Pi (*C macro*), 899
 Pol (*C macro*), 905
 Poles (*C macro*), 904
 PolyLog (*C macro*), 911
 Polynomial (*C macro*), 905
 PolynomialDegree (*C macro*), 905
 PolynomialFractions (*C macro*), 905
 PolynomialRootIndexed (*C macro*), 900
 PolynomialRootNearest (*C macro*), 900
 Polynomials (*C macro*), 905
 Pos (*C macro*), 900
 Pow (*C macro*), 900
 Prime (*C macro*), 907
 PrimePi (*C macro*), 907
 PrimeProduct (*C macro*), 903
 Primes (*C macro*), 901
 PrimeSum (*C macro*), 903
 PrimitiveDirichletCharacters (*C macro*), 912
 PrimitiveReducedPositiveIntegralBinaryQuadraticForms
 (*C macro*), 913
 print_memory_usage (*C function*), 25
 Product (*C macro*), 903
 prof_repeat (*C function*), 23
 ProjectiveComplexNumbers (*C macro*), 902
 ProjectiveRealNumbers (*C macro*), 902
 PSL2Z (*C macro*), 905
 psl2z_clear (*C function*), 747
 psl2z_equal (*C function*), 747
 psl2z_fprint (*C function*), 747
 psl2z_init (*C function*), 747
 psl2z_inv (*C function*), 747
 psl2z_is_correct (*C function*), 747
 psl2z_is_one (*C function*), 747
 psl2z_mul (*C function*), 747
 psl2z_one (*C function*), 747
 psl2z_print (*C function*), 747
 psl2z_randtest (*C function*), 747
 psl2z_set (*C function*), 747
 psl2z_struct (*C type*), 747
 psl2z_swap (*C function*), 747
 psl2z_t (*C type*), 747
 PTR_TO_COEFF (*C function*), 156
- ## Q
- qadic_add (*C function*), 1084
 qadic_clear (*C function*), 1082
 qadic_ctx_clear (*C function*), 1082
 qadic_ctx_degree (*C function*), 1082
 qadic_ctx_init (*C function*), 1081
 qadic_ctx_init_conway (*C function*), 1081
 qadic_ctx_init_modulus (*C function*), 1082
 qadic_ctx_init_modulus_nmod (*C function*),
 1082
 qadic_ctx_print (*C function*), 1082
 qadic_equal (*C function*), 1084
 qadic_exp (*C function*), 1085
 qadic_exp_balanced (*C function*), 1085
 qadic_exp_rectangular (*C function*), 1085
 qadic_fprint_pretty (*C function*), 1088
 qadic_frobenius (*C function*), 1087
 qadic_gen (*C function*), 1083
 qadic_get_padic (*C function*), 1083
 qadic_init (*C function*), 1082
 qadic_init2 (*C function*), 1082
 qadic_inv (*C function*), 1084

qadic_is_one (*C function*), 1084
qadic_is_zero (*C function*), 1084
qadic_log (*C function*), 1086
qadic_log_balanced (*C function*), 1086
qadic_log_rectangular (*C function*), 1086
qadic_mul (*C function*), 1084
qadic_neg (*C function*), 1084
qadic_norm (*C function*), 1088
qadic_norm_analytic (*C function*), 1088
qadic_norm_resultant (*C function*), 1088
qadic_one (*C function*), 1083
qadic_pow (*C function*), 1084
qadic_prec (*C function*), 1083
qadic_print_pretty (*C function*), 1088
qadic_randtest (*C function*), 1083
qadic_randtest_int (*C function*), 1083
qadic_randtest_not_zero (*C function*), 1083
qadic_randtest_val (*C function*), 1083
qadic_reduce (*C function*), 1083
qadic_set (*C function*), 1083
qadic_set_ui (*C function*), 1083
qadic_sqrt (*C function*), 1085
qadic_sub (*C function*), 1084
qadic_teichmuller (*C function*), 1087
qadic_trace (*C function*), 1087
qadic_val (*C function*), 1083
qadic_zero (*C function*), 1083
qfb_array_clear (*C function*), 522
qfb_clear (*C function*), 522
qfb_discriminant (*C function*), 523
qfb_equal (*C function*), 523
qfb_exponent (*C function*), 524
qfb_exponent_element (*C function*), 524
qfb_exponent_grh (*C function*), 524
qfb_hash_clear (*C function*), 522
qfb_hash_find (*C function*), 522
qfb_hash_init (*C function*), 522
qfb_hash_insert (*C function*), 522
qfb_init (*C function*), 522
qfb_inverse (*C function*), 524
qfb_is_primitive (*C function*), 524
qfb_is_principal_form (*C function*), 524
qfb_is_reduced (*C function*), 523
qfb_nucomp (*C function*), 523
qfb_nudupl (*C function*), 523
qfb_pow (*C function*), 524
qfb_pow_ui (*C function*), 523
qfb_prime_form (*C function*), 524
qfb_principal_form (*C function*), 524
qfb_print (*C function*), 523
qfb_reduce (*C function*), 523
qfb_reduced_forms (*C function*), 523
qfb_reduced_forms_large (*C function*), 523
qfb_set (*C function*), 523
QQ (*C macro*), 901
qqbar_abs (*C function*), 548
qqbar_abs2 (*C function*), 548
qqbar_acos_pi (*C function*), 553
qqbar_acot_pi (*C function*), 553
qqbar_acsc_pi (*C function*), 553
qqbar_add (*C function*), 549
qqbar_add_fmpq (*C function*), 549
qqbar_add_fmpz (*C function*), 549
qqbar_add_si (*C function*), 549
qqbar_add_ui (*C function*), 549
qqbar_asec_pi (*C function*), 553
qqbar_asin_pi (*C function*), 553
qqbar_atan_pi (*C function*), 553
qqbar_binary_op (*C function*), 557
qqbar_binop_within_limits (*C function*), 546
qqbar_cache_enclosure (*C function*), 551
qqbar_ceil (*C function*), 549
qqbar_clear (*C function*), 545
qqbar_cmp_im (*C function*), 548
qqbar_cmp_re (*C function*), 548
qqbar_cmp_root_order (*C function*), 548
qqbar_cmpabs (*C function*), 548
qqbar_cmpabs_im (*C function*), 548
qqbar_cmpabs_re (*C function*), 548
QQBAR_COEFFS (*C macro*), 544
qqbar_conj (*C function*), 548
qqbar_conjugates (*C function*), 551
qqbar_cos_pi (*C function*), 553
qqbar_cot_pi (*C function*), 553
qqbar_csc_pi (*C function*), 553
qqbar_csgn (*C function*), 549
qqbar_degree (*C function*), 545
qqbar_denominator (*C function*), 551
qqbar_div (*C function*), 550
qqbar_div_fmpq (*C function*), 550
qqbar_div_fmpz (*C function*), 550
qqbar_div_si (*C function*), 550
qqbar_div_ui (*C function*), 550
qqbar_eigenvalues_fmpq_mat (*C function*), 552
qqbar_eigenvalues_fmpz_mat (*C function*), 552
QQBAR_ENCLOSURE (*C macro*), 544
qqbar_enclosure_raw (*C function*), 557
qqbar_equal (*C function*), 548
qqbar_equal_fmpq_poly_val (*C function*), 548
qqbar_evaluate_fmpq_poly (*C function*), 551
qqbar_evaluate_fmpz_mpoly (*C function*), 551
qqbar_evaluate_fmpz_mpoly_horner (*C function*), 551
qqbar_evaluate_fmpz_mpoly_iter (*C function*), 551
qqbar_evaluate_fmpz_poly (*C function*), 551
qqbar_exp_pi_i (*C function*), 553
qqbar_express_in_field (*C function*), 554
qqbar_floor (*C function*), 549
qqbar_fmpq_div (*C function*), 550
qqbar_fmpq_pow_si_ui (*C function*), 550
qqbar_fmpq_root_ui (*C function*), 550
qqbar_fmpq_sub (*C function*), 549
qqbar_fmpz_div (*C function*), 550
qqbar_fmpz_poly_composed_op (*C function*), 557
qqbar_fmpz_sub (*C function*), 549

`qqbar_get_acb` (*C function*), 551
`qqbar_get_arb` (*C function*), 551
`qqbar_get_arb_im` (*C function*), 551
`qqbar_get_arb_re` (*C function*), 551
`qqbar_get_fexpr_formula` (*C function*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_ALL`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_AUTO_FORM`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_CUBICS`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_CYCLOTOMICS`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_DEFLATED`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_DEPRESSES`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_EXP_FORM`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_GAUSSIAN`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_QUADRATIC`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_QUARTIC`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_QUINTIC`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_RADICAL`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_SEPARATION`
 (*C macro*), 556
`qqbar_get_fexpr_formula.QQBAR_FORMULA_TRIG_FORM`
 (*C macro*), 556
`qqbar_get_fexpr_repr` (*C function*), 555
`qqbar_get_fexpr_root_indexed` (*C function*),
 555
`qqbar_get_fexpr_root_nearest` (*C function*),
 555
`qqbar_get_fmpq` (*C function*), 546
`qqbar_get_fmpz` (*C function*), 546
`qqbar_get_quadratic` (*C function*), 554
`qqbar_guess` (*C function*), 554
`qqbar_hash` (*C function*), 548
`qqbar_height` (*C function*), 546
`qqbar_height_bits` (*C function*), 546
`qqbar_i` (*C function*), 546
`qqbar_im` (*C function*), 548
`qqbar_init` (*C function*), 545
`qqbar_inv` (*C function*), 550
`qqbar_is_algebraic_integer` (*C function*), 545
`qqbar_is_i` (*C function*), 545
`qqbar_is_integer` (*C function*), 545
`qqbar_is_neg_i` (*C function*), 545
`qqbar_is_neg_one` (*C function*), 545
`qqbar_is_one` (*C function*), 545
`qqbar_is_rational` (*C function*), 545
`qqbar_is_real` (*C function*), 546
`qqbar_is_root_of_unity` (*C function*), 553
`qqbar_is_zero` (*C function*), 545
`qqbar_log_pi_i` (*C function*), 553
`qqbar_mul` (*C function*), 549
`qqbar_mul_2exp_si` (*C function*), 549
`qqbar_mul_fmpq` (*C function*), 549
`qqbar_mul_fmpz` (*C function*), 549
`qqbar_mul_si` (*C function*), 549
`qqbar_mul_ui` (*C function*), 549
`qqbar_neg` (*C function*), 549
`qqbar_numerator` (*C function*), 551
`qqbar_one` (*C function*), 546
`qqbar_phi` (*C function*), 546
`QQBAR_POLY` (*C macro*), 544
`qqbar_pow` (*C function*), 550
`qqbar_pow_fmpq` (*C function*), 550
`qqbar_pow_fmpz` (*C function*), 550
`qqbar_pow_si` (*C function*), 550
`qqbar_pow_ui` (*C function*), 550
`qqbar_print` (*C function*), 547
`qqbar_printn` (*C function*), 547
`qqbar_printnd` (*C function*), 547
`qqbar_ptr` (*C type*), 544
`qqbar_randtest` (*C function*), 547
`qqbar_randtest_nonreal` (*C function*), 547
`qqbar_randtest_real` (*C function*), 547
`qqbar_re` (*C function*), 548
`qqbar_re_im` (*C function*), 548
`qqbar_root_of_unity` (*C function*), 553
`qqbar_root_ui` (*C function*), 550
`qqbar_roots_fmpq_poly` (*C function*), 552
`qqbar_roots_fmpz_poly` (*C function*), 552
`qqbar_rsqr` (*C function*), 550
`qqbar_scalar_op` (*C function*), 550
`qqbar_sec_pi` (*C function*), 553
`qqbar_set` (*C function*), 545
`qqbar_set_d` (*C function*), 545
`qqbar_set_fexpr` (*C function*), 555
`qqbar_set_fmpq` (*C function*), 545
`qqbar_set_fmpz` (*C function*), 545
`qqbar_set_re_im` (*C function*), 545
`qqbar_set_re_im_d` (*C function*), 545
`qqbar_set_si` (*C function*), 545
`qqbar_set_ui` (*C function*), 545
`qqbar_sgn` (*C function*), 548
`qqbar_sgn_im` (*C function*), 549
`qqbar_sgn_re` (*C function*), 549
`qqbar_si_div` (*C function*), 550
`qqbar_si_sub` (*C function*), 549
`qqbar_sin_pi` (*C function*), 553
`qqbar_sqr` (*C function*), 550
`qqbar_sqrt` (*C function*), 550
`qqbar_sqrt_ui` (*C function*), 550
`qqbar_srcptr` (*C type*), 544
`qqbar_struct` (*C type*), 544
`qqbar_sub` (*C function*), 549
`qqbar_sub_fmpq` (*C function*), 549
`qqbar_sub_fmpz` (*C function*), 549
`qqbar_sub_si` (*C function*), 549

qqbar_sub_ui (*C function*), 549
qqbar_swap (*C function*), 545
qqbar_t (*C type*), 544
qqbar_tan_pi (*C function*), 553
qqbar_ui_div (*C function*), 550
qqbar_ui_sub (*C function*), 549
qqbar_within_limits (*C function*), 546
qqbar_zero (*C function*), 546
QSeriesCoefficient (*C macro*), 906
qsieve_add_to_hashtable (*C function*), 308
qsieve_collect_relations (*C function*), 308
qsieve_compare_relation (*C function*), 308
qsieve_compute_C (*C function*), 308
qsieve_compute_pre_data (*C function*), 307
qsieve_do_sieving (*C function*), 308
qsieve_do_sieving2 (*C function*), 308
qsieve_evaluate_candidate (*C function*), 308
qsieve_evaluate_sieve (*C function*), 308
qsieve_factor (*C function*), 309
qsieve_get_table_entry (*C function*), 308
qsieve_init_AO (*C function*), 307
qsieve_init_poly_first (*C function*), 308
qsieve_init_poly_next (*C function*), 308
qsieve_insert_relation2 (*C function*), 309
qsieve_knuth_schroeppel (*C function*), 307
qsieve_merge_relation (*C function*), 308
qsieve_next_AO (*C function*), 307
qsieve_parse_relation (*C function*), 308
qsieve_primes_increment (*C function*), 307
qsieve_primes_init (*C function*), 307
qsieve_process_relation (*C function*), 309
qsieve_remove_duplicates (*C function*), 309
qsieve_write_to_file (*C function*), 308
QuotientRing (*C macro*), 906

R

radix_add (*C function*), 310
radix_clear (*C function*), 309
radix_cmp_bn_half (*C function*), 312
radix_digit_radix (*C function*), 309
radix_div (*C function*), 312
radix_div_approx (*C function*), 312
radix_div_approx_invmul (*C function*), 312
radix_divexact (*C function*), 312
radix_divexact_1 (*C function*), 311
radix_divrem (*C function*), 312
radix_divrem_1 (*C function*), 311
radix_divrem_newton (*C function*), 312
radix_divrem_newton_karp_markstein (*C function*), 312
radix_divrem_preinv (*C function*), 312
radix_divrem_two (*C function*), 311
radix_divrem_via_mpn (*C function*), 312
radix_get_mpn (*C function*), 313
radix_get_mpn_basecase (*C function*), 313
radix_get_mpn_divconquer (*C function*), 313
radix_get_str_decimal (*C function*), 313
radix_get_str_sum (*C function*), 313

radix_init (*C function*), 309
radix_init_randtest (*C function*), 309
radix_integer_abs (*C function*), 314
radix_integer_add (*C function*), 314
radix_integer_cdiv_q (*C function*), 315
radix_integer_cdiv_qr (*C function*), 315
radix_integer_cdiv_r (*C function*), 315
radix_integer_clear (*C function*), 314
radix_integer_cmp (*C function*), 314
radix_integer_cmpabs (*C function*), 314
radix_integer_div (*C function*), 315
radix_integer_divexact (*C function*), 315
radix_integer_equal (*C function*), 314
radix_integer_fdiv_q (*C function*), 315
radix_integer_fdiv_qr (*C function*), 315
radix_integer_fdiv_r (*C function*), 315
radix_integer_fit_limbs (*C function*), 314
radix_integer_get_digit (*C function*), 314
radix_integer_get_fmpz (*C function*), 314
radix_integer_get_limb (*C function*), 314
radix_integer_init (*C function*), 314
radix_integer_invmul_limbs (*C function*), 316
radix_integer_is_neg_one (*C function*), 314
radix_integer_is_normalised (*C function*), 314
radix_integer_is_one (*C function*), 314
radix_integer_is_zero (*C function*), 314
radix_integer_lshift_digits (*C function*), 315
radix_integer_lshift_limbs (*C function*), 315
radix_integer_mod_limbs (*C function*), 315
radix_integer_mul (*C function*), 314
radix_integer_mullow_limbs (*C function*), 315
radix_integer_neg (*C function*), 314
radix_integer_neg_one (*C function*), 314
radix_integer_one (*C function*), 314
radix_integer_rand_limbs (*C function*), 314
radix_integer_randtest_limbs (*C function*), 314
radix_integer_rshift_digits (*C function*), 315
radix_integer_rshift_limbs (*C function*), 315
radix_integer_set (*C function*), 314
radix_integer_set_digit (*C function*), 315
radix_integer_set_fmpz (*C function*), 314
radix_integer_set_limb (*C function*), 315
radix_integer_set_si (*C function*), 314
radix_integer_set_ui (*C function*), 314
radix_integer_sgn (*C function*), 314
radix_integer_size_digits (*C function*), 314
radix_integer_size_limbs (*C function*), 314
radix_integer_smod_limbs (*C function*), 315
radix_integer_ssize_digits (*C function*), 314
radix_integer_ssize_limbs (*C function*), 314
radix_integer_struct (*C type*), 313
radix_integer_sub (*C function*), 314
radix_integer_t (*C type*), 313
radix_integer_tdiv_q (*C function*), 315
radix_integer_tdiv_qr (*C function*), 315
radix_integer_tdiv_r (*C function*), 315
radix_integer_trunc_limbs (*C function*), 315

-
- `radix_integer_valuation_digits` (*C function*), 315
 - `radix_integer_valuation_limbs` (*C function*), 315
 - `radix_integer_zero` (*C function*), 314
 - `radix_inv_approx` (*C function*), 312
 - `radix_inv_approx_basecase` (*C function*), 312
 - `radix_invmod_bn` (*C function*), 312
 - `radix_limb_exponent` (*C function*), 309
 - `radix_limb_radix` (*C function*), 309
 - `radix_lshift_digits` (*C function*), 310
 - `radix_mul` (*C function*), 311
 - `radix_mul_1` (*C function*), 311
 - `radix_mul_two` (*C function*), 311
 - `radix_mulmid` (*C function*), 311
 - `radix_mulmid_classical` (*C function*), 311
 - `radix_mulmid_fft_small` (*C function*), 311
 - `radix_mulmid_KS` (*C function*), 311
 - `radix_mulmid_naive` (*C function*), 311
 - `radix_neg` (*C function*), 310
 - `radix_rand_digits` (*C function*), 310
 - `radix_rand_limbs` (*C function*), 310
 - `radix_randtest_digits` (*C function*), 310
 - `radix_randtest_limbs` (*C function*), 310
 - `radix_rshift_digits` (*C function*), 310
 - `radix_rsqr_1_approx` (*C function*), 312
 - `radix_rsqr_1_approx_basecase` (*C function*), 312
 - `radix_set_mpn` (*C function*), 313
 - `radix_set_mpn_basecase` (*C function*), 313
 - `radix_set_mpn_divconquer` (*C function*), 313
 - `radix_set_mpn_need_alloc` (*C function*), 313
 - `radix_size_digits` (*C function*), 310
 - `radix_size_digits_1` (*C function*), 310
 - `radix_sqr` (*C function*), 311
 - `radix_struct` (*C type*), 309
 - `radix_sub` (*C function*), 310
 - `radix_t` (*C type*), 309
 - `Range` (*C macro*), 901
 - `Re` (*C macro*), 906
 - `RealAbs` (*C macro*), 906
 - `RealAlgebraicNumbers` (*C macro*), 901
 - `RealBall` (*C macro*), 902
 - `RealDerivative` (*C macro*), 904
 - `RealInfinites` (*C macro*), 902
 - `RealLimit` (*C macro*), 904
 - `RealSignedInfinites` (*C macro*), 902
 - `RealSingularityClosure` (*C macro*), 903
 - `Repeat` (*C macro*), 897
 - `Residue` (*C macro*), 904
 - `RiemannHypothesis` (*C macro*), 911
 - `RiemannXi` (*C macro*), 911
 - `RiemannZeta` (*C macro*), 911
 - `RiemannZetaZero` (*C macro*), 911
 - `RightLimit` (*C macro*), 904
 - `Rings` (*C macro*), 906
 - `RisingFactorial` (*C macro*), 909
 - `Root` (*C macro*), 900
 - `RootOfUnity` (*C macro*), 900
 - `Row` (*C macro*), 905
 - `RowMatrix` (*C macro*), 905
 - `RR` (*C macro*), 901
- ## S
- `Same` (*C macro*), 897
 - `sdiv_qrnnd` (*C macro*), 279
 - `Sec` (*C macro*), 907
 - `Sech` (*C macro*), 907
 - `SequenceLimit` (*C macro*), 904
 - `SequenceLimitInferior` (*C macro*), 904
 - `SequenceLimitSuperior` (*C macro*), 904
 - `Ser` (*C macro*), 905
 - `Set` (*C macro*), 898
 - `SetMinus` (*C macro*), 899
 - `Sets` (*C macro*), 899
 - `ShowExpandedNormalForm` (*C macro*), 913
 - `Sign` (*C macro*), 906
 - `SignExtendedComplexNumbers` (*C macro*), 902
 - `Sin` (*C macro*), 907
 - `Sinc` (*C macro*), 908
 - `SingularValues` (*C macro*), 905
 - `Sinh` (*C macro*), 907
 - `SinhIntegral` (*C macro*), 910
 - `SinIntegral` (*C macro*), 910
 - `SL2Z` (*C macro*), 905
 - `SloaneA` (*C macro*), 908
 - `slong` (*C type*), 16
 - `smul_ppmm` (*C macro*), 279
 - `Solutions` (*C macro*), 903
 - `sp2gz_block_diag` (*C function*), 756
 - `sp2gz_decompose` (*C function*), 757
 - `sp2gz_dim` (*C function*), 756
 - `sp2gz_embed` (*C function*), 756
 - `sp2gz_fundamental` (*C function*), 757
 - `sp2gz_inv` (*C function*), 757
 - `sp2gz_is_block_diag` (*C function*), 757
 - `sp2gz_is_correct` (*C function*), 757
 - `sp2gz_is_embedded` (*C function*), 757
 - `sp2gz_is_j` (*C function*), 757
 - `sp2gz_is_trig` (*C function*), 757
 - `sp2gz_j` (*C function*), 756
 - `sp2gz_nb_fundamental` (*C function*), 756
 - `sp2gz_randtest` (*C function*), 757
 - `sp2gz_restrict` (*C function*), 756
 - `sp2gz_set_blocks` (*C function*), 756
 - `sp2gz_trig` (*C function*), 756
 - `SpecialLinearGroup` (*C macro*), 905
 - `Spectrum` (*C macro*), 905
 - `SphericalHarmonicY` (*C macro*), 909
 - `Sqrt` (*C macro*), 900
 - `SquaresR` (*C macro*), 907
 - `start_clock` (*C function*), 23
 - `Step` (*C macro*), 897
 - `StieltjesGamma` (*C macro*), 912
 - `StirlingCycle` (*C macro*), 908
 - `StirlingS1` (*C macro*), 908

StirlingS2 (*C macro*), 908
StirlingSeriesRemainder (*C macro*), 909
stop_clock (*C function*), 23
Sub (*C macro*), 900
sub_dddmmsss (*C macro*), 278
sub_ddmsss (*C macro*), 278
Subscript (*C macro*), 913
Subset (*C macro*), 899
SubsetEqual (*C macro*), 899
Subsets (*C macro*), 899
Sum (*C macro*), 903
Supremum (*C macro*), 903
SymmetricPolynomial (*C macro*), 908

T

Tan (*C macro*), 907
Tanh (*C macro*), 907
thread_pool_clear (*C function*), 25
thread_pool_get_size (*C function*), 25
thread_pool_give_back (*C function*), 25
thread_pool_handle (*C type*), 25
thread_pool_init (*C function*), 25
thread_pool_request (*C function*), 25
thread_pool_set_size (*C function*), 25
thread_pool_t (*C type*), 25
thread_pool_wait (*C function*), 25
thread_pool_wake (*C function*), 25
TIMEIT_END_REPEAT (*C macro*), 23
TIMEIT_ONCE_START (*C macro*), 23
TIMEIT_ONCE_STOP (*C macro*), 23
timeit_print (*C function*), 22
TIMEIT_REPEAT (*C macro*), 23
timeit_start (*C function*), 22
TIMEIT_START (*C macro*), 23
timeit_stop (*C function*), 22
TIMEIT_STOP (*C macro*), 23
trig_prod_init (*C function*), 296
True (*C macro*), 897
truth_and (*C function*), 40
truth_not (*C function*), 40
truth_or (*C function*), 40
truth_println (*C function*), 40
truth_t (*C enum*), 39
truth_t.T_FALSE (*C macro*), 39
truth_t.T_TRUE (*C macro*), 39
truth_t.T_UNKNOWN (*C macro*), 39
Tuple (*C macro*), 898
Tuples (*C macro*), 899

U

udiv_qrnnd (*C macro*), 279
udiv_qrnnd_preinv (*C macro*), 279
ulong (*C type*), 16
umul_ppmm (*C macro*), 279
Undefined (*C macro*), 899
Union (*C macro*), 899
UniqueSolution (*C macro*), 903
UniqueZero (*C macro*), 903

UnitCircle (*C macro*), 902
unity_zp (*C type*), 288
unity_zp_add (*C function*), 289
unity_zp_aut (*C function*), 290
unity_zp_aut_inv (*C function*), 290
unity_zp_clear (*C function*), 288
unity_zp_coeff_add_fmpz (*C function*), 289
unity_zp_coeff_add_ui (*C function*), 289
unity_zp_coeff_dec (*C function*), 289
unity_zp_coeff_inc (*C function*), 289
unity_zp_coeff_set_fmpz (*C function*), 289
unity_zp_coeff_set_ui (*C function*), 289
unity_zp_copy (*C function*), 288
unity_zp_equal (*C function*), 288
unity_zp_init (*C function*), 288
unity_zp_is_unity (*C function*), 288
unity_zp_jacobi_sum_2q_one (*C function*), 290
unity_zp_jacobi_sum_2q_two (*C function*), 290
unity_zp_jacobi_sum_pq (*C function*), 290
unity_zp_mul (*C function*), 289
unity_zp_mul_inplace (*C function*), 289
unity_zp_mul_scalar_ui (*C function*), 289
unity_zp_pow_2k_fmpz (*C function*), 289
unity_zp_pow_2k_ui (*C function*), 289
unity_zp_pow_fmpz (*C function*), 289
unity_zp_pow_sliding_fmpz (*C function*), 290
unity_zp_pow_ui (*C function*), 289
unity_zp_reduce_cyclotomic (*C function*), 290
unity_zp_set_zero (*C function*), 288
unity_zp_sqr (*C function*), 289
unity_zp_sqr_inplace (*C function*), 289
unity_zp_swap (*C function*), 288
unity_zpq (*C type*), 288
unity_zpq_add (*C function*), 291
unity_zpq_clear (*C function*), 290
unity_zpq_coeff_add (*C function*), 291
unity_zpq_coeff_set_fmpz (*C function*), 291
unity_zpq_coeff_set_ui (*C function*), 291
unity_zpq_copy (*C function*), 290
unity_zpq_equal (*C function*), 290
unity_zpq_gauss_sum (*C function*), 291
unity_zpq_gauss_sum_sigma_pow (*C function*),
291
unity_zpq_init (*C function*), 290
unity_zpq_mul (*C function*), 291
unity_zpq_mul_unity_p_pow (*C function*), 291
unity_zpq_pow (*C function*), 291
unity_zpq_pow_ui (*C function*), 291
unity_zpq_swap (*C function*), 290
Unknown (*C macro*), 900
UnsignedInfinity (*C macro*), 902
UpperGamma (*C macro*), 910
UpperHalfPlane (*C macro*), 902
UWORD_MAX (*C macro*), 16
UWORD_MIN (*C macro*), 16

V

vec1d (*C type*), 31

vec1d_abs (*C function*), 34
 vec1d_add (*C function*), 34
 vec1d_addsub (*C function*), 34
 vec1d_blendv (*C function*), 35
 vec1d_div (*C function*), 34
 vec1d_fmadd (*C function*), 34
 vec1d_fmsub (*C function*), 34
 vec1d_fnmadd (*C function*), 34
 vec1d_fnmsub (*C function*), 34
 vec1d_half (*C function*), 34
 vec1d_load (*C function*), 32
 vec1d_load_aligned (*C function*), 32
 vec1d_load_unaligned (*C function*), 32
 vec1d_max (*C function*), 34
 vec1d_min (*C function*), 34
 vec1d_mul (*C function*), 34
 vec1d_mulmod (*C function*), 35
 vec1d_neg (*C function*), 34
 vec1d_nmulmod (*C function*), 36
 vec1d_one (*C function*), 33
 vec1d_reduce_0n_to_pmhn (*C function*), 35
 vec1d_reduce_2n_to_n (*C function*), 35
 vec1d_reduce_pm1n_to_pmhn (*C function*), 35
 vec1d_reduce_pm1no_to_0n (*C function*), 35
 vec1d_reduce_to_0n (*C function*), 35
 vec1d_reduce_to_pm1n (*C function*), 35
 vec1d_reduce_to_pm1no (*C function*), 35
 vec1d_round (*C function*), 33
 vec1d_same (*C function*), 33
 vec1d_same_mod (*C function*), 35
 vec1d_set_d (*C function*), 32
 vec1d_store (*C function*), 32
 vec1d_store_aligned (*C function*), 32
 vec1d_store_unaligned (*C function*), 32
 vec1d_sub (*C function*), 34
 vec1d_zero (*C function*), 33
 vec1n (*C type*), 31
 vec2d (*C type*), 31
 vec2n (*C type*), 31
 vec4d (*C type*), 31
 vec4d_abs (*C function*), 34
 vec4d_add (*C function*), 34
 vec4d_addsub (*C function*), 34
 vec4d_blendv (*C function*), 35
 vec4d_cmp_ge (*C function*), 33
 vec4d_cmp_gt (*C function*), 33
 vec4d_convert_limited_vec4n (*C function*), 33
 vec4d_div (*C function*), 34
 vec4d_fmadd (*C function*), 34
 vec4d_fmsub (*C function*), 34
 vec4d_fnmadd (*C function*), 34
 vec4d_fnmsub (*C function*), 34
 vec4d_get_index (*C function*), 32
 vec4d_half (*C function*), 34
 vec4d_load (*C function*), 32
 vec4d_load_aligned (*C function*), 32
 vec4d_load_unaligned (*C function*), 32
 vec4d_max (*C function*), 34
 vec4d_min (*C function*), 34
 vec4d_mul (*C function*), 34
 vec4d_mulmod (*C function*), 35
 vec4d_neg (*C function*), 34
 vec4d_nmulmod (*C function*), 36
 vec4d_one (*C function*), 33
 vec4d_permute2_0_2 (*C function*), 33
 vec4d_permute2_1_3 (*C function*), 33
 vec4d_permute_0_2_1_3 (*C function*), 33
 vec4d_permute_3_1_2_0 (*C function*), 33
 vec4d_permute_3_2_1_0 (*C function*), 33
 vec4d_print (*C function*), 32
 vec4d_reduce_0n_to_pmhn (*C function*), 35
 vec4d_reduce_2n_to_n (*C function*), 35
 vec4d_reduce_pm1n_to_pmhn (*C function*), 35
 vec4d_reduce_pm1no_to_0n (*C function*), 35
 vec4d_reduce_to_0n (*C function*), 35
 vec4d_reduce_to_pm1n (*C function*), 35
 vec4d_reduce_to_pm1no (*C function*), 35
 vec4d_round (*C function*), 33
 vec4d_same (*C function*), 33
 vec4d_same_mod (*C function*), 35
 vec4d_set_d (*C function*), 32
 vec4d_set_d4 (*C function*), 32
 vec4d_store (*C function*), 32
 vec4d_store_aligned (*C function*), 32
 vec4d_store_unaligned (*C function*), 32
 vec4d_sub (*C function*), 34
 VEC4D_TRANSPOSE (*C macro*), 33
 vec4d_unpack_hi_permute_0_2_1_3 (*C function*), 33
 vec4d_unpack_lo_permute_0_2_1_3 (*C function*), 33
 vec4d_unpackhi (*C function*), 33
 vec4d_unpackhi_permute_3_1_2_0 (*C function*), 33
 vec4d_unpacklo (*C function*), 33
 vec4d_unpacklo_permute_3_1_2_0 (*C function*), 33
 vec4d_zero (*C function*), 33
 vec4n (*C type*), 31
 vec4n_add (*C function*), 34
 vec4n_addmod (*C function*), 36
 vec4n_addmod_limited (*C function*), 36
 vec4n_bit_and (*C function*), 35
 vec4n_bit_shift_right (*C function*), 35
 vec4n_load_unaligned (*C function*), 32
 vec4n_print (*C function*), 32
 vec4n_set_n (*C function*), 32
 vec4n_set_n4 (*C function*), 32
 vec4n_store_unaligned (*C function*), 32
 vec4n_sub (*C function*), 34
 vec8d (*C type*), 31
 vec8d_add (*C function*), 34
 vec8d_blendv (*C function*), 35
 vec8d_div (*C function*), 34
 vec8d_fmadd (*C function*), 34
 vec8d_fmsub (*C function*), 34

`vec8d_fmadd` (*C function*), 34
`vec8d_fnmsub` (*C function*), 34
`vec8d_get_index` (*C function*), 32
`vec8d_load` (*C function*), 32
`vec8d_load_aligned` (*C function*), 32
`vec8d_load_unaligned` (*C function*), 32
`vec8d_max` (*C function*), 34
`vec8d_min` (*C function*), 34
`vec8d_mul` (*C function*), 34
`vec8d_mulmod` (*C function*), 35
`vec8d_neg` (*C function*), 34
`vec8d_nmulmod` (*C function*), 36
`vec8d_one` (*C function*), 33
`vec8d_reduce_2n_to_n` (*C function*), 35
`vec8d_reduce_pm1n_to_pmhn` (*C function*), 35
`vec8d_reduce_pm1no_to_0n` (*C function*), 35
`vec8d_reduce_to_0n` (*C function*), 35
`vec8d_reduce_to_pm1n` (*C function*), 35
`vec8d_reduce_to_pm1no` (*C function*), 35
`vec8d_round` (*C function*), 33
`vec8d_same` (*C function*), 33
`vec8d_set_d` (*C function*), 32
`vec8d_set_d8` (*C function*), 32
`vec8d_store` (*C function*), 32
`vec8d_store_aligned` (*C function*), 32
`vec8d_store_unaligned` (*C function*), 32
`vec8d_sub` (*C function*), 34
`vec8d_zero` (*C function*), 33
`vec8n` (*C type*), 31
`vec8n_addmod` (*C function*), 36
`vec8n_addmod_limited` (*C function*), 36
`vec8n_bit_and` (*C function*), 35
`vec8n_bit_shift_right` (*C function*), 35
`vec8n_convert_limited_vec8d` (*C function*), 33
`vec8n_load_unaligned` (*C function*), 32
`vec8n_set_n` (*C function*), 32

W

`WeierstrassP` (*C macro*), 913
`WeierstrassSigma` (*C macro*), 913
`WeierstrassZeta` (*C macro*), 913
`Where` (*C macro*), 897
`WORD_MAX` (*C macro*), 16
`WORD_MIN` (*C macro*), 16

X

`XGCD` (*C macro*), 907

Z

`z_kronecker` (*C function*), 278
`z_mul_checked` (*C function*), 277
`z_randint` (*C function*), 277
`z_randtest` (*C function*), 277
`z_randtest_not_zero` (*C function*), 277
`z_sizeinbase` (*C function*), 277
`Zero` (*C macro*), 905
`ZeroMatrix` (*C macro*), 905
`Zeros` (*C macro*), 903

`ZZ` (*C macro*), 901