Package 'SimDesign'

February 24, 2026

Title Structure for Organizing Monte Carlo Simulation Designs

Version 2.24

Description Provides tools to safely and efficiently organize and execute Monte Carlo simulation experiments in R.
The package controls the structure and back-end of Monte Carlo simulation experiments by utilizing a generate-analyse-summarise workflow. The workflow safeguards against common simulation coding issues, such as automatically re-simulating non-convergent results, prevents inadvertently overwriting simulation files, catches error and warning messages during execution, implicitly supports parallel processing with high-quality random number generation, and provides tools for managing high-performance computing (HPC) array jobs submitted to schedulers such as SLURM. For a pedagogical introduction to the package see Sigal and Chalmers (2016) <doi:10.1080/10691898.2016.1246953>. For a more in-depth overview of
the package and its design philosophy see Chalmers and Adkins (2020) <doi:10.20982/tqmp.16.4.p248>.

VignetteBuilder knitr

Depends R (>= 4.1.0)

Imports methods, testthat, parallel, parallelly, dplyr, sessioninfo, beepr, pbapply (>= 1.3-0), future, future.apply, progressr, R.utils, codetools, clipr, e1071, stats

Suggests snow, knitr, ggplot2, tidyr, purrr, shiny, copula, extraDistr, renv, cli, job, future.batchtools, FrF2, rmarkdown, RPushbullet, httr

License GPL (>= 2)

ByteCompile yes

LazyData true

URL http://philchalmers.github.io/SimDesign/, https://github.com/philchalmers/SimDesign/wiki

RoxygenNote 7.3.3

Encoding UTF-8

NeedsCompilation no

Author Phil Chalmers [aut, cre] (ORCID:
<https://orcid.org/0000-0001-5332-2810>),
Matthew Sigal [ctb],
Ogreden Oguzhan [ctb],
Mikko Rönkkö [aut],
Moritz Ketzer [ctb]

Maintainer Phil Chalmers <rphilip.chalmers@gmail.com>

Repository CRAN

Date/Publication 2026-02-24 06:20:49 UTC

Contents

`addMissing` *Add missing values to a vector given a MCAR, MAR, or MNAR scheme*

Description

Given an input vector, replace elements of this vector with missing values according to some scheme. Default method replaces input values with a MCAR scheme (where on average 10% of the values will be replaced with `NA`s). MAR and MNAR are supported by replacing the default `FUN` argument.

Usage

```
addMissing(y, fun = function(y, rate = 0.1, ...) rep(rate, length(y)), ...)
```

Arguments

y	an input vector that should contain missing data in the form of NA's
fun	a user defined function indicating the missing data mechanism for each element in y. Function must return a vector of probability values with the length equal to the length of y. Each value in the returned vector indicates the probability that the respective element in y will be replaced with NA. Function must contain the argument y, representing the input vector, however any number of additional arguments can be included
...	additional arguments to be passed to FUN

Details

Given an input vector y, and other relevant variables inside (X) and outside (Z) the data-set, the three types of missingness are:

MCAR Missing completely at random (MCAR). This is realized by randomly sampling the values of the input vector (y) irrespective of the possible values in X and Z. Therefore missing values are randomly sampled and do not depend on any data characteristics and are truly random

MAR Missing at random (MAR). This is realized when values in the dataset (X) predict the missing data mechanism in y; conceptually this is equivalent to $P(y = NA|X)$. This requires the user to define a custom missing data function

MNAR Missing not at random (MNAR). This is similar to MAR except that the missing mechanism comes from the value of y itself or from variables outside the working dataset; conceptually this is equivalent to $P(y = NA|X, Z, y)$. This requires the user to define a custom missing data function

Value

the input vector y with the sampled NA values (according to the FUN scheme)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
## Not run:

set.seed(1)
y <- rnorm(1000)
```

```
## 10% missing rate with default FUN
head(ymiss <- addMissing(y), 10)

## 50% missing with default FUN
head(ymiss <- addMissing(y, rate = .5), 10)

## missing values only when female and low
X <- data.frame(group = sample(c('male', 'female'), 1000, replace=TRUE),
                level = sample(c('high', 'low'), 1000, replace=TRUE))
head(X)

fun <- function(y, X, ...){
    p <- rep(0, length(y))
    p[X$group == 'female' & X$level == 'low'] <- .2
    p
}

ymiss <- addMissing(y, X, fun=fun)
tail(cbind(ymiss, X), 10)

## missingness as a function of elements in X (i.e., a type of MAR)
fun <- function(y, X){
   # missingness with a logistic regression approach
   df <- data.frame(y, X)
   mm <- model.matrix(y ~ group + level, df)
   cfs <- c(-5, 2, 3) #intercept, group, and level coefs
   z <- cfs %*% t(mm)
   plogis(z)
}

ymiss <- addMissing(y, X, fun=fun)
tail(cbind(ymiss, X), 10)

## missing values when y elements are large (i.e., a type of MNAR)
fun <- function(y) ifelse(abs(y) > 1, .4, 0)
ymiss <- addMissing(y, fun=fun)
tail(cbind(y, ymiss), 10)


## End(Not run)
```

`Analyse` *Compute estimates and statistics*

Description

Compute all relevant test statistics, parameter estimates, detection rates, and so on. This is the computational heavy lifting portion of the Monte Carlo simulation. Users may define a single Analysis

function to perform all the analyses in the same function environment, or may define a list of named functions to runSimulation to allow for a more modularized approach to performing the analyses in independent blocks (but that share the same generated data). Note that if a suitable Generate function was not supplied then this function can be used to be generate and analyse the Monte Carlo data (though in general this setup is not recommended for larger simulations).

Usage

```
Analyse(condition, dat, fixed_objects)
```

Arguments

condition	a single row from the design input (as a data.frame), indicating the simulation conditions
dat	the dat object returned from the Generate function (usually a data.frame, matrix, vector, or list)
fixed_objects	object passed down from runSimulation

Details

In some cases, it may be easier to change the output to a named list containing different parameter configurations (e.g., when determining RMSE values for a large set of population parameters).

The use of try functions is generally not required in this function because Analyse is internally wrapped in a try call. Therefore, if a function stops early then this will cause the function to halt internally, the message which triggered the stop will be recorded, and Generate will be called again to obtain a different dataset. That said, it may be useful for users to throw their own stop commands if the data should be re-drawn for other reasons (e.g., an estimated model terminated correctly but the maximum number of iterations were reached).

Value

returns a named numeric vector or data.frame with the values of interest (e.g., p-values, effects sizes, etc), or a list containing values of interest (e.g., separate matrix and vector of parameter estimates corresponding to elements in parameters). If a data.frame is returned with more than 1 row then these objects will be wrapped into suitable list objects

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

stop, AnalyseIf, manageWarnings

Examples

```
## Not run:

analyse <- function(condition, dat, fixed_objects) {

   # require packages/define functions if needed, or better yet index with the :: operator
    require(stats)
    mygreatfunction <- function(x) print('Do some stuff')

    #wrap computational statistics in try() statements to control estimation problems
    welch <- t.test(DV ~ group, dat)
    ind <- stats::t.test(DV ~ group, dat, var.equal=TRUE)

    # In this function the p values for the t-tests are returned,
    #  and make sure to name each element, for future reference
    ret <- c(welch = welch$p.value,
             independent = ind$p.value)

    return(ret)
}

# A more modularized example approach

analysis_welch <- function(condition, dat, fixed_objects) {
    welch <- t.test(DV ~ group, dat)
    ret <- c(p=welch$p.value)
    ret
}

analysis_ind <- function(condition, dat, fixed_objects) {
    ind <- t.test(DV ~ group, dat, var.equal=TRUE)
    ret <- c(p=ind$p.value)
    ret
}

# pass functions as a named list
# runSimulation(..., analyse=list(welch=analyse_welch, independent=analysis_ind))


## End(Not run)
```

`AnalyseIf` *Perform a test that indicates whether a given* `Analyse()` *function should be executed*

Description

This function is designed to prevent specific analysis function executions when the design conditions are not met. Primarily useful when the `analyse` argument to `runSimulation` was input as a named list object, however some of the analysis functions are not interesting/compatible with the generated data and should therefore be skipped.

Usage

```
AnalyseIf(x, condition = NULL)
```

Arguments

x	logical statement to evaluate. If the statement evaluates to TRUE then the remainder of the defined function will be evaluated
condition	(optional) the current design condition. This does not need to be supplied if the expression in x evaluates to valid logical (e.g., use Attach(condition) prior to using AnalyseIf, or use with(condition, AnalyseIf(someLogicalTest)))

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

Analyse, runSimulation

Examples

```
## Not run:

Design <- createDesign(N=c(10,20,30), var.equal = c(TRUE, FALSE))

Generate <- function(condition, fixed_objects) {
  Attach(condition)
  dat <- data.frame(DV = rnorm(N*2), IV = gl(2, N, labels=c('G1', 'G2')))
  dat
}

# always run this analysis for each row in Design
Analyse1 <- function(condition, dat, fixed_objects) {
  mod <- t.test(DV ~ IV, data=dat)
  mod$p.value
}

# Only perform analysis when variances are equal and N = 20 or 30
Analyse2 <- function(condition, dat, fixed_objects) {
  AnalyseIf(var.equal && N %in% c(20, 30), condition)
  mod <- t.test(DV ~ IV, data=dat, var.equal=TRUE)
  mod$p.value
```

```
}

Summarise <- function(condition, results, fixed_objects) {
  ret <- EDR(results, alpha=.05)
  ret
}

#--------------------------------------------------------------------

# append names 'Welch' and 'independent' to associated output
res <- runSimulation(design=Design, replications=100, generate=Generate,
                     analyse=list(Welch=Analyse1, independent=Analyse2),
                     summarise=Summarise)
res

# leave results unnamed
res <- runSimulation(design=Design, replications=100, generate=Generate,
                     analyse=list(Analyse1, Analyse2),
                     summarise=Summarise)



## End(Not run)
```

`Attach` *Attach objects for easier reference*

Description

The behaviour of this function is very similar to `attach`, however it is environment specific, and therefore only remains defined in a given function rather than in the Global Environment. Hence, this function is much safer to use than the `attach`, which incidentally should never be used in your code. This is useful primarily as a convenience function when you prefer to call the variable names in `condition` directly rather than indexing with `condition$sample_size` or `with(condition, sample_size)`, for example.

Usage

```
Attach(
  ...,
  omit = NULL,
  check = TRUE,
  attach_listone = TRUE,
  RStudio_flags = FALSE,
  clip = interactive()
)
```

Arguments

...	a comma separated list of data.frame, tibble, list, or matrix objects containing (column) elements that should be placed in the current working environment
omit	an optional character vector containing the names of objects that should not be attached to the current environment. For instance, if the objects named 'a' and 'b' should not be attached then use omit = c('a', 'b'). When NULL (default) all objects are attached
check	logical; check to see if the function will accidentally replace previously defined variables with the same names as in condition? Default is TRUE, which will avoid this error
attach_listone	logical; if the element to be assign is a list of length one then assign the first element of this list with the associated name. This generally avoids adding an often unnecessary list 1 index, such as name <- list[[1L]]
RStudio_flags	logical; print R script output comments that disable flagged missing variables in RStudio? Requires the form Attach(Design, RStudio_flags=TRUE) or in an interactive debugging session Attach(condition, RStudio_flags=TRUE)
clip	when Rstudio_flags = TRUE should the output be copied to the clipboard using clipr? Makes it easier to paste into existing code. Only clipped in interactive mode

Details

Note that if you are using RStudio with the *"Warn if variable used has no definition in scope"* diagnostic flag then using Attach() will raise suspensions. To suppress such issues, you can either disable such flags (the atomic solution) or evaluate the following output in the R console and place the output in your working simulation file.

```
Attach(Design, RStudio_flags = TRUE)
```

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation, Generate

Examples

```
Design <- createDesign(N1=c(10,20),
                       N2=c(10,20),
                       sd=c(1,2))
Design

# does not use Attach()
Generate <- function(condition, fixed_objects ) {
    # condition = single row of Design input (e.g., condition <- Design[1,])
    N1 <- condition$N1
    N2 <- condition$N2
    sd <- condition$sd

    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)),
                      DV = c(group1, group2))
    dat
}

# similar to above, but using the Attach() function instead of indexing
Generate <- function(condition, fixed_objects ) {
    Attach(condition) # N1, N2, and sd are now 'attached' and visible

    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)),
                      DV = c(group1, group2))
    dat
}

######################
# NOTE: if you're using RStudio with code diagnostics on then evaluate + add the
# following output to your source file to manually support the flagged variables

Attach(Design, RStudio_flags=TRUE)

# Below is the same example, however with false positive missing variables suppressed
# when # !diagnostics ... is added added to the source file(s)

# !diagnostics suppress=N1,N2,sd
Generate <- function(condition, fixed_objects ) {
    Attach(condition) # N1, N2, and sd are now 'attached' and visible

    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)),
                      DV = c(group1, group2))
    dat
}
```

BF_sim *Example simulation from Brown and Forsythe (1974)*

Description

Example results from the Brown and Forsythe (1974) article on robust estimators for variance ratio tests. Statistical tests are organized by columns and the unique design conditions are organized by rows. See BF_sim_alternative for an alternative form of the same simulation. Code for this simulation is available of the wiki (https://github.com/philchalmers/SimDesign/wiki).

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Brown, M. B. and Forsythe, A. B. (1974). Robust tests for the equality of variances. *Journal of the American Statistical Association, 69*(346), 364–367.

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
## Not run:
data(BF_sim)
head(BF_sim)

#Type I errors
subset(BF_sim, var_ratio == 1)

## End(Not run)
```

BF_sim_alternative *(Alternative) Example simulation from Brown and Forsythe (1974)*

Description

Example results from the Brown and Forsythe (1974) article on robust estimators for variance ratio tests. Statistical tests and distributions are organized by columns and the unique design conditions are organized by rows. See BF_sim for an alternative form of the same simulation where distributions are also included in the rows. Code for this simulation is available on the wiki (https://github.com/philchalmers/SimDesign/wiki).

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Brown, M. B. and Forsythe, A. B. (1974). Robust tests for the equality of variances. *Journal of the American Statistical Association, 69*(346), 364–367.

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
## Not run:
data(BF_sim_alternative)
head(BF_sim_alternative)

#' #Type I errors
subset(BF_sim_alternative, var_ratio == 1)

## End(Not run)
```

---

bias *Compute (relative/standardized) bias summary statistic*

Description

Computes the (relative) bias of a sample estimate from the parameter value. Accepts estimate and parameter values, as well as estimate values which are in deviation form. If relative bias is requested the `estimate` and `parameter` inputs are both required.

Usage

```
bias(
  estimate,
  parameter = NULL,
  type = "bias",
  center = mean,
  deviance = colSDs,
  abs = FALSE,
  percent = FALSE,
  unname = FALSE
)
```

Arguments

estimate	a numeric vector, matrix/data.frame, or list of parameter estimates. If a vector, the length is equal to the number of replications. If a matrix/data.frame, the number of rows must equal the number of replications. list objects will be looped over using the same rules after above after first translating the information into one-dimensional vectors and re-creating the structure upon return
parameter	a numeric scalar/vector indicating the fixed parameters. If a single value is supplied and estimate is a matrix/data.frame then the value will be recycled for each column; otherwise, each element will be associated with each respective column in the estimate input. If NULL then it will be assumed that the estimate input is in a deviation form (therefore mean(estimate)) will be returned)
type	type of bias statistic to return. Default ('bias') computes the standard bias (average difference between sample and population), 'relative' computes the relative bias statistic (i.e., divide the bias by the value in parameter; note that multiplying this by 100 gives the "percent bias" measure, or if Type I error rates (α) are supplied will result in the "percentage error"), 'abs_relative' computes the relative bias for each replication independently, takes the absolute value of each term, then computes the mean estimate, and 'standardized' computes the standardized bias estimate (standard bias divided by the standard deviation of the sample estimates)
center	function to compute the central tendency. Default uses mean, reflecting the canonical $mean(\hat{p}_i - p)$ difference, but other options can be supplied to provide more robust central tendency estimates, such as median or \(x) mean(x, trim = 0.1)
deviance	function to compute the deviance criterion, currently used when type = 'standardize'. Default uses colSDs, reflecting the canonical standard deviation of an incoming matrix object (required), but other options can be supplied to provide more robust deviation estimates, such as \(x) apply(x, 2, IQR) for the interquartile range. Note that if using an alternative center then, if relevant, this should be adjusted too
abs	logical; find the absolute bias between the parameters and estimates? This effectively just applies the abs transformation to the returned result. Default is FALSE
percent	logical; change returned result to percentage by multiplying by 100? Default is FALSE
unname	logical; apply unname to the results to remove any variable names?

Value

returns a numeric vector indicating the overall (relative/standardized) bias in the estimates

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

RMSE

Examples

```
pop <- 2
samp <- rnorm(100, 2, sd = 0.5)
bias(samp, pop)
bias(samp, pop, type = 'relative')
bias(samp, pop, type = 'standardized')
bias(samp, pop, type = 'abs_relative')

dev <- samp - pop
bias(dev)

# equivalent here
bias(mean(samp), pop)

# different center function
bias(samp, pop, center=median)  # median instead of mean

# matrix input
mat <- cbind(M1=rnorm(100, 2, sd = 0.5), M2 = rnorm(100, 2, sd = 1))
bias(mat, parameter = 2)
bias(mat, parameter = 2, type = 'relative')
bias(mat, parameter = 2, type = 'standardized')

# different parameter associated with each column
mat <- cbind(M1=rnorm(1000, 2, sd = 0.25), M2 = rnorm(1000, 3, sd = .25))
bias(mat, parameter = c(2,3))
bias(mat, parameter = c(2,3), type='relative')
bias(mat, parameter = c(2,3), type='standardized')

# same, but with data.frame
df <- data.frame(M1=rnorm(100, 2, sd = 0.5), M2 = rnorm(100, 2, sd = 1))
bias(df, parameter = c(2,2))

# parameters of the same size
parameters <- 1:10
estimates <- parameters + rnorm(10)
bias(estimates, parameters)
```

```
# relative difference dividing by the magnitude of parameters
bias(estimates, parameters, type = 'abs_relative')

# relative bias as a percentage
bias(estimates, parameters, type = 'abs_relative', percent = TRUE)

# percentage error (PE) statistic given alpha (Type I error) and EDR() result
# edr <- EDR(results, alpha = .05)
edr <- c(.04, .05, .06, .08)
bias(matrix(edr, 1L), .05, type = 'relative', percent = TRUE)
```

`bootPredict` *Compute prediction estimates for the replication size using bootstrap MSE estimates*

Description

This function computes bootstrap mean-square error estimates to approximate the sampling behavior of the meta-statistics in SimDesign's `summarise` functions. A single design condition is supplied, and a simulation with `max(Rstar)` replications is performed whereby the generate-analyse results are collected. After obtaining these replication values, the replications are further drawn from (with replacement) using the differing sizes in `Rstar` to approximate the bootstrap MSE behavior given different replication sizes. Finally, given these bootstrap estimates linear regression models are fitted using the predictor term `one_sqrtR = 1 / sqrt(Rstar)` to allow extrapolation to replication sizes not observed in `Rstar`. For more information about the method and subsequent bootstrap MSE plots, refer to Koehler, Brown, and Haneuse (2009).

Usage

```
bootPredict(
  condition,
  generate,
  analyse,
  summarise,
  fixed_objects = NULL,
  ...,
  Rstar = seq(100, 500, by = 100),
  boot_draws = 1000
)

boot_predict(...)
```

Arguments

`condition`	a `data.frame` consisting of one row from the original `design` input object used within `runSimulation`

generate	see runSimulation
analyse	see runSimulation
summarise	see runSimulation
fixed_objects	see runSimulation
...	additional arguments to be passed to runSimulation
Rstar	a vector containing the size of the bootstrap subsets to obtain. Default investigates the vector [100, 200, 300, 400, 500] to compute the respective MSE terms
boot_draws	number of bootstrap replications to draw. Default is 1000

Value

returns a list of linear model objects (via lm) for each meta-statistics returned by the summarise() function

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Koehler, E., Brown, E., & Haneuse, S. J.-P. A. (2009). On the Assessment of Monte Carlo Error in Simulation-Based Statistical Analyses. *The American Statistician, 63*, 155-162.

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
set.seed(4321)
Design <- createDesign(sigma = c(1, 2))

#------------------------------------------------------------------

Generate <- function(condition, fixed_objects) {
    dat <- rnorm(100, 0, condition$sigma)
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    CIs <- t.test(dat)$conf.int
    names(CIs) <- c('lower', 'upper')
    ret <- c(mean = mean(dat), CIs)
    ret
}

Summarise <- function(condition, results, fixed_objects) {
```

```
    ret <- c(mu_bias = bias(results[,"mean"], 0),
             mu_coverage = ECR(results[,c("lower", "upper")], parameter = 0))
    ret
}

## Not run:
# boot_predict supports only one condition at a time
out <- bootPredict(condition=Design[1L, , drop=FALSE],
    generate=Generate, analyse=Analyse, summarise=Summarise)
out # list of fitted linear model(s)

# extract first meta-statistic
mu_bias <- out$mu_bias

dat <- model.frame(mu_bias)
print(dat)

# original R metric plot
R <- 1 / dat$one_sqrtR^2
plot(R, dat$MSE, type = 'b', ylab = 'MSE', main = "Replications by MSE")

plot(MSE ~ one_sqrtR, dat, main = "Bootstrap prediction plot", xlim = c(0, max(one_sqrtR)),
     ylim = c(0, max(MSE)), ylab = 'MSE', xlab = expression(1/sqrt(R)))
beta <- coef(mu_bias)
abline(a = 0, b = beta, lty = 2, col='red')

# what is the replication value when x-axis = .02? What's its associated expected MSE?
1 / .02^2 # number of replications
predict(mu_bias, data.frame(one_sqrtR = .02)) # y-axis value

# approximately how many replications to obtain MSE = .001?
(beta / .001)^2

## End(Not run)
```

`Bradley1978` *Bradley's (1978) empirical robustness interval*

Description

Robustness interval criteria for empirical detection rate estimates and empirical coverage estimates defined by Bradley (1978). See `EDR` and `ECR` to obtain such estimates.

Usage

```
Bradley1978(
  rate,
  alpha = 0.05,
  type = "liberal",
```

```
  CI = FALSE,
  out.logical = FALSE,
  out.labels = c("conservative", "robust", "liberal"),
  unname = FALSE
)
```

Arguments

rate	(optional) numeric vector containing the empirical detection rate(s) or empirical confidence interval estimates. If supplied a character vector with elements defined in out.labels or a logical vector will be returned indicating whether the detection rate estimate is considered 'robust'. When the input is an empirical coverage rate the argument CI must be set to TRUE. If this input is missing, the interval criteria will be printed to the console
alpha	Type I error rate to evaluated (default is .05)
type	character vector indicating the type of interval classification to use. Default is 'liberal', however can be 'stringent' to use Bradley's more stringent robustness criteria
CI	logical; should this robust interval be constructed on empirical detection rates (FALSE) or empirical coverage rates (TRUE)?
out.logical	logical; should the output vector be TRUE/FALSE indicating whether the supplied empirical detection rate/CI should be considered "robust"? Default is FALSE, in which case the out.labels elements are used instead
out.labels	character vector of length three indicating the classification labels according to the desired robustness interval
unname	logical; apply unname to the results to remove any variable names?

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Bradley, J. V. (1978). Robustness? *British Journal of Mathematical and Statistical Psychology, 31*, 144-152.

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

EDR, ECR, Serlin2000

Examples

```
# interval criteria used for empirical detection rates
Bradley1978()
Bradley1978(type = 'stringent')
Bradley1978(alpha = .01, type = 'stringent')

# intervals applied to empirical detection rate estimates
edr <- c(test1 = .05, test2 = .027, test3 = .051, test4 = .076, test5 = .024)

Bradley1978(edr)
Bradley1978(edr, out.logical=TRUE) # is robust?

#####
# interval criteria used for coverage estimates

Bradley1978(CI = TRUE)
Bradley1978(CI = TRUE, type = 'stringent')
Bradley1978(CI = TRUE, alpha = .01, type = 'stringent')

# intervals applied to empirical coverage rate estimates
ecr <- c(test1 = .950, test2 = .973, test3 = .949, test4 = .924, test5 = .976)

Bradley1978(ecr, CI=TRUE)
Bradley1978(ecr, CI=TRUE, out.logical=TRUE) # is robust?
```

CC *Compute congruence coefficient*

Description

Computes the congruence coefficient, also known as an "unadjusted" correlation or Tucker's congruence coefficient.

Usage

```
CC(x, y = NULL, unname = FALSE)
```

Arguments

x	a vector or `data.frame`/`matrix` containing the variables to use. If a vector then the input y is required, otherwise the congruence coefficient is computed for all bivariate combinations
y	(optional) the second vector input to use if x is a vector
unname	logical; apply `unname` to the results to remove any variable names?

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

cor

Examples

```
vec1 <- runif(1000)
vec2 <- runif(1000)

CC(vec1, vec2)
# compare to cor()
cor(vec1, vec2)

# column input
df <- data.frame(vec1, vec2, vec3 = runif(1000))
CC(df)
cor(df)
```

---

clusterSetRNGSubStream

Set RNG sub-stream for Pierre L'Ecuyer's RngStreams

Description

Sets the sub-stream RNG state within for Pierre L'Ecuyer's (1999) algorithm. Should be used within distributed array jobs after suitable L'Ecuyer's (1999) have been distributed to each array, and each array is further defined to use multi-core processing. See clusterSetRNGStream for further information.

Usage

```
clusterSetRNGSubStream(cl, seed)
```

Arguments

cl	A cluster from the parallel package, or (if NULL) the registered cluster
seed	An integer vector of length 7 as given by .Random.seed when the L'Ecuyer-CMR RNG is in use. SeeRNG for the valid values

Value

invisible NULL

`colVars` *Form Column Standard Deviation and Variances*

Description

Form column standard deviation and variances for numeric arrays (or data frames).

Usage

```
colVars(x, na.rm = FALSE, unname = FALSE)

colSDs(x, na.rm = FALSE, unname = FALSE)
```

Arguments

x	an array of two dimensions containing numeric, complex, integer or logical values, or a numeric data frame
na.rm	logical; remove missing values in each respective column?
unname	logical; apply unname to the results to remove any variable names?

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

See Also

colMeans

Examples

```
results <- matrix(rnorm(100), ncol=4)
colnames(results) <- paste0('stat', 1:4)

colVars(results)
colSDs(results)

results[1,1] <- NA
colSDs(results)
colSDs(results, na.rm=TRUE)
colSDs(results, na.rm=TRUE, unname=TRUE)
```

createDesign *Create the simulation design object*

Description

Create a partially or fully-crossed data object reflecting the unique simulation design conditions. Each row of the returned object represents a unique simulation condition, and each column represents the named factor variables under study.

Usage

```
createDesign(
  ...,
  subset,
  fractional = NULL,
  tibble = TRUE,
  stringsAsFactors = FALSE,
  fully.crossed = TRUE
)

## S3 method for class 'Design'
print(x, list2char = TRUE, pillar.sigfig = 5, show.IDs = FALSE, ...)

## S3 method for class 'Design'
x[i, j, ..., drop = FALSE]

rbindDesign(..., keep.IDs = FALSE)
```

Arguments

`...`	comma separated list of named input objects representing the simulation factors to completely cross. Note that these arguments are passed to `expand.grid` to perform the complete crossings
`subset`	(optional) a logical vector indicating elements or rows to keep to create a partially crossed simulation design
`fractional`	a fractional design matrix returned from the `FrF2` package. Note that the order of the factor names/labels are associated with the respective `...` inputs
`tibble`	logical; return a `tibble` object instead of a `data.frame`? Default is TRUE
`stringsAsFactors`	logical; should character variable inputs be coerced to factors when building a `data.frame`? Default is FALSE
`fully.crossed`	logical; create a fully-crossed design object? Setting to `FALSE` will attempt to combine the design elements column-wise via `data.frame(...)` instead of `expand.grid(...)`
`x`	object of class `'Design'`

list2char	logical; for tibble object re-evaluate list elements as character vectors for better printing of the levels? Note that this does not change the original classes of the object, just how they are printed. Default is TRUE
pillar.sigfig	number of significant digits to print. Default is 5
show.IDs	logical; print the internally stored Design ID indicators?
i	row index
j	column index
drop	logical; drop to lower dimension class?
keep.IDs	logical; keep the internal ID variables in the Design objects? Use this when row-binding conditions that are matched with previous conditions (e.g., when using expandDesign)

Value

a tibble or data.frame containing the simulation experiment conditions to be evaluated in runSimulation

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

expandDesign

Examples

```
## Not run:

# modified example from runSimulation()

Design <- createDesign(N = c(10, 20),
                       SD = c(1, 2))
Design

# remove N=10, SD=2 row from initial definition
Design <- createDesign(N = c(10, 20),
                       SD = c(1, 2),
                       subset = !(N == 10 & SD == 2))
Design

# example with list inputs
```

```
Design <- createDesign(N = c(10, 20),
                       SD = c(1, 2),
                       combo = list(c(0,0), c(0,0,1)))
Design   # notice levels printed (not typical for tibble)
print(Design, list2char = FALSE)   # standard tibble output

Design <- createDesign(N = c(10, 20),
                       SD = c(1, 2),
                       combo = list(c(0,0), c(0,0,1)),
                       combo2 = list(c(5,10,5), c(6,7)))
Design
print(Design, list2char = FALSE)   # standard tibble output

# design without crossing (inputs taken-as is)
Design <- createDesign(N = c(10, 20),
                       SD = c(1, 2), cross=FALSE)
Design   # only 2 rows

##########

## fractional factorial example

library(FrF2)
# help(FrF2)

# 7 factors in 32 runs
fr <- FrF2(32,7)
dim(fr)
fr[1:6,]

# Create working simulation design given -1/1 combinations
fDesign <- createDesign(sample_size=c(100,200),
                        mean_diff=c(.25, 1, 2),
                        variance.ratio=c(1,4, 8),
                        equal_size=c(TRUE, FALSE),
                        dists=c('norm', 'skew'),
                        same_dists=c(TRUE, FALSE),
                        symmetric=c(TRUE, FALSE),
                        # remove same-normal combo
                        subset = !(symmetric & dists == 'norm'),
                        fractional=fr)
fDesign


## End(Not run)
```

---

descript *Compute univariate descriptive statistics*

Description

Function returns univariate data summaries for each variable supplied. For presentation purposes, discrete and continuous variables are treated separately, the former of which reflects count/proportion information while the ladder are supplied to a (customizable) list of univariate summary functions. As such, quantitative/continuous variable information is kept distinct in the output, while discrete variables (e.g., `factors` and `character` vectors) are returned by using the `discrete` argument.

Usage

```
descript(df, funs = get_descriptFuns(), discrete = FALSE)

get_descriptFuns()
```

Arguments

`df`	typically a `data.frame` or `tibble`-like structure containing the variables of interest Note that `factor` and `character` vectors will be treated as discrete observations, and by default are omitted from the computation of the quantitative descriptive statistics specified in `funs`. However, setting `discrete = TRUE` will provide count-type information for these discrete variables, in which case arguments to `funs` are ignored
`funs`	functions to apply when `discrete = FALSE`. Can be modified by the user to include or exclude further functions, however each supplied function must return a scalar. Use `get_discreteFuns()` to return the full list of functions, which may then be augmented or subsetted based on the user's requirements. Default descriptive statistic returned are: `n` number of non-missing observations `mean` mean `trim` trimmed mean (10%) `sd` standard deviation `skew` skewness (from `e1701`) `kurt` kurtosis (from `e1071`) `min` minimum `P25` 25th percentile (a.k.a., 1st/lower quartile, Q1), returned from `quantile`) `P50` median (50th percentile) `P75` 75th percentile (a.k.a, 3rd/upper quartile, Q3), returned from `quantile`) `max` maximum Note that by default the `na.rm` behavior is set to `TRUE` in each function call
`discrete`	logical; include summary statistics for `discrete` variables only? If `TRUE` then only count and proportion information for the discrete variables will be returned. For greater flexibility in creating cross-tabulated count/proportion information see `xtabs`

Details

The purpose of this function is to provide a more pipe-friendly API for selecting and subsetting variables using the `dplyr` syntax, where conditional statistics are evaluated internally using the `by` function (when multiple variables are to be summarised). As a special case, if only a single variable is being summarised then the canonical output from `dplyr::summarise` will be returned.

Conditioning: As the function is intended to support pipe-friendly code specifications, conditioning/group subset specifications are declared using `group_by` and subsequently passed to `descript`.

See Also

`summarise`, `group_by`, `xtabs`

Examples

```
library(dplyr)

data(mtcars)

if(FALSE){
  # run the following to see behavior with NA values in dataset
  mtcars[sample(1:nrow(mtcars), 3), 'cyl'] <- NA
  mtcars[sample(1:nrow(mtcars), 5), 'mpg'] <- NA
}

fmtcars <- within(mtcars, {
cyl <- factor(cyl)
am <- factor(am, labels=c('automatic', 'manual'))
vs <- factor(vs)
})

# with and without factor variables
mtcars |> descript()
fmtcars |> descript()               # factors/discrete vars omitted
fmtcars |> descript(discrete=TRUE)  # discrete variables only

# for discrete variables, xtabs() is generally nicer as cross-tabs can
# be specified explicitly (though can be cumbersome)
xtabs(~ am, fmtcars)
xtabs(~ am, fmtcars) |> prop.table()
xtabs(~ am + cyl + vs, fmtcars)
xtabs(~ am + cyl + vs, fmtcars) |> prop.table()

# usual pipe chaining
fmtcars |> select(mpg, wt) |> descript()
fmtcars |> filter(mpg > 20) |> select(mpg, wt) |> descript()

# conditioning with group_by()
fmtcars |> group_by(cyl) |> descript()
fmtcars |> group_by(cyl, am) |> descript()
fmtcars |> group_by(cyl, am) |> select(mpg, wt) |> descript()
```

```
# with single variables, typical dplyr::summarise() output returned
fmtcars |> select(mpg) |> descript()
fmtcars |> group_by(cyl) |> select(mpg) |> descript()
fmtcars |> group_by(cyl, am) |> select(mpg) |> descript()

# discrete variables also work with group_by(), though again
#  xtabs() is generally more flexible
fmtcars |> group_by(cyl) |> descript(discrete=TRUE)
fmtcars |> group_by(am) |> descript(discrete=TRUE)
fmtcars |> group_by(cyl, am) |> descript(discrete=TRUE)

# only return a subset of summary statistics
funs <- get_descriptFuns()
sfuns <- funs[c('n', 'mean', 'sd')] # subset
fmtcars |> descript(funs=sfuns) # only n, miss, mean, and sd

# add a new functions
funs2 <- c(sfuns,
           trim_20 = \(x) mean(x, trim=.2, na.rm=TRUE),
           median= \(x) median(x, na.rm=TRUE))
fmtcars |> descript(funs=funs2)
```

ECR *Compute empirical coverage rates*

Description

Computes the detection rate for determining empirical coverage rates given a set of estimated confidence intervals. Note that using `1 - ECR(CIs, parameter)` will provide the empirical detection rate. Also supports computing the average width of the CIs, which may be useful when comparing the efficiency of CI estimators.

Usage

```
ECR(
  CIs,
  parameter,
  tails = FALSE,
  CI_width = FALSE,
  complement = FALSE,
  names = NULL,
  unname = FALSE
)
```

Arguments

`CIs`	a `numeric` vector or `matrix` of confidence interval values for a given parameter value, where the first element/column indicates the lower confidence interval

and the second element/column the upper confidence interval. If a vector of length 2 is passed instead then the returned value will be either a 1 or 0 to indicate whether the parameter value was or was not within the interval, respectively. Otherwise, the input must be a matrix with an even number of columns

`parameter` a numeric scalar indicating the fixed parameter value. Alternative, a `numeric` vector object with length equal to the number of rows as `CIs` (use to compare sets of parameters at once)

`tails` logical; when TRUE returns a vector of length 2 to indicate the proportion of times the parameter was lower or higher than the supplied interval, respectively. This is mainly only useful when the coverage region is not expected to be symmetric, and therefore is generally not required. Note that `1 - sum(ECR(CIs, parameter, tails=TRUE)) == ECR(CIs, parameter)`

`CI_width` logical; rather than returning the overall coverage rate, return the average width of the CIs instead? Useful when comparing the efficiency of different CI estimators

`complement` logical; rather than computing the proportion of population parameters within the CI, return the proportion outside the advertised CI (1 - ECR = alpha). In the case where only one value is provided, which normally would return a 0 if outside the CI or 1 if inside, the values will be switched (useful when using, for example, CI tests of for the significance of parameters)

`names` an optional character vector used to name the returned object. Generally useful when more than one CI estimate is investigated at once

`unname` logical; apply `unname` to the results to remove any variable names?

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

`EDR`

Examples

```
CIs <- matrix(NA, 100, 2)
for(i in 1:100){
   dat <- rnorm(100)
   CIs[i,] <- t.test(dat)$conf.int
}
```

```
ECR(CIs, 0)
ECR(CIs, 0, tails = TRUE)
ECR(CIs, 0, complement = TRUE) # proportion outside interval

# single vector input
CI <- c(-1, 1)
ECR(CI, 0)
ECR(CI, 0, complement = TRUE)
ECR(CI, 2)
ECR(CI, 2, complement = TRUE)
ECR(CI, 2, tails = TRUE)

# parameters of the same size as CI
parameters <- 1:10
CIs <- cbind(parameters - runif(10), parameters + runif(10))
parameters <- parameters + rnorm(10)
ECR(CIs, parameters)

# average width of CIs
ECR(CIs, parameters, CI_width=TRUE)

# ECR() for multiple CI estimates in the same object
parameter <- 10
CIs <- data.frame(lowerCI_1=parameter - runif(10),
                  upperCI_1=parameter + runif(10),
                  lowerCI_2=parameter - 2*runif(10),
                  upperCI_2=parameter + 2*runif(10))
head(CIs)
ECR(CIs, parameter)
ECR(CIs, parameter, tails=TRUE)
ECR(CIs, parameter, CI_width=TRUE)

# often a good idea to provide names for the output
ECR(CIs, parameter, names = c('this', 'that'))
ECR(CIs, parameter, CI_width=TRUE, names = c('this', 'that'))
ECR(CIs, parameter, tails=TRUE, names = c('this', 'that'))
```

EDR *Compute the empirical detection/rejection rate for Type I errors and Power*

Description

Computes the detection/rejection rate for determining empirical Type I error and power rates using information from p-values.

Usage

```
EDR(p, alpha = 0.05, unname = FALSE)
```

Arguments

p	a numeric vector or matrix/data.frame of p-values from the desired statistical estimator. If a matrix, each statistic must be organized by column, where the number of rows is equal to the number of replications
alpha	the detection threshold (typical values are .10, .05, and .01). Default is .05
unname	logical; apply unname to the results to remove any variable names?

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

ECR, Bradley1978

Examples

```
rates <- numeric(100)
for(i in 1:100){
   dat <- rnorm(100)
   rates[i] <- t.test(dat)$p.value
}

EDR(rates)
EDR(rates, alpha = .01)

# multiple rates at once
rates <- cbind(runif(1000), runif(1000))
EDR(rates)
```

---

expandDesign *Expand the simulation design object for array computing*

Description

Repeat each design row the specified number of times. This is primarily used for cluster computing where jobs are distributed with batches of replications and later aggregated into a complete simulation object (see runArraySimulation and SimCollect).

Usage

```
expandDesign(Design, repeat_conditions)
```

Arguments

Design	object created by createDesign which should have its rows repeated for optimal HPC schedulers
repeat_conditions	integer vector used to repeat each design row the specified number of times. Can either be a single integer, which repeats each row this many times, or an integer vector equal to the number of total rows in the created object. This argument is useful when distributing independent row conditions to cluster computing environments, particularly with different replication information. For example, if 1000 replications in total are the target but the condition is repeated over 4 rows then only 250 replications per row would be required across the repeated conditions. See SimCollect for combining the simulation objects once complete

Value

a tibble or data.frame containing the simulation experiment conditions to be evaluated in runSimulation

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

expandReplications, createDesign, SimCollect, runArraySimulation

Examples

```
## Not run:

# repeat each row 4 times (for cluster computing)
Design <- createDesign(N = c(10, 20),
                       SD.equal = c(TRUE, FALSE))
Design4 <- expandDesign(Design, 4)
Design4

# repeat first two rows 2x and the rest 4 times (for cluster computing
#   where first two conditions are faster to execute)
```

```
Design <- createDesign(SD.equal = c(TRUE, FALSE),
                       N = c(10, 100, 1000))
Design24 <- expandDesign(Design, c(2,2,rep(4, 4)))
Design24


## End(Not run)
```

`expandReplications` *Expand the replications to match* `expandDesign`

Description

Expands the replication budget to match the `expandDesign` structure.

Usage

```
expandReplications(replications, repeat_conditions)
```

Arguments

`replications`	number of replications. Can be a scalar to reflect the same replications overall, or a vector of unequal replication budgets.
`repeat_conditions`	integer vector used to repeat each design row the specified number of times. Can either be a single integer, which repeats each row this many times, or an integer vector equal to the number of total rows in the created object.

Value

an integer vector of the replication budget matching the expanded structure in `expandDesign`

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

`expandDesign`

Examples

```
## Not run:

# repeat each row 4 times (for cluster computing)
Design <- createDesign(N = c(10, 20),
                       SD.equal = c(TRUE, FALSE))
Design4 <- expandDesign(Design, 4)
Design4

# match the replication budget. Target is 1000 replications
(replications4 <- expandReplications(1000, 4))

# hence, evaluate each row in Design4 250 times
cbind(Design4, replications4)

####
# Unequal Design intensities

Design24 <- createDesign(SD.equal = c(TRUE, FALSE),
                       N = c(10, 100, 1000))
# split first two conditions into half rows, next two conditions into quarters,
#  while N=1000 condition into tenths
expand <- c(2,2,4,4,10,10)
eDesign <- expandDesign(Design, expand)
eDesign

# target replications is R=1000 per condition
(replications24 <- expandReplications(1000, expand))
cbind(eDesign, replications24)


## End(Not run)
```

`Generate` *Generate data*

Description

Generate data from a single row in the `design` input (see `runSimulation`). R contains numerous approaches to generate data, some of which are contained in the base package, as well as in `SimDesign` (e.g., `rmgh`, `rValeMaurelli`, `rHeadrick`). However the majority can be found in external packages. See CRAN's list of possible distributions here: https://CRAN.R-project.org/view=Distributions. Note that this function technically can be omitted if the data generation is provided in the `Analyse` step, though in general this is not recommended.

Usage

```
Generate(condition, fixed_objects)
```

Arguments

condition	a single row from the design input (as a data.frame), indicating the simulation conditions
fixed_objects	object passed down from runSimulation

Details

The use of try functions is generally not required in this function because Generate is internally wrapped in a try call. Therefore, if a function stops early then this will cause the function to halt internally, the message which triggered the stop will be recorded, and Generate will be called again to obtain a different dataset. That said, it may be useful for users to throw their own stop commands if the data should be re-drawn for other reasons (e.g., an estimated model terminated correctly but the maximum number of iterations were reached).

Value

returns a single object containing the data to be analyzed (usually a vector, matrix, or data.frame), or list

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

addMissing, Attach, rmgh, rValeMaurelli, rHeadrick

Examples

```
## Not run:

generate <- function(condition, fixed_objects) {
    N1 <- condition$sample_sizes_group1
    N2 <- condition$sample_sizes_group2
    sd <- condition$standard_deviations

    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)),
                      DV = c(group1, group2))
    # just a silly example of a simulated parameter
    pars <- list(random_number = rnorm(1))

    list(dat=dat, parameters=pars)
}
```

```
# similar to above, but using the Attach() function instead of indexing
generate <- function(condition, fixed_objects) {
    Attach(condition)
    N1 <- sample_sizes_group1
    N2 <- sample_sizes_group2
    sd <- standard_deviations

    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)),
                      DV = c(group1, group2))
    dat
}

generate2 <- function(condition, fixed_objects) {
    mu <- sample(c(-1,0,1), 1)
    dat <- rnorm(100, mu)
    dat        #return simple vector (discard mu information)
}

generate3 <- function(condition, fixed_objects) {
    mu <- sample(c(-1,0,1), 1)
    dat <- data.frame(DV = rnorm(100, mu))
    dat
}


## End(Not run)
```

`GenerateIf` *Perform a test that indicates whether a given* `Generate()` *function should be executed*

Description

This function is designed to prevent specific generate function executions when the design conditions are not met. Primarily useful when the `generate` argument to `runSimulation` was input as a named list object, however should only be applied for some specific design condition (otherwise, the data generation moves to the next function in the list).

Usage

```
GenerateIf(x, condition = NULL)
```

Arguments

`x`	logical statement to evaluate. If the statement evaluates to `TRUE` then the remainder of the defined function will be evaluated

condition (optional) the current design condition. This does not need to be supplied if the expression in x evaluates to valid logical (e.g., use Attach(condition) prior to using AnalyseIf, or use with(condition, AnalyseIf(someLogicalTest)))

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

Analyse, runSimulation

Examples

```
## Not run:

# SimFunctions(nGenerate = 2)

Design <- createDesign(N=c(10,20,30), var.equal = c(TRUE, FALSE))

Generate.G1 <- function(condition, fixed_objects) {
  GenerateIf(condition$var.equal == FALSE) # only run when unequal vars
  Attach(condition)
  dat <- data.frame(DV = c(rnorm(N), rnorm(N, sd=2)),
                    IV = gl(2, N, labels=c('G1', 'G2')))
  dat
}

Generate.G2 <- function(condition, fixed_objects) {
  Attach(condition)
  dat <- data.frame(DV = rnorm(N*2), IV = gl(2, N, labels=c('G1', 'G2')))
  dat
}

# always run this analysis for each row in Design
Analyse <- function(condition, dat, fixed_objects) {
  mod <- t.test(DV ~ IV, data=dat)
  mod$p.value
}

Summarise <- function(condition, results, fixed_objects) {
  ret <- EDR(results, alpha=.05)
  ret
}
```

```
#-------------------------------------------------------------------

# append names 'Welch' and 'independent' to associated output
res <- runSimulation(design=Design, replications=1000,
                     generate=list(G1=Generate.G1, G2=Generate.G2),
                     analyse=Analyse,
                     summarise=Summarise)
res


## End(Not run)
```

---

genSeeds *Generate random seeds*

Description

Generate seeds to be passed to `runSimulation`'s `seed` input. Values are sampled from 1 to 2147483647, or are generated using L'Ecuyer-CMRG's (2002) method (returning either a list if `arrayID` is omitted, or the specific row value from this list if `arrayID` is included).

Usage

```
genSeeds(design = 1L, iseed = NULL, arrayID = NULL, old.seeds = NULL)

gen_seeds(...)
```

Arguments

`design`	design matrix that requires a unique seed per condition, or a number indicating the number of seeds to generate. Default generates one number
`iseed`	the initial `set.seed` number used to generate a sequence of independent seeds according to the L'Ecuyer-CMRG (2002) method. This is recommended whenever quality random number generation is required across similar (if not identical) simulation jobs (e.g., see `runArraySimulation`). If `arrayID` is not specified then this will return a list of the associated seed for the full `design`
`arrayID`	(optional) single integer input corresponding to the specific row in the `design` object when using the `iseed` input. This is used in functions such as `runArraySimulation` to pull out the specific seed rather than manage a complete list, and is therefore more memory efficient
`old.seeds`	(optional) vector or matrix of last seeds used in previous simulations to avoid repeating the same seed on a subsequent run. Note that this approach should be used sparingly as seeds set more frequently are more likely to correlate, and therefore provide less optimal random number behaviour (e.g., if performing a simulation on two runs to achieve 5000 * 2 = 10,000 replications this is likely reasonable, but for simulations with 100 * 2 = 200 replications this is more likely to be sub-optimal). Length must be equal to the number of rows in `design`

... does nothing

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

Examples

```
# generate 1 seed (default)
genSeeds()

# generate 5 unique seeds
genSeeds(5)

# generate from nrow(design)
design <- createDesign(factorA=c(1,2,3),
                       factorB=letters[1:3])
seeds <- genSeeds(design)
seeds

# construct new seeds that are independent from original (use this sparingly)
newseeds <- genSeeds(design, old.seeds=seeds)
newseeds

# can be done in batches too
newseeds2 <- genSeeds(design, old.seeds=cbind(seeds, newseeds))
cbind(seeds, newseeds, newseeds2) # all unique

############
# generate seeds for runArraySimulation()
(iseed <- genSeeds())  # initial seed
seed_list <- genSeeds(design, iseed=iseed)
seed_list

# expand number of unique seeds given iseed (e.g., in case more replications
# are required at a later date)
seed_list_tmp <- genSeeds(nrow(design)*2, iseed=iseed)
str(seed_list_tmp) # first 9 seeds identical to seed_list

# more usefully for HPC, extract only the seed associated with an arrayID
arraySeed.15 <- genSeeds(nrow(design)*2, iseed=iseed, arrayID=15)
arraySeed.15
```

---

getArrayID *Get job array ID (e.g., from SLURM or other HPC array distributions)*

Description

Get the array ID from an HPC array distribution job (e.g., from SLURM or from optional command line arguments). The array ID is used to index the rows in the design object in runArraySimulation. For instance, a SLURM array with 10 independent jobs might have the following shell instructions.

Usage

```
getArrayID(type = "slurm", trailingOnly = TRUE, ID.shift = 0L)
```

Arguments

type	an integer indicating the element from the result of commandArgs to extract, or a character specifying the the type of. Default is 'slurm'
trailingOnly	logical value passed to commandArgs. Only used when type is an integer
ID.shift	single integer value used to shift the array ID by a constant. Useful when there are array range limitation that must be specified in the shell files (e.g., array can only be 10000 but there are more rows in the design object). For example, if the array ID should be 10000 through 12000, but the cluster computer enviroment does not allow these indices, then including the arrange range as 1-2000 in the shell file with shift=9999 would add this constant to the detected arrayID, thereby indexing the remaining row elements in the design object

Details

#!/bin/bash -l

#SBATCH -time=00:01:00

#SBATCH -array=1-10

which names the associated jobs with the numbers 1 through 10. getArrayID() then extracts this information per array, which is used as the runArraySimulation(design, ..., arrayID = getArrayID()) to pass specific rows for the design object.

See Also

runArraySimulation

Examples

```
## Not run:

# get slurm array ID
arrayID <- getArrayID()

# get ID based on first optional argument in shell specification
arrayID <- getArrayID(type = 1)

# pass to
# runArraySimulation(design, ...., arrayID = arrayID)
```

```
# increase detected arrayID by constant 9999 (for array
    specification limitations)
 arrayID <- getArrayID(ID.shift=9999)


## End(Not run)
```

IRMSE *Compute the integrated root mean-square error*

Description

Computes the average/cumulative deviation given two continuous functions and an optional function representing the probability density function. Only one-dimensional integration is supported.

Usage

```
IRMSE(
  estimate,
  parameter,
  fn,
  density = function(theta, ...) 1,
  lower = -Inf,
  upper = Inf,
  ...
)
```

Arguments

`estimate`	a vector of parameter estimates
`parameter`	a vector of population parameters
`fn`	a continuous function where the first argument is to be integrated and the second argument is a vector of parameters or parameter estimates. This function represents a implied continuous function which uses the sample estimates or population parameters
`density`	(optional) a density function used to marginalize (i.e., average), where the first argument is to be integrated, and must be of the form `density(theta, ...)` or `density(theta, param1, param2)`, where `param1` is a placeholder name for the hyper-parameters associated with the probability density function. If omitted then the cumulative different between the respective functions will be computed instead
`lower`	lower bound to begin numerical integration from
`upper`	upper bound to finish numerical integration to
`...`	additional parameters to pass to `fnest`, `fnparam`, `density`, and `integrate`,

Details

The integrated root mean-square error (IRMSE) is of the form

$$IRMSE(\theta) = \sqrt{\int [f(\theta, \hat{\psi}) - f(\theta, \psi)]^2 g(\theta, ...)}$$

where $g(\theta, ...)$ is the density function used to marginalize the continuous sample ($f(\theta, \hat{\psi})$) and population ($f(\theta, \psi)$) functions.

Value

returns a single `numeric` term indicating the average/cumulative deviation given the supplied continuous functions

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

RMSE

Examples

```
# logistic regression function with one slope and intercept
fn <- function(theta, param) 1 / (1 + exp(-(param[1] + param[2] * theta)))

# sample and population sets
est <- c(-0.4951, 1.1253)
pop <- c(-0.5, 1)

theta <- seq(-10,10,length.out=1000)
plot(theta, fn(theta, pop), type = 'l', col='red', ylim = c(0,1))
lines(theta, fn(theta, est), col='blue', lty=2)

# cumulative result (i.e., standard integral)
IRMSE(est, pop, fn)

# integrated RMSE result by marginalizing over a N(0,1) distribution
den <- function(theta, mean, sd) dnorm(theta, mean=mean, sd=sd)

IRMSE(est, pop, fn, den, mean=0, sd=1)
```

```
# this specification is equivalent to the above
den2 <- function(theta, ...) dnorm(theta, ...)

IRMSE(est, pop, fn, den2, mean=0, sd=1)
```

`listAvailableNotifiers` *List All Available Notifiers*

Description

Automatically detects all S3 classes that have a specialized `notify()` method (like `notify.MyNotifier`) and prints them as a character vector of class names (e.g., `"PushbulletNotifier"`, `"TelegramNotifier"`).

Note that only classes defined and loaded at the time you call this function will appear. If you just created a new notifier in another file or package, ensure it's sourced/loaded first.

Usage

```
listAvailableNotifiers()
```

Value

A character vector of class names that have `notify.<ClassName>` methods.

Examples

```
## Not run:
listAvailableNotifiers()
# [1] "PushbulletNotifier" "TelegramNotifier"

## End(Not run)
```

`MAE` *Compute the mean absolute error*

Description

Computes the average absolute deviation of a sample estimate from the parameter value. Accepts estimate and parameter values, as well as estimate values which are in deviation form.

Usage

```
MAE(
  estimate,
  parameter = NULL,
  type = "MAE",
  center = mean,
  percent = FALSE,
  unname = FALSE
)
```

Arguments

estimate	a numeric vector, matrix/data.frame, or list of parameter estimates. If a vector, the length is equal to the number of replications. If a matrix/data.frame the number of rows must equal the number of replications. list objects will be looped over using the same rules after above after first translating the information into one-dimensional vectors and re-creating the structure upon return
parameter	a numeric scalar/vector or matrix indicating the fixed parameter values. If a single value is supplied and estimate is a matrix/data.frame then the value will be recycled for each column; otherwise, each element will be associated with each respective column in the estimate input. If NULL, then it will be assumed that the estimate input is in a deviation form (therefore mean(abs(estimate)) will be returned)
type	type of deviation to compute. Can be 'MAE' (default) for the mean absolute error, 'NMSE' for the normalized MAE (MAE / (max(estimate) - min(estimate))), or 'SMSE' for the standardized MAE (MAE / sd(estimate))
center	function to compute the central tendency. Default uses mean, reflecting the canonical $mean((\hat{p}_i - p)^2)$ difference, but other options can be supplied to provide more robust central tendency estimates, such as median (i.e., median-squared error) or \(x) mean(x, trim = 0.1)
percent	logical; change returned result to percentage by multiplying by 100? Default is FALSE
unname	logical; apply unname to the results to remove any variable names?

Value

returns a numeric vector indicating the overall mean absolute error in the estimates

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

RMSE

Examples

```
pop <- 1
samp <- rnorm(100, 1, sd = 0.5)
MAE(samp, pop)

dev <- samp - pop
MAE(dev)
MAE(samp, pop, type = 'NMAE')
MAE(samp, pop, type = 'SMAE')

# matrix input
mat <- cbind(M1=rnorm(100, 2, sd = 0.5), M2 = rnorm(100, 2, sd = 1))
MAE(mat, parameter = 2)

# same, but with data.frame
df <- data.frame(M1=rnorm(100, 2, sd = 0.5), M2 = rnorm(100, 2, sd = 1))
MAE(df, parameter = c(2,2))

# parameters of the same size
parameters <- 1:10
estimates <- parameters + rnorm(10)
MAE(estimates, parameters)
```

---

manageMessages *Increase the intensity or suppress the output of an observed message*

Description

Function provides more nuanced management of known message outputs messages that appear in function calls outside the front-end users control (e.g., functions written in third-party packages). Specifically, this function provides a less nuclear approach than quiet and friends, which suppresses all cat and messages raised, and instead allows for specific messages to be raised either to warnings or, even more extremely, to errors. Note that for messages that are not suppressed the order with which the output and message calls appear in the original function is not retained.

Usage

```
manageMessages(
  expr,
  allow = NULL,
```

```
  message2warning = NULL,
  message2error = NULL,
  ...
)
```

Arguments

expr	expression to be evaluated (e.g., ret <- myfun(args)). Function should either be used as a wrapper, such as manageMassages(ret <- myfun(args), ...) or ret <- manageMassages(myfun(args), ...), or more readably as a pipe, ret <- myfun(args) |> manageMassages(...)
allow	(optional) a character vector indicating messages that should still appear, while all other messages should remain suppressed. Each supplied message is matched using a grepl expression, so partial matching is supported (though more specific messages are less likely to throw false positives). If NULL, all messages will be suppressed unless they appear in message2error or message2warning
message2warning	(optional) Input can be a character vector containing messages that should probably be considered warning messages for the current application instead. Each supplied character vector element is matched using a grepl expression, so partial matching is supported (though more specific messages are less likely to throw false positives).
message2error	(optional) Input can be a character vector containing known-to-be-severe messages that should be converted to errors for the current application. See message2warning for details.
...	additional arguments passed to grepl

Value

returns the original result of eval(expr), with warning messages either left the same, increased to errors, or suppressed (depending on the input specifications)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

See Also

manageWarnings, quiet

Examples

```
## Not run:

myfun <- function(x, warn=FALSE){
   message('This function is rather chatty')
   cat("It even prints in different output forms!\n")
   message('And even at different ')
   cat(" many times!\n")
   cat("Too many messages can be annoying \n")
   if(warn)
     warning('It may even throw warnings ')
   x
}

out <- myfun(1)
out

# tell the function to shhhh
out <- quiet(myfun(1))
out

# same default behaviour as quiet(), but potential for nuance
out2 <- manageMessages(myfun(1))
identical(out, out2)

# allow some messages to still get printed
out2 <- manageMessages(myfun(1), allow = "many times!")
out2 <- manageMessages(myfun(1), allow = "This function is rather chatty")

# note: . matches single character (regex)
out2 <- manageMessages(myfun(1), allow = c("many times.",
                                           "This function is rather chatty"))

# convert specific message to warning
out3 <- manageMessages(myfun(1), message2warning = "many times!")
identical(out, out3)

# other warnings also get through
out3 <- manageMessages(myfun(1, warn=TRUE), message2warning = "times!")
identical(out, out3)

# convert message to error
manageMessages(myfun(1), message2error = "m... times!")

# multiple message intensity changes
manageMessages(myfun(1),
  message2warning = "It even prints in different output forms",
  message2error = "many times!")

manageMessages(myfun(1),
  allow = c("This function is rather chatty",
            "Too many messages can be annoying"),
```

```
  message2warning = "It even prints in different output forms",
  message2error = "many times!")


## End(Not run)
```

`manageWarnings` *Manage specific warning messages*

Description

Function provides more nuanced management of known warning messages that appear in function calls outside the front-end users control (e.g., functions written in third-party packages). Specifically, this function provides a less nuclear approach than `suppressWarnings`, which suppresses all warning messages rather than those which are known to be innocuous to the current application, or when globally setting `options(warn=2)`, which has the opposite effect of treating all warnings messages as errors in the function executions. To avoid these two extreme behaviors, `character` vectors can instead be supplied to this function to either leave the raised warnings as-is (default behaviour), raise only specific warning messages to errors, or specify specific warning messages that can be generally be ignored (and therefore suppressed) while allowing new or yet to be discovered warnings to still be raised.

Usage

```
manageWarnings(expr, warning2error = FALSE, suppress = NULL, ...)
```

Arguments

`expr`	expression to be evaluated (e.g., ret <- `myfun(args)`). Function should either be used as a wrapper, such as `manageWarnings(ret <- myfun(args), ...)` or `ret <- manageWarnings(myfun(args), ...)`, or more readably as a pipe, `ret <- myfun(args) |> manageWarnings(...)`
`warning2error`	`logical` or `character` vector to control the conversion of warnings to errors. Setting this input to `TRUE` will treat all observed warning messages as errors (same behavior as `options(warn=2)`, though defined on a per expression basis rather than globally), while setting to `FALSE` (default) will leave all warning messages as-is, retaining the default behavior Alternatively, and more useful for specificity reasons, input can be a `character` vector containing known-to-be-severe warning messages that should be converted to errors. Each supplied `character` vector element is matched using a `grepl` expression, so partial matching is supported (though more specific messages are less likely to throw false positives).
`suppress`	a `character` vector indicating warning messages that are known to be innocuous a priori and can therefore be suppressed. Each supplied warning message is matched using a `grepl` expression, so partial matching is supported (though more specific messages are less likely to throw false positives). If `NULL`, no warning message will be suppressed

... additional arguments passed to grepl

Details

In general, global/nuclear behaviour of warning messages should be avoided as they are generally bad practice. On one extreme, when suppressing all warning messages using suppressWarnings, potentially important warning messages will become muffled, which can be problematic if the code developer has not become aware of these (now muffled) warnings. Moreover, this can become a long-term sustainability issue when third-party functions that the developer's code depends upon throw new warnings in the future as the code developer will be less likely to become aware of these newly implemented warnings.

On the other extreme, where all warning messages are turned into errors using `options(warn=2)`, innocuous warning messages can and will be (unwantingly) raised to an error. This negatively affects the logical workflow of the developer's functions, where more error messages must now be manually managed (e.g., via tryCatch), including the known to be innocuous warning messages as these will now considered as errors.

To avoid these extremes, front-end users should first make note of the warning messages that have been raised in their prior executions, and organized these messages into vectors of ignorable warnings (least severe), known/unknown warnings that should remain as warnings (even if not known by the code developer yet), and explicit warnings that ought to be considered errors for the current application (most severe). Once collected, these can be passed to the respective `warning2error` argument to increase the intensity of a specific warning raised, or to the `suppress` argument to suppress only the messages that have been deemed ignorable a priori (and therefore allowing all other warning messages to be raised).

Value

returns the original result of `eval(expr)`, with warning messages either left the same, increased to errors, or suppressed (depending on the input specifications)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

See Also

manageMessages, quiet

Examples

```
## Not run:

fun <- function(warn1=FALSE, warn2=FALSE, warn3=FALSE,
```

```
                warn_trailing = FALSE, error=FALSE){
   if(warn1) warning('Message one')
   if(warn2) warning('Message two')
   if(warn3) warning('Message three')
   if(warn_trailing) warning(sprintf('Message with lots of random trailings: %s',
                             paste0(sample(letters, sample(1:20, 1)), collapse=',')))
   if(error) stop('terminate function call')
   return('Returned from fun()')
}

# normal run (no warnings or errors)
out <- fun()
out

# these are all the same
manageWarnings(out <- fun())
out <- manageWarnings(fun())
out <- fun() |> manageWarnings()

# errors treated normally
fun(error=TRUE)
fun(error=TRUE) |> manageWarnings()

# all warnings/returns treated normally by default
ret1 <- fun(warn1=TRUE)
ret2 <- fun(warn1=TRUE) |> manageWarnings()
identical(ret1, ret2)

# all warnings converted to errors (similar to options(warn=2), but local)
fun(warn1=TRUE) |> manageWarnings(warning2error=TRUE)
fun(warn2=TRUE) |> manageWarnings(warning2error=TRUE)

# Specific warnings treated as errors (others stay as warnings)
# Here, treat first warning message as error but not the second or third
ret <- fun(warn1=TRUE) # warning
ret <- fun(warn1=TRUE) |> manageWarnings("Message one")  # now error
ret <- fun(warn2=TRUE) |> manageWarnings("Message one")  # still a warning

# multiple warnings raised but not converted as they do not match criteria
fun(warn2=TRUE, warn3=TRUE)
fun(warn2=TRUE, warn3=TRUE) |> manageWarnings("Message one")

# Explicitly convert multiple warning messages, allowing others through.
#   This is generally the best use of the function's specificity
fun(warn1=TRUE, warn2=TRUE)
fun(warn1=TRUE) |>   # error given either message
        manageWarnings(c("Message one", "Message two"))
fun(warn2=TRUE) |>
       manageWarnings(c("Message one", "Message two"))

# last warning gets through (left as valid warning)
ret <- fun(warn3=TRUE) |>
            manageWarnings(c("Message one", "Message two"))
```

```
ret

# suppress warnings that have only partial matching
fun(warn_trailing=TRUE)
fun(warn_trailing=TRUE)
fun(warn_trailing=TRUE)

# partial match, therefore suppressed
fun(warn_trailing=TRUE) |>
  manageWarnings(suppress="Message with lots of random trailings: ")

# multiple suppress strings
fun(warn_trailing=TRUE) |>
  manageWarnings(suppress=c("Message with lots of random trailings: ",
                            "Suppress this too"))

# could also use .* to catch all remaining characters (finer regex control)
fun(warn_trailing=TRUE) |>
  manageWarnings(suppress="Message with lots of random trailings: .*")


############
# Combine with quiet() and suppress argument to suppress innocuous messages

fun <- function(warn1=FALSE, warn2=FALSE, warn3=FALSE, error=FALSE){
   message('This function is rather chatty')
   cat("It even prints in different output forms!\n")
   if(warn1) warning('Message one')
   if(warn2) warning('Message two')
   if(warn3) warning('Message three')
   if(error) stop('terminate function call')
   return('Returned from fun()')
}

# normal run (no warnings or errors, but messages)
out <- fun()
out <- quiet(fun()) # using "indoor voice"

# suppress all print messages and warnings (not recommended)
fun(warn2=TRUE) |> quiet()
fun(warn2=TRUE) |> quiet() |> suppressWarnings()

# convert all warning to errors, and keep suppressing messages via quiet()
fun(warn2=TRUE) |> quiet() |> manageWarnings(warning2error=TRUE)

# define tolerable warning messages (only warn1 deemed ignorable)
ret <- fun(warn1=TRUE) |> quiet() |>
  manageWarnings(suppress = 'Message one')

# all other warnings raised to an error except ignorable ones
fun(warn1=TRUE, warn2=TRUE) |> quiet() |>
  manageWarnings(warning2error=TRUE, suppress = 'Message one')
```

```
# only warn2 raised to an error explicitly (warn3 remains as warning)
ret <- fun(warn1=TRUE, warn3=TRUE) |> quiet() |>
  manageWarnings(warning2error = 'Message two',
                 suppress = 'Message one')

fun(warn1=TRUE, warn2 = TRUE, warn3=TRUE) |> quiet() |>
  manageWarnings(warning2error = 'Message two',
                 suppress = 'Message one')


############################
# Practical example, converting warning into error for model that
# failed to converged normally

 library(lavaan)

## The industrialization and Political Democracy Example
## Bollen (1989), page 332
model <- '
  # latent variable definitions
     ind60 =~ x1 + x2 + x3
     dem60 =~ y1 + a*y2 + b*y3 + c*y4
     dem65 =~ y5 + a*y6 + b*y7 + c*y8

  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60

  # residual correlations
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'

# throws a warning
fit <- sem(model, data = PoliticalDemocracy, control=list(iter.max=60))

# for a simulation study, often better to treat this as an error
fit <- sem(model, data = PoliticalDemocracy, control=list(iter.max=60)) |>
  manageWarnings(warning2error = "the optimizer warns that a solution has NOT been found!")


## End(Not run)
```

MSRSE *Compute the relative performance behavior of collections of standard errors*

Description

The mean-square relative standard error (MSRSE) compares standard error estimates to the standard deviation of the respective parameter estimates. Values close to 1 indicate that the behavior of the standard errors closely matched the sampling variability of the parameter estimates.

Usage

```
MSRSE(SE, SD, percent = FALSE, unname = FALSE)
```

Arguments

SE	a numeric scalar/vector indicating the average standard errors across the replications, or a matrix of collected standard error estimates themselves to be used to compute the average standard errors. Each column/element in this input corresponds to the column/element in SD
SD	a numeric scalar/vector indicating the standard deviation across the replications, or a matrix of collected parameter estimates themselves to be used to compute the standard deviations. Each column/element in this input corresponds to the column/element in SE
percent	logical; change returned result to percentage by multiplying by 100? Default is FALSE
unname	logical; apply unname to the results to remove any variable names?

Details

Mean-square relative standard error (MSRSE) is expressed as

$$MSRSE = \frac{E(SE(\psi)^2)}{SD(\psi)^2} = \frac{1/R * \sum_{r=1}^{R} SE(\psi_r)^2}{SD(\psi)^2}$$

where $SE(\psi_r)$ represents the estimate of the standard error at the rth simulation replication, and $SD(\psi)$ represents the standard deviation estimate of the parameters across all R replications. Note that $SD(\psi)^2$ is used, which corresponds to the variance of ψ.

Value

returns a vector of ratios indicating the relative performance of the standard error estimates to the observed parameter standard deviation. Values less than 1 indicate that the standard errors were larger than the standard deviation of the parameters (hence, the SEs are interpreted as more conservative), while values greater than 1 were smaller than the standard deviation of the parameters (i.e., more liberal SEs)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
Generate <- function(condition, fixed_objects) {
   X <- rep(0:1, each = 50)
   y <- 10 + 5 * X + rnorm(100, 0, .2)
   data.frame(y, X)
}

Analyse <- function(condition, dat, fixed_objects) {
   mod <- lm(y ~ X, dat)
   so <- summary(mod)
   ret <- c(SE = so$coefficients[,"Std. Error"],
            est = so$coefficients[,"Estimate"])
   ret
}

Summarise <- function(condition, results, fixed_objects) {
   MSRSE(SE = results[,1:2], SD = results[,3:4])
}

results <- runSimulation(replications=500, generate=Generate,
                         analyse=Analyse, summarise=Summarise)
results
```

---

nc *Auto-named Concatenation of Vector or List*

Description

This is a wrapper to the function c, however names the respective elements according to their input object name. For this reason, nesting nc() calls is not recommended (joining independent nc() calls via c() is however reasonable).

Usage

```
nc(..., use.names = FALSE, error.on.duplicate = TRUE)
```

Arguments

...	objects to be concatenated
use.names	logical indicating if names should be preserved (unlike c, default is FALSE)
error.on.duplicate	logical; if the same object name appears in the returning object should an error be thrown? Default is TRUE

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
A <- 1
B <- 2
C <- 3

names(C) <- 'LetterC'

# compare the following
c(A, B, C) # unnamed

nc(A, B, C) # named
nc(this=A, B, C) # respects override named (same as c() )
nc(this=A, B, C, use.names = TRUE) # preserve original name

## Not run:
# throws errors if names not unique
nc(this=A, this=B, C)
nc(LetterC=A, B, C, use.names=TRUE)

## End(Not run)

# poor input choice names
nc(t.test(c(1:2))$p.value, t.test(c(3:4))$p.value)

# better to explicitly provide name
nc(T1 = t.test(c(1:2))$p.value,
   T2 = t.test(c(3:4))$p.value)

# vector of unnamed inputs
A <- c(5,4,3,2,1)
B <- c(100, 200)

nc(A, B, C) # A's and B's numbered uniquely
c(A, B, C)  # compare
```

```
nc(beta=A, B, C) # replacement of object name

# retain names attributes (but append object name, when appropriate)
names(A) <- letters[1:5]
nc(A, B, C)
nc(beta=A, B, C)
nc(A, B, C, use.names=TRUE)

# mix and match if some named elements work while others do not
c( nc(A, B, use.names=TRUE), nc(C))

## Not run:
# error, 'b' appears twice
names(B) <- c('b', 'b2')
nc(A, B, C, use.names=TRUE)

## End(Not run)

# List input
A <- list(1)
B <- list(2:3)
C <- list('C')

names(C) <- 'LetterC'

# compare the following
c(A, B, C) # unnamed

nc(A, B, C) # named
nc(this=A, B, C) # respects override named (same as c() and list() )
nc(this=A, B, C, use.names = TRUE) # preserve original name
```

`new_PushbulletNotifier` *Create a Pushbullet Notifier*

Description

Constructs a notifier object for sending messages via Pushbullet. This requires a Pushbullet account, the Pushbullet application installed on both a mobile device and computer, and a properly configured JSON file (typically `~/.rpushbullet.json`, using `RPushbullet::pbSetup()`).

Usage

```
new_PushbulletNotifier(
  config_path = "~/.rpushbullet.json",
  verbose_issues = FALSE
)
```

Arguments

config_path	A character string specifying the path to the Pushbullet configuration file. Defaults to "~/.rpushbullet.json".
verbose_issues	Logical. If TRUE, includes detailed information about warnings and errors in notifications. Default is FALSE.

Details

To use RPushbullet in SimDesign, create a PushbulletNotifier object using new_PushbulletNotifier() and pass it to the notifier argument in runSimulation().

Value

An S3 object of class "PushbulletNotifier" and "Notifier".

Examples

```
# Create a Pushbullet notifier (requires a valid configuration file)
pushbullet_notifier <- new_PushbulletNotifier(verbose_issues = TRUE)
```

---

new_TelegramNotifier *Create a Telegram Notifier*

Description

Constructs a notifier object for sending messages via Telegram. Requires a valid Telegram bot token and chat ID.

Usage

```
new_TelegramNotifier(bot_token, chat_id, verbose_issues = FALSE)
```

Arguments

bot_token	A character string representing your Telegram bot token, typically something like "123456:ABC-xxxx".
chat_id	A character string or numeric representing the chat/group to send messages to.
verbose_issues	Logical. If TRUE, provides detailed information about warnings and errors in the notifications.

Details

To use send notifications over Telegram with httr in SimDesign, install httr, set set up a Telegram bot, and obtain a bot token and chat ID. For more information, see the Telegram Bots API. Then use the new_TelegramNotifier() function to create a TelegramNotifier object and pass it to the notifier argument in runSimulation().

Value

An S3 object of class "TelegramNotifier".

Examples

```
# Create a Telegram notifier (requires setting up a Telegram Bot)
telegram_notifier <- new_TelegramNotifier(bot_token = "123456:ABC-xyz", chat_id = "987654321")
```

PBA *Probabilistic Bisection Algorithm*

Description

The function PBA searches a specified interval for a root (i.e., zero) of the function f(x) with respect to its first argument. However, this function differs from deterministic cousins such as uniroot in that f may contain stochastic error components, and instead provides a Bayesian interval where the root is likely to lie. Note that it is assumed that E[f(x)] is non-decreasing in x and that the root is between the search interval (evaluated approximately when check.interval=TRUE). See Waeber, Frazier, and Henderson (2013) for details.

Usage

```
PBA(
  f.root,
  interval,
  ...,
  p = 0.6,
  integer = FALSE,
  tol = if (integer) 0.01 else 1e-04,
  maxiter = 300L,
  miniter = 100L,
  wait.time = NULL,
  f.prior = NULL,
  resolution = 10000L,
  check.interval = TRUE,
  check.interval.only = FALSE,
  verbose = interactive()
)

## S3 method for class 'PBA'
print(x, ...)

## S3 method for class 'PBA'
plot(x, type = "posterior", main = "Probabilistic Bisection Posterior", ...)
```

Arguments

f.root	noisy function for which the root is sought
interval	a vector containing the end-points of the interval to be searched for the root of the form c(lower, upper). Note that if the interval is specified as c(upper, lower), where upper > lower then it the search will be organized such that increasing the value of the root estimate will result in lower f(x) values
...	additional named arguments to be passed to f
p	assumed constant for probability of correct responses (must be > 0.5)
integer	logical; should the values of the root be considered integer or numeric? The former uses a discreet grid to track the updates, while the latter currently creates a grid with resolution points
tol	tolerance criteria for convergence based on average of the f(x) evaluations
maxiter	the maximum number of iterations (default 300)
miniter	minimum number of iterations (default 100)
wait.time	(optional) instead of terminating after specific estimate criteria are satisfied (e.g., tol), terminate after a specific wait time. Input is specified either as a numeric vector in seconds or as a character vector to be formatted by timeFormater. Note that users should increase the number of maxiter as well so that termination can occur if either the maximum iterations are satisfied or the specified wait time has elapsed (whichever occurs first)
f.prior	density function indicating the likely location of the prior (e.g., if root is within [0,1] then dunif works, otherwise custom functions will be required)
resolution	constant indicating the number of equally spaced grid points to track when integer = FALSE.
check.interval	logical; should an initial check be made to determine whether f(interval[1L]) and f(interval[2L]) have opposite signs? Default is TRUE
check.interval.only	logical; return only TRUE or FALSE to test whether there is a likely root given interval? Setting this to TRUE can be useful when you are unsure about the root location interval and may want to use a higher replication input from SimSolve
verbose	logical; should the iterations and estimate be printed to the console?
x	an object of class PBA
type	type of plot to draw for PBA object. Can be either 'posterior' or 'history' to plot the PBA posterior distribution or the mediation iteration history
main	plot title

References

Horstein, M. (1963). Sequential transmission using noiseless feedback. IEEE Trans. Inform. Theory, 9(3):136-143.

Waeber, R., Frazier, P. I. & Henderson, S. G. (2013). Bisection Search with Noisy Responses. SIAM Journal on Control and Optimization, Society for Industrial & Applied Mathematics (SIAM), 51, 2261-2279.

See Also

uniroot, RobbinsMonro

Examples

```
# find x that solves f(x) - b = 0 for the following
f.root <- function(x, b = .6) 1 / (1 + exp(-x)) - b
f.root(.3)

xs <- seq(-3,3, length.out=1000)
plot(xs, f.root(xs), type = 'l', ylab = "f(x)", xlab='x', las=1)
abline(h=0, col='red')

retuni <- uniroot(f.root, c(0,1))
retuni
abline(v=retuni$root, col='blue', lty=2)

# PBA without noisy root
retpba <- PBA(f.root, c(0,1))
retpba
retpba$root
plot(retpba)
plot(retpba, type = 'history')

# Same problem, however root function is now noisy. Hence, need to solve
#  fhat(x) - b + e = 0, where E(e) = 0
f.root_noisy <- function(x) 1 / (1 + exp(-x)) - .6 + rnorm(1, sd=.02)
sapply(rep(.3, 10), f.root_noisy)

# uniroot "converges" unreliably
set.seed(123)
uniroot(f.root_noisy, c(0,1))$root
uniroot(f.root_noisy, c(0,1))$root
uniroot(f.root_noisy, c(0,1))$root

# probabilistic bisection provides better convergence
retpba.noise <- PBA(f.root_noisy, c(0,1))
retpba.noise
plot(retpba.noise)
plot(retpba.noise, type = 'history')

## Not run:
# ignore termination criteria and instead run for 30 seconds or 50000 iterations
retpba.noise_30sec <- PBA(f.root_noisy, c(0,1), wait.time = "0:30", maxiter=50000)
retpba.noise_30sec


## End(Not run)
```

quiet *Suppress verbose function messages*

Description

This function is used to suppress information printed from external functions that make internal use of message and cat, which provide information in interactive R sessions. For simulations, the session is not interactive, and therefore this type of output should be suppressed. For similar behaviour for suppressing warning messages, see manageWarnings.

Usage

```
quiet(..., cat = TRUE, keep = FALSE, attr.name = "quiet.messages")
```

Arguments

...	the functional expression to be evaluated
cat	logical; also capture calls from cat? If FALSE only message will be suppressed
keep	logical; return a character vector of the messages/concatenate and print strings as an attribute to the resulting object from expr(...)?
attr.name	attribute name to use when keep = TRUE

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

manageWarnings

Examples

```
myfun <- function(x, warn=FALSE){
   message('This function is rather chatty')
   cat("It even prints in different output forms!\n")
   message('And even at different....')
   cat("...times!\n")
   if(warn)
     warning('It may even throw warnings!')
   x
}

out <- myfun(1)
```

```
out

# tell the function to shhhh
out <- quiet(myfun(1))
out

# which messages are suppressed? Extract stored attribute
out <- quiet(myfun(1), keep = TRUE)
attr(out, 'quiet.messages')

# Warning messages still get through (see manageWarnings(suppress)
#  for better alternative than using suppressWarnings())
out2 <- myfun(2, warn=TRUE) |> quiet() # warning gets through
out2

# suppress warning message explicitly, allowing others to be raised if present
myfun(2, warn=TRUE) |> quiet() |>
   manageWarnings(suppress='It may even throw warnings!')
```

RAB *Compute the relative absolute bias of multiple estimators*

Description

Computes the relative absolute bias given the bias estimates for multiple estimators.

Usage

```
RAB(x, percent = FALSE, unname = FALSE)
```

Arguments

x	a numeric vector of bias estimates (see bias), where the first element will be used as the reference
percent	logical; change returned result to percentage by multiplying by 100? Default is FALSE
unname	logical; apply unname to the results to remove any variable names?

Value

returns a vector of absolute bias ratios indicating the relative bias effects compared to the first estimator. Values less than 1 indicate better bias estimates than the first estimator, while values greater than 1 indicate worse bias than the first estimator

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
pop <- 1
samp1 <- rnorm(5000, 1)
bias1 <- bias(samp1, pop)
samp2 <- rnorm(5000, 1)
bias2 <- bias(samp2, pop)

RAB(c(bias1, bias2))
RAB(c(bias1, bias2), percent = TRUE) # as a percentage
```

---

rbind.SimDesign *Combine two separate SimDesign objects by row*

Description

This function combines two Monte Carlo simulations executed by SimDesign's runSimulation function which, for all intents and purposes, could have been executed in a single run. This situation arises when a simulation has been completed, however the Design object was later modified to include more levels in the defined simulation factors. Rather than re-executing the previously completed simulation combinations, only the new combinations need to be evaluated into a different object and then rbind together to create the complete object combinations.

Usage

```
## S3 method for class 'SimDesign'
rbind(...)
```

Arguments

...	two or more SimDesign objects that should be combined by rows

Value

same object that is returned by runSimulation

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
## Not run:

# modified example from runSimulation()

Design <- createDesign(N = c(10, 20),
                       SD = c(1, 2))

Generate <- function(condition, fixed_objects) {
    dat <- with(condition, rnorm(N, 10, sd=SD))
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    ret <- mean(dat) # mean of the sample data vector
    ret
}

Summarise <- function(condition, results, fixed_objects) {
    ret <- c(mu=mean(results), SE=sd(results)) # mean and SD summary of the sample means
    ret
}

Final1 <- runSimulation(design=Design, replications=1000,
                        generate=Generate, analyse=Analyse, summarise=Summarise)
Final1

###
# later decide that N = 30 should have also been investigated. Rather than
# running the following object ....
newDesign <- createDesign(N = c(10, 20, 30),
                          SD = c(1, 2))

# ... only the new subset levels are executed to save time
subDesign <- subset(newDesign, N == 30)
subDesign

Final2 <- runSimulation(design=subDesign, replications=1000,
                        generate=Generate, analyse=Analyse, summarise=Summarise)
Final2

# glue results together by row into one object as though the complete 'Design'
# object were run all at once
```

```
Final <- rbind(Final1, Final2)
Final

summary(Final)


## End(Not run)
```

RD *Compute the relative difference*

Description

Computes the relative difference statistic of the form `(est - pop)/ pop`, which is equivalent to the form `est/pop - 1`. If matrices are supplied then an equivalent matrix variant will be used of the form `(est - pop) * solve(pop)`. Values closer to 0 indicate better relative parameter recovery. Note that for single variable inputs this is equivalent to `bias(..., type = 'relative')`.

Usage

```
RD(est, pop, as.vector = TRUE, unname = FALSE)
```

Arguments

`est`	a `numeric` vector, `matrix/data.frame`, or `list` containing the parameter estimates
`pop`	a `numeric` vector or matrix containing the true parameter values. Must be of comparable dimension to `est`
`as.vector`	logical; always wrap the result in a `as.vector` function before returning?
`unname`	logical; apply `unname` to the results to remove any variable names?

Value

returns a `vector` or `matrix` depending on the inputs and whether `as.vector` was used

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
# vector
pop <- seq(1, 100, length.out=9)
est1 <- pop + rnorm(9, 0, .2)
(rds <- RD(est1, pop))
summary(rds)

# matrix
pop <- matrix(c(1:8, 10), 3, 3)
est2 <- pop + rnorm(9, 0, .2)
RD(est2, pop, as.vector = FALSE)
(rds <- RD(est2, pop))
summary(rds)
```

RE *Compute the relative efficiency of multiple estimators*

Description

Computes the relative efficiency given the RMSE (default) or MSE values for multiple estimators.

Usage

```
RE(x, MSE = FALSE, percent = FALSE, unname = FALSE)
```

Arguments

x	a numeric vector of root mean square error values (see RMSE), where the first element will be used as the reference. Otherwise, the object could contain MSE values if the flag MSE = TRUE is also included
MSE	logical; are the input value mean squared errors instead of root mean square errors?
percent	logical; change returned result to percentage by multiplying by 100? Default is FALSE
unname	logical; apply unname to the results to remove any variable names?

Value

returns a vector of variance ratios indicating the relative efficiency compared to the first estimator. Values less than 1 indicate better efficiency than the first estimator, while values greater than 1 indicate worse efficiency than the first estimator

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
pop <- 1
samp1 <- rnorm(100, 1, sd = 0.5)
RMSE1 <- RMSE(samp1, pop)
samp2 <- rnorm(100, 1, sd = 1)
RMSE2 <- RMSE(samp2, pop)

RE(c(RMSE1, RMSE2))
RE(c(RMSE1, RMSE2), percent = TRUE) # as a percentage

# using MSE instead
mse <- c(RMSE1, RMSE2)^2
RE(mse, MSE = TRUE)
```

---

rejectionSampling *Rejection sampling (i.e., accept-reject method)*

Description

This function supports the rejection sampling (i.e., accept-reject) approach to drawing values from seemingly difficult (probability) density functions by sampling values from more manageable proxy distributions.

Usage

```
rejectionSampling(
  n,
  df,
  dg,
  rg,
  M,
  method = "optimize",
  interval = NULL,
  logfuns = FALSE,
  maxM = 1e+05,
  parstart = rg(1L),
  ESRS_Mstart = 1.0001
)
```

Arguments

n	number of samples to draw
df	the desired (potentially un-normed) density function to draw independent samples from. Must be in the form of a function with a single input corresponding to the values sampled from rg. Function is assumed to be vectorized (if not, see Vectorize)
dg	the proxy (potentially un-normed) density function to draw samples from in lieu of drawing samples from df. The support for this density function should be the same as df (i.e., when df(x) > 0 then dg(x) > 0). Must be in the form of a function with a single input corresponding to the values sampled from rg. Function is assumed to be vectorized (if not, see Vectorize)
rg	the proxy random number generation function, associated with dg, used to draw proposal samples from. Must be in the form of a function with a single input corresponding to the number of values to draw, while the output can either be a vector or a matrix (if a matrix, each independent observation must be stored in a unique row). Function is assumed to be vectorized (if not, see Vectorize)
M	the upper-bound of the ratio of probability density functions to help minimize the number of discarded draws and define the corresponding rescaled proposal envelope. When missing, M is computed internally by finding a reasonable maximum of log(df(x)) - log(dg(x)), and this value is returned to the console. When both df and dg are true probability density functions (i.e., integrate to 1) the acceptance probability is equal to 1/M
method	when M is missing, the optimization of M is done either by finding the mode of the log-density values ("optimize") or by using the "Empirical Supremum Rejection Sampling" method ("ESRS")
interval	when M is missing, for univariate density function draws, the interval to search within via optimize. If not specified, a sample of 5000 values from the rg function definition will be collected, and the min/max will be obtained via this random sample
logfuns	logical; have the df and dg function been written so as to return log-densities instead of the original densities? The FALSE default assumes the original densities are returned (use TRUE when higher accuracy is required when generating each density definition)
maxM	logical; if when optimizing M the value is greater than this cut-off then stop; ampler would likelihood be too efficient, or optimization is failing
parstart	starting value vector for optimization of M in multidimensional distributions
ESRS_Mstart	starting M value for the ESRS algorithm

Details

The accept-reject algorithm is a flexible approach to obtaining i.i.d.'s from a difficult to sample from (probability) density function where either the transformation method fails or inverse transform method is difficult to manage. The algorithm does so by sampling from a more "well-behaved" proxy distribution (with identical support, up to some proportionality constant M that reshapes the proposal density to envelope the target density), and accepts the draws if they are likely within

the target density. Hence, the closer the shape of `dg(x)` is to the desired `df(x)`, the more likely the draws are to be accepted; otherwise, many iterations of the accept-reject algorithm may be required, which decreases the computational efficiency.

Value

returns a vector or matrix of draws (corresponding to the output class from `rg`) from the desired `df`

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Caffo, B. S., Booth, J. G., and Davison, A. C. (2002). Empirical supremum rejection sampling. Biometrika, 89, 745–754.

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
## Not run:

# Generate X ~ beta(a,b), where a and b are a = 2.7 and b = 6.3,
# and the support is Y ~ Unif(0,1)
dfn <- function(x) dbeta(x, shape1 = 2.7, shape2 = 6.3)
dgn <- function(x) dunif(x, min = 0, max = 1)
rgn <- function(n) runif(n, min = 0, max = 1)

# when df and dg both integrate to 1, acceptance probability = 1/M
M <- rejectionSampling(df=dfn, dg=dgn, rg=rgn)
M
dat <- rejectionSampling(10000, df=dfn, dg=dgn, rg=rgn, M=M)
hist(dat, 100)
hist(rbeta(10000, 2.7, 6.3), 100) # compare

# obtain empirical estimate of M via ESRS method
M <- rejectionSampling(1000, df=dfn, dg=dgn, rg=rgn, method='ESRS')
M

# generate using better support function (here, Y ~ beta(2,6)),
#   and use log setup in initial calls (more numerically accurate)
dfn <- function(x) dbeta(x, shape1 = 2.7, shape2 = 6.3, log = TRUE)
dgn <- function(x) dbeta(x, shape1 = 2, shape2 = 6, log = TRUE)
rgn <- function(n) rbeta(n, shape1 = 2, shape2 = 6)
M <- rejectionSampling(df=dfn, dg=dgn, rg=rgn, logfuns=TRUE) # better M
M
```

```
## Alternative estimation of M
## M <- rejectionSampling(10000, df=dfn, dg=dgn, rg=rgn, logfuns=TRUE,
##                        method='ESRS')
dat <- rejectionSampling(10000, df=dfn, dg=dgn, rg=rgn, M=M, logfuns=TRUE)
hist(dat, 100)

#-------------------------------------------------------
# sample from wonky (and non-normalized) density function, like below
dfn <- function(x){
    ret <- numeric(length(x))
    ret[x <= .5] <- dnorm(x[x <= .5])
    ret[x > .5] <-  dnorm(x[x > .5]) + dchisq(x[x > .5], df = 2)
    ret
}
y <- seq(-5,5, length.out = 1000)
plot(y, dfn(y), type = 'l', main = "Function to sample from")

# choose dg/rg functions that have support within the range [-inf, inf]
rgn <- function(n) rnorm(n, sd=4)
dgn <- function(x) dnorm(x, sd=4)

## example M height from above graphic
##  (M selected using ESRS to help stochastically avoid local mins)
M <- rejectionSampling(10000, df=dfn, dg=dgn, rg=rgn, method='ESRS')
M
lines(y, dgn(y)*M, lty = 2)
dat <- rejectionSampling(10000, df=dfn, dg=dgn, rg=rgn, M=M)
hist(dat, 100, prob=TRUE)

# true density (normalized)
C <- integrate(dfn, -Inf, Inf)$value
ndfn <- function(x) dfn(x) / C
curve(ndfn, col='red', lwd=2, add=TRUE)


#-------------------------------------------------------
# multivariate distribution
dfn <- function(x) sum(log(c(dnorm(x[1]) + dchisq(x[1], df = 5),
                   dnorm(x[2], -1, 2))))
rgn <- function(n) c(rnorm(n, sd=3), rnorm(n, sd=3))
dgn <- function(x) sum(log(c(dnorm(x[1], sd=3), dnorm(x[1], sd=3))))

# M <- rejectionSampling(df=dfn, dg=dgn, rg=rgn, logfuns=TRUE)
dat <- rejectionSampling(5000, df=dfn, dg=dgn, rg=rgn, M=4.6, logfuns=TRUE)
hist(dat[,1], 30)
hist(dat[,2], 30)
plot(dat)



## End(Not run)
```

`reSummarise` *Run a summarise step for results that have been saved to the hard drive*

Description

When `runSimulation()` uses the option `save_results = TRUE` the R replication results from the Generate-Analyse functions are stored to the hard drive. As such, additional summarise components may be required at a later time, whereby the respective `.rds` files must be read back into R to be summarised. This function performs the reading of these files, application of a provided summarise function, and final collection of the respective results.

Usage

```
reSummarise(
  summarise,
  dir = NULL,
  files = NULL,
  results = NULL,
  Design = NULL,
  fixed_objects = NULL,
  boot_method = "none",
  boot_draws = 1000L,
  CI = 0.95,
  prefix = "results-row"
)
```

Arguments

`summarise`	a summarise function to apply to the read-in files. See `runSimulation` for details. Note that if the simulation contained only one row then the new summarise function can be defined as either `summarise <- function(results, fixed_objects)`, if `fixed_objects` is required, or `summarise <- function(results)`,
`dir`	directory pointing to the .rds files to be read-in that were saved from `runSimulation(..., save_results=TRUE)`. If `NULL`, it is assumed the current working directory contains the .rds files
`files`	(optional) names of files to read-in. If `NULL` all files located within `dir` will be used
`results`	(optional) the results of `runSimulation` when no `summarise` function was provided. Can be either a `tibble` or `matrix` (indicating that exactly one design condition was evaluated), or a `list` of `matrix`/`tibble` objects indicating that multiple conditions were performed with no summarise evaluation. Alternatively, if `store_results = TRUE` in the `runSimulation()` execution then the final SimDesign object may be passed, where the generate-analyse information will be extracted from the object instead

Design	(optional) if results input used, and design condition information important in the summarise step, then the original design object from runSimulation should be included
fixed_objects	(optional) see runSimulation for details
boot_method	method for performing non-parametric bootstrap confidence intervals for the respective meta-statistics computed by the Summarise function. See runSimulation for details
boot_draws	number of non-parametric bootstrap draws to sample for the summarise function after the generate-analyse replications are collected. Default is 1000
CI	bootstrap confidence interval level (default is 95%)
prefix	character indicating prefix used for stored files

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
Design <- createDesign(N = c(10, 20, 30))

Generate <- function(condition, fixed_objects) {
    dat <- with(condition, rnorm(N, 10, 5)) # distributed N(10, 5)
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    ret <- c(mean=mean(dat), median=median(dat)) # mean/median of sample data
    ret
}

Summarise <- function(condition, results, fixed_objects){
    colMeans(results)
}

## Not run:
# run the simulation
runSimulation(design=Design, replications=50,
              generate=Generate, analyse=Analyse,
              summarise=Summarise, save_results=TRUE,
              save_details = list(save_results_dirname='simresults'))
```

```
res <- reSummarise(Summarise, dir = 'simresults/')
res

Summarise2 <- function(condition, results, fixed_objects){
    ret <- c(mean_ests=colMeans(results), SE=colSDs(results))
    ret
}

res2 <- reSummarise(Summarise2, dir = 'simresults/')
res2

SimClean(dir='simresults/')


## End(Not run)

###
# Similar, but with results stored within the final object

res <- runSimulation(design=Design, replications=50, store_results = TRUE,
                     generate=Generate, analyse=Analyse, summarise=Summarise)
res

# same summarise but with bootstrapping
res2 <- reSummarise(Summarise, results = res, boot_method = 'basic')
res2
```

---

rHeadrick *Generate non-normal data with Headrick's (2002) method*

Description

Generate multivariate non-normal distributions using the fifth-order polynomial method described by Headrick (2002).

Usage

```
rHeadrick(
  n,
  mean = rep(0, nrow(sigma)),
  sigma = diag(length(mean)),
  skew = rep(0, nrow(sigma)),
  kurt = rep(0, nrow(sigma)),
  gam3 = NaN,
  gam4 = NaN,
  return_coefs = FALSE,
  coefs = NULL,
```

```
  control = list(trace = FALSE, max.ntry = 15, obj.tol = 1e-10, n.valid.sol = 1)
)
```

Arguments

n	number of samples to draw
mean	a vector of k elements for the mean of the variables
sigma	desired k x k covariance matrix between bivariate non-normal variables
skew	a vector of k elements for the skewness of the variables
kurt	a vector of k elements for the kurtosis of the variables
gam3	(optional) explicitly supply the gamma 3 value? Default computes this internally
gam4	(optional) explicitly supply the gamma 4 value? Default computes this internally
return_coefs	logical; return the estimated coefficients only? See below regarding why this is useful.
coefs	(optional) supply previously estimated coefficients? This is useful when there must be multiple data sets drawn and will avoid repetitive computations. Must be the object returned after passing return_coefs = TRUE
control	a list of control parameters when locating the polynomial coefficients

Details

This function is primarily a wrapper for the code written by Oscar L. Olvera Astivia (last edited Feb 26, 2015) with some modifications (e.g., better starting values for the Newton optimizer, passing previously saved coefs, etc).

Author(s)

Oscar L. Olvera Astivia and Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Headrick, T. C. (2002). Fast fifth-order polynomial transforms for generating univariate and multivariate nonnormal distributions. *Computational Statistics & Data Analysis, 40*, 685-711.

Olvera Astivia, O. L., & Zumbo, B. D. (2015). A Cautionary Note on the Use of the Vale and Maurelli Method to Generate Multivariate, Nonnormal Data for Simulation Purposes. *Educational and Psychological Measurement, 75*, 541-567.

Examples

```
## Not run:
set.seed(1)

N <- 200
mean <- c(rep(0,4))
Sigma <- matrix(.49, 4, 4)
diag(Sigma) <- 1
skewness <- c(rep(1,4))
kurtosis <- c(rep(2,4))

nonnormal <- rHeadrick(N, mean, Sigma, skewness, kurtosis)
cor(nonnormal) |> round(3)
descript(nonnormal)

#-----------
# compute the coefficients, then supply them back to the function to avoid
# extra computations

cfs <- rHeadrick(N, mean, Sigma, skewness, kurtosis, return_coefs = TRUE)
cfs

# compare
system.time(nonnormal <- rHeadrick(N, mean, Sigma, skewness, kurtosis))
system.time(nonnormal <- rHeadrick(N, mean, Sigma, skewness, kurtosis,
                                   coefs=cfs))

## End(Not run)
```

`rint` *Generate integer values within specified range*

Description

Efficiently generate positive and negative integer values with (default) or without replacement. This function is mainly a wrapper to the `sample.int` function (which itself is much more efficient integer sampler than the more general `sample`), however is intended to work with both positive and negative integer ranges since `sample.int` only returns positive integer values that must begin at `1L`.

Usage

```
rint(n, min, max, replace = TRUE, prob = NULL)
```

Arguments

`n`	number of samples to draw
`min`	lower limit of the distribution. Must be finite

max	upper limit of the distribution. Must be finite
replace	should sampling be with replacement?
prob	a vector of probability weights for obtaining the elements of the vector being sampled

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
set.seed(1)

# sample 1000 integer values within 20 to 100
x <- rint(1000, min = 20, max = 100)
summary(x)

# sample 1000 integer values within 100 to 10 billion
x <- rint(1000, min = 100, max = 1e8)
summary(x)

# compare speed to sample()
system.time(x <- rint(1000, min = 100, max = 1e8))
system.time(x2 <- sample(100:1e8, 1000, replace = TRUE))

# sample 1000 integer values within -20 to 20
x <- rint(1000, min = -20, max = 20)
summary(x)
```

---

rinvWishart *Generate data with the inverse Wishart distribution*

Description

Function generates data in the form of symmetric matrices from the inverse Wishart distribution given a covariance matrix and degrees of freedom.

Usage

```
rinvWishart(n = 1, df, sigma)
```

Arguments

n	number of matrix observations to generate. By default n = 1, which returns a single symmetric matrix. If n > 1 then a list of n symmetric matrices are returned instead
df	degrees of freedom
sigma	positive definite covariance matrix

Value

a numeric matrix with columns equal to ncol(sigma) when n = 1, or a list of n matrices with the same properties

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
# random inverse Wishart matrix given variances [3,6], covariance 2, and df=15
sigma <- matrix(c(3,2,2,6), 2, 2)
x <- rinvWishart(sigma = sigma, df = 15)
x

# list of matrices
x <- rinvWishart(20, sigma = sigma, df = 15)
x
```

rmgh *Generate data with the multivariate g-and-h distribution*

Description

Generate non-normal distributions using the multivariate g-and-h distribution. Can be used to generate several different classes of univariate and multivariate distributions.

Usage

```
rmgh(n, g, h, mean = rep(0, length(g)), sigma = diag(length(mean)))
```

Arguments

n	number of samples to draw
g	the g parameter(s) which control the skew of a distribution in terms of both direction and magnitude
h	the h parameter(s) which control the tail weight or elongation of a distribution and is positively related with kurtosis
mean	a vector of k elements for the mean of the variables
sigma	desired k x k covariance matrix between bivariate non-normal variables

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
set.seed(1)

# univariate
norm <- rmgh(10000,1e-5,0)
hist(norm)

skew <- rmgh(10000,1/2,0)
hist(skew)

neg_skew_platykurtic <- rmgh(10000,-1,-1/2)
hist(neg_skew_platykurtic)
```

```
# multivariate
sigma <- matrix(c(2,1,1,4), 2)
mean <- c(-1, 1)
twovar <- rmgh(10000, c(-1/2, 1/2), c(0,0),
    mean=mean, sigma=sigma)
hist(twovar[,1])
hist(twovar[,2])
plot(twovar)
```

RMSE *Compute the (normalized) root mean square error*

Description

Computes the average deviation (root mean square error; also known as the root mean square deviation) of a sample estimate from the parameter value. Accepts estimate and parameter values, as well as estimate values which are in deviation form.

Usage

```
RMSE(
  estimate,
  parameter = NULL,
  type = "RMSE",
  center = mean,
  MSE = FALSE,
  percent = FALSE,
  unname = FALSE
)

RMSD(
  estimate,
  parameter = NULL,
  type = "RMSE",
  center = mean,
  MSE = FALSE,
  percent = FALSE,
  unname = FALSE
)
```

Arguments

`estimate`	a `numeric` vector, `matrix`/`data.frame`, or `list` of parameter estimates. If a vector, the length is equal to the number of replications. If a `matrix`/`data.frame`, the number of rows must equal the number of replications. `list` objects will be looped over using the same rules after above after first translating the information into one-dimensional vectors and re-creating the structure upon return

parameter	a numeric scalar/vector indicating the fixed parameter values. If a single value is supplied and estimate is a matrix/data.frame then the value will be recycled for each column; otherwise, each element will be associated with each respective column in the estimate input. If NULL then it will be assumed that the estimate input is in a deviation form (therefore sqrt(mean(estimate^2)) will be returned)
type	type of deviation to compute. Can be 'RMSE' (default) for the root mean square-error, 'NRMSE' for the normalized RMSE (RMSE / (max(estimate) - min(estimate))), 'SRMSE' for the standardized RMSE (RMSE / sd(estimate)), 'CV' for the coefficient of variation, or 'RMSLE' for the root mean-square log-error
center	function to compute the central tendency. Default uses mean, reflecting the canonical $mean((\hat{p}_i - p)^2)$ difference (and subsequently square-rooted), but other options can be supplied to provide more robust central tendency estimates, such as median (i.e., root median-squared error) or \(x) mean(x, trim = 0.1). Note that the centering function is also used in the denominator of type = 'CV' and 'RMSLE'
MSE	logical; return the mean square error equivalent of the results instead of the root mean-square error (in other words, the result is squared)? Default is FALSE
percent	logical; change returned result to percentage by multiplying by 100? Default is FALSE
unname	logical; apply unname to the results to remove any variable names?

Value

returns a numeric vector indicating the overall average deviation in the estimates

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

bias

MAE

Examples

```
pop <- 1
samp <- rnorm(100, 1, sd = 0.5)
RMSE(samp, pop)
```

```
dev <- samp - pop
RMSE(dev)

RMSE(samp, pop, type = 'NRMSE')
RMSE(dev, type = 'NRMSE')
RMSE(dev, pop, type = 'SRMSE')
RMSE(samp, pop, type = 'CV')
RMSE(samp, pop, type = 'RMSLE')

# percentage reported
RMSE(samp, pop, type = 'NRMSE')
RMSE(samp, pop, type = 'NRMSE', percent = TRUE)

# matrix input
mat <- cbind(M1=rnorm(100, 2, sd = 0.5), M2 = rnorm(100, 2, sd = 1))
RMSE(mat, parameter = 2)
RMSE(mat, parameter = c(2, 3))

# different parameter associated with each column
mat <- cbind(M1=rnorm(1000, 2, sd = 0.25), M2 = rnorm(1000, 3, sd = .25))
RMSE(mat, parameter = c(2,3))

# same, but with data.frame
df <- data.frame(M1=rnorm(100, 2, sd = 0.5), M2 = rnorm(100, 2, sd = 1))
RMSE(df, parameter = c(2,2))

# parameters of the same size
parameters <- 1:10
estimates <- parameters + rnorm(10)
RMSE(estimates, parameters)
```

`rmvnorm` *Generate data with the multivariate normal (i.e., Gaussian) distribution*

Description

Function generates data from the multivariate normal distribution given some mean vector and/or covariance matrix.

Usage

```
rmvnorm(n, mean = rep(0, nrow(sigma)), sigma = diag(length(mean)))
```

Arguments

`n`	number of observations to generate
`mean`	mean vector, default is `rep(0, length = ncol(sigma))`
`sigma`	positive definite covariance matrix, default is `diag(length(mean))`

Value

a numeric matrix with columns equal to `length(mean)`

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
# random normal values with mean [5, 10] and variances [3,6], and covariance 2
sigma <- matrix(c(3,2,2,6), 2, 2)
mu <- c(5,10)
x <- rmvnorm(1000, mean = mu, sigma = sigma)
head(x)
summary(x)
plot(x[,1], x[,2])
```

---

rmvt *Generate data with the multivariate t distribution*

Description

Function generates data from the multivariate t distribution given a covariance matrix, non-centrality parameter (or mode), and degrees of freedom.

Usage

```
rmvt(n, sigma, df, delta = rep(0, nrow(sigma)), Kshirsagar = FALSE)
```

Arguments

n	number of observations to generate
sigma	positive definite covariance matrix
df	degrees of freedom. df = 0 and df = Inf corresponds to the multivariate normal distribution
delta	the vector of non-centrality parameters of length n which specifies the either the modes (default) or non-centrality parameters
Kshirsagar	logical; triggers whether to generate data with non-centrality parameters or to adjust the simulated data to the mode of the distribution. The default uses the mode

Value

a numeric matrix with columns equal to ncol(sigma)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
# random t values given variances [3,6], covariance 2, and df = 15
sigma <- matrix(c(3,2,2,6), 2, 2)
x <- rmvt(1000, sigma = sigma, df = 15)
head(x)
summary(x)
plot(x[,1], x[,2])
```

RobbinsMonro *Robbins-Monro (1951) stochastic root-finding algorithm*

Description

Function performs stochastic root solving for the provided `f(x)` using the Robbins-Monro (1951) algorithm. Differs from deterministic cousins such as `uniroot` in that `f` may contain stochastic error components, where the root is obtained through the running average method provided by noise filter (see also `PBA`). Assumes that `E[f(x)]` is non-decreasing in `x`.

Usage

```
RobbinsMonro(
  f,
  p,
  ...,
  Polyak_Juditsky = FALSE,
  maxiter = 500L,
  miniter = 100L,
  k = 3L,
  tol = 1e-05,
  verbose = interactive(),
  fn.a = function(iter, a = 1, b = 1/2, c = 0, ...) a/(iter + c)^b
)

## S3 method for class 'RM'
print(x, ...)

## S3 method for class 'RM'
plot(x, par = 1, main = NULL, Polyak_Juditsky = FALSE, ...)
```

Arguments

`f`	noisy function for which the root is sought
`p`	vector of starting values to be passed as `f(p, ...)`
`...`	additional named arguments to be passed to `f`
`Polyak_Juditsky`	logical; apply the Polyak and Juditsky (1992) running-average method? Returns the final running average estimate using the Robbins-Monro updates (also applies to `plot`). Note that this should only be used when the step-sizes are sufficiently large so that the Robbins-Monro have the ability to stochastically explore around the root (not just approach it from one side, which occurs when using small steps)
`maxiter`	the maximum number of iterations (default 500)
`miniter`	minimum number of iterations (default 100)

k	number of consecutive tol criteria required before terminating
tol	tolerance criteria for convergence on the changes in the updated p elements. Must be achieved on k (default 3) successive occasions
verbose	logical; should the iterations and estimate be printed to the console?
fn.a	function to create the a coefficient in the Robbins-Monro noise filter. Requires the first argument is the current iteration (iter), provide one or more arguments, and (optionally) the .... Sequence function is of the form recommended by Spall (2000). Note that if a different function is provided it must satisfy the property that $\sum_{i=1}^{\infty} a_i = \infty$ and $\sum_{i=1}^{\infty} a_i^2 < \infty$
x	an object of class RM
par	which parameter in the original vector p to include in the plot
main	plot title

References

Polyak, B. T. and Juditsky, A. B. (1992). Acceleration of Stochastic Approximation by Averaging. SIAM Journal on Control and Optimization, 30(4):838.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. Ann.Math.Statistics, 22:400-407.

Spall, J.C. (2000). Adaptive stochastic approximation by the simultaneous perturbation method. IEEE Trans. Autom. Control 45, 1839-1853.

See Also

uniroot, PBA

Examples

```
# find x that solves f(x) - b = 0 for the following
f.root <- function(x, b = .6) 1 / (1 + exp(-x)) - b
f.root(.3)

xs <- seq(-3,3, length.out=1000)
plot(xs, f.root(xs), type = 'l', ylab = "f(x)", xlab='x')
abline(h=0, col='red')

retuni <- uniroot(f.root, c(0,1))
retuni
abline(v=retuni$root, col='blue', lty=2)

# Robbins-Monro without noisy root, start with p=.9
retrm <- RobbinsMonro(f.root, .9)
retrm
plot(retrm)

# Same problem, however root function is now noisy. Hence, need to solve
#  fhat(x) - b + e = 0, where E(e) = 0
```

```
f.root_noisy <- function(x) 1 / (1 + exp(-x)) - .6 + rnorm(1, sd=.02)
sapply(rep(.3, 10), f.root_noisy)

# uniroot "converges" unreliably
set.seed(123)
uniroot(f.root_noisy, c(0,1))$root
uniroot(f.root_noisy, c(0,1))$root
uniroot(f.root_noisy, c(0,1))$root

# Robbins-Monro provides better convergence
retrm.noise <- RobbinsMonro(f.root_noisy, .9)
retrm.noise
plot(retrm.noise)

# different power (b) for fn.a()
retrm.b2 <- RobbinsMonro(f.root_noisy, .9, b = .01)
retrm.b2
plot(retrm.b2)

# use Polyak-Juditsky averaging (b should be closer to 0 to work well)
retrm.PJ <- RobbinsMonro(f.root_noisy, .9, b = .01,
                         Polyak_Juditsky = TRUE)
retrm.PJ   # final Polyak_Juditsky estimate
plot(retrm.PJ) # Robbins-Monro history
plot(retrm.PJ, Polyak_Juditsky = TRUE) # Polyak_Juditsky history
```

RSE *Compute the relative standard error ratio*

Description

Computes the relative standard error ratio given the set of estimated standard errors (SE) and the deviation across the R simulation replications (SD). The ratio is formed by finding the expectation of the SE terms, and compares this expectation to the general variability of their respective parameter estimates across the R replications (ratio should equal 1). This is used to roughly evaluate whether the SEs being advertised by a given estimation method matches the sampling variability of the respective estimates across samples.

Usage

```
RSE(SE, ests, unname = FALSE)
```

Arguments

SE	a `numeric` matrix of SE estimates across the replications (extracted from the `results` object in the Summarise step). Alternatively, can be a vector containing the mean of the SE estimates across the R simulation replications

ests	a numeric matrix object containing the parameter estimates under investigation found within the Summarise function. This input is used to compute the standard deviation/variance estimates for each column to evaluate how well the expected SE matches the standard deviation
unname	logical; apply unname to the results to remove any variable names?

Value

returns vector of variance ratios, (RSV = SE^2/SD^2)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
R <- 10000
par_ests <- cbind(rnorm(R), rnorm(R, sd=1/10),
                  rnorm(R, sd=1/15))
colnames(par_ests) <- paste0("par", 1:3)
(SDs <- colSDs(par_ests))

SEs <- cbind(1 + rnorm(R, sd=.01),
             1/10 + + rnorm(R, sd=.01),
             1/15 + rnorm(R, sd=.01))
(E_SEs <- colMeans(SEs))
RSE(SEs, par_ests)

# equivalent to the form
colMeans(SEs) / SDs
```

---

rtruncate *Generate a random set of values within a truncated range*

Description

Function generates data given a supplied random number generating function that are constructed to fall within a particular range. Sampled values outside this range are discarded and re-sampled until the desired criteria has been met.

Usage

```
rtruncate(n, rfun, range, ..., redraws = 100L)
```

Arguments

n	number of observations to generate. This should be the first argument passed to rfun
rfun	a function to generate random values. Function can return a numeric/integer vector or matrix, and additional arguments requred for this function are passed through the argument ...
range	a numeric vector of length two, where the first element indicates the lower bound and the second the upper bound. When values are generated outside these two bounds then data are redrawn until the bounded criteria is met. When the output of rfun is a matrix then this input can be specified as a matrix with two rows, where each the first row corresponds to the lower bound and the second row the upper bound for each generated column in the output
...	additional arguments to be passed to rfun
redraws	the maximum number of redraws to take before terminating the iterative sequence. This is in place as a safety in case the range is too small given the random number generator, causing too many consecutive rejections. Default is 100

Details

In simulations it is often useful to draw numbers from truncated distributions rather than across the full theoretical range. For instance, sampling parameters within the range [-4,4] from a normal distribution. The rtruncate function has been designed to accept any sampling function, where the first argument is the number of values to sample, and will draw values iteratively until the number of values within the specified bound are obtained. In situations where it is unlikely for the bounds to be located (e.g., sampling from a standard normal distribution where all values are within [-10,-6]) then the sampling scheme will throw an error if too many re-sampling executions are required (default will stop if more that 100 calls to rfun are required).

Value

either a numeric vector or matrix, where all values are within the desired range

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
# n = 1000 truncated normal vector between [-2,3]
vec <- rtruncate(1000, rnorm, c(-2,3))
summary(vec)

# truncated correlated multivariate normal between [-1,4]
mat <- rtruncate(1000, rmvnorm, c(-1,4),
   sigma = matrix(c(2,1,1,1),2))
summary(mat)

# truncated correlated multivariate normal between [-1,4] for the
#  first column and [0,3] for the second column
mat <- rtruncate(1000, rmvnorm, cbind(c(-1,4), c(0,3)),
   sigma = matrix(c(2,1,1,1),2))
summary(mat)

# truncated chi-square with df = 4 between [2,6]
vec <- rtruncate(1000, rchisq, c(2,6), df = 4)
summary(vec)
```

---

runArraySimulation *Run a Monte Carlo simulation using array job submissions per condition*

Description

This function has the same purpose as runSimulation, however rather than evaluating each row in a design object (potentially with parallel computing architecture) this function evaluates the simulation per independent row condition. This is mainly useful when distributing the jobs to HPC clusters where a job array number is available (e.g., via SLURM), where the simulation results must be saved to independent files as they complete. Use of expandDesign is useful for distributing replications to different jobs, while genSeeds is required to ensure high-quality random number generation across the array submissions. See the associated vignette for a brief tutorial of this setup.

Usage

```
runArraySimulation(
  design,
  ...,
  replications,
  iseed,
  filename,
```

```
  dirname = NULL,
  arrayID = getArrayID(),
  array2row = function(arrayID) arrayID,
  addArrayInfo = TRUE,
  parallel = FALSE,
  cl = NULL,
  ncores = parallelly::availableCores(omit = 1L),
  save_details = list(),
  control = list(),
  verbose = interactive()
)
```

Arguments

design	design object containing simulation conditions on a per row basis. This function is design to submit each row as in independent job on a HPC cluster. See runSimulation for further details
...	additional arguments to be passed to runSimulation
replications	number of independent replications to perform per condition (i.e., each row in design). See runSimulation for further details
iseed	initial seed to be passed to genSeeds's argument of the same name, along with the supplied arrayID
filename	file name to save simulation files to (does not need to specify extension). However, the array ID will be appended to each filename. For example, if filename = 'mysim' then files stored will be 'mysim-1.rds', 'mysim-2.rds', and so on for each row ID in design
dirname	directory to save the files associated with filename to. If omitted the files will be stored in the same working directory where the script was submitted
arrayID	array identifier from the scheduler. Must be a number between 1 and nrow(design). If not specified then getArrayID will be called automatically, which assumes the environmental variables are available according the SLURM scheduler
array2row	user defined function with the single argument arrayID. Used to convert the detected arrayID into a suitable row index in the design object input. By default each arrayID is associated with its respective row in design. For example, if each arrayID should evaluate 10 rows in the design object then the function function(arrayID){1:10 + 10 * (arrayID-1)} can be passed to array2row
addArrayInfo	logical; should the array ID and original design row number be added to the SimResults(...) output?
parallel	logical; use parallel computations via the a "SOCK" cluster? Only useful when the instruction shell file requires more than 1 core (number of cores detected via ncores). For this application the random seeds further distributed using nextRNGSubStream
cl	cluster definition. If omitted a "SOCK" cluster will be defined

ncores — number of cores to use when parallel=TRUE. Note that the default uses 1 minus the number of available cores, therefore this will only be useful when ncores > 2 as defined in the shell instruction file

save_details — optional list of extra file saving details. See runSimulation

control — control list passed to runSimulation. In addition to the original control elements two additional arguments have been added: max_time and max_RAM, both of which as specified as character vectors with one element.

max_time specifies the maximum time allowed for a single simulation condition to execute (default does not set any time limits), and is formatted according to the specification in timeFormater. This is primarily useful when the HPC cluster will time out after some known elapsed time. In general, this input should be set to somewhere around 80-90 before the cluster is terminated can be saved. Default applies no time limit

Similarly, max_RAM controls the (approximate) maximum size that the simulation storage objects can grow before RAM becomes an issue. This can be specified either in terms of megabytes (MB), gigabytes (GB), or terabytes (TB). For example, max_RAM = "4GB" indicates that if the simulation storage objects are larger than 4GB then the workflow will terminate early, returning only the successful results up to this point). Useful for larger HPC cluster jobs with RAM constraints that could terminate abruptly. As a rule of thumb this should be set to around 90 available. Default applies no memory limit

verbose — logical; pass a verbose flag to runSimulation. Unlike runSimulation this is set to FALSE during interactive sessions, though set to TRUE when non-interactive and information about the session itself should be stored (e.g., in SLURM .out files)

Details

Due to the nature of how the replication are split it is important that the L'Ecuyer-CMRG (2002) method of random seeds is used across all array ID submissions (cf. runSimulation's parallel approach, which uses this method to distribute random seeds within each isolated condition rather than between all conditions). As such, this function requires the seeds to be generated using genSeeds with the iseed and arrayID inputs to ensure that each job is analyzing a high-quality set of random numbers via L'Ecuyer-CMRG's (2002) method, incremented using nextRNGStream.

Additionally, for timed simulations on HPC clusters it is also recommended to pass a control = list(max_time) value to avoid discarding conditions that require more than the specified time in the shell script. The max_time value should be less than the maximum time allocated on the HPC cluster (e.g., approximately 90 depends on how long, and how variable, each replication is). Simulations with missing replications should submit a new set of jobs at a later time to collect the missing information.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

See Also

runSimulation, expandDesign, genSeeds, SimCheck, SimCollect, getArrayID

Examples

```
library(SimDesign)

Design <- createDesign(N = c(10, 20, 30))

Generate <- function(condition, fixed_objects) {
    dat <- with(condition, rnorm(N, 10, 5)) # distributed N(10, 5)
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    ret <- c(mean=mean(dat), median=median(dat)) # mean/median of sample data
    ret
}

Summarise <- function(condition, results, fixed_objects){
    colMeans(results)
}

## Not run:

# define initial seed (do this only once to keep it constant!)
# iseed <- genSeeds()
iseed <- 554184288

### On cluster submission, the active array ID is obtained via getArrayID(),
###   and therefore should be used in real SLURM submissions
arrayID <- getArrayID(type = 'slurm')

# However, the following example arrayID is set to
#  the first row only for testing purposes
arrayID <- 1L

# run the simulation (results not caught on job submission, only files saved)
res <- runArraySimulation(design=Design, replications=50,
                      generate=Generate, analyse=Analyse,
                      summarise=Summarise, arrayID=arrayID,
                      iseed=iseed, filename='mysim') # saved as 'mysim-1.rds'
res
SimResults(res) # condition and replication count stored

# same, but evaluated with multiple cores
```

```
res <- runArraySimulation(design=Design, replications=50,
                      generate=Generate, analyse=Analyse,
                      summarise=Summarise, arrayID=arrayID,
                      parallel=TRUE, ncores=3,
                      iseed=iseed, filename='myparsim')
res
SimResults(res) # condition and replication count stored

dir()
SimClean(c('mysim-1.rds', 'myparsim-1.rds'))

########################
# Same submission job as above, however split the replications over multiple
# evaluations and combine when complete
Design5 <- expandDesign(Design, 5)
Design5

# iseed <- genSeeds()
iseed <- 554184288

# arrayID <- getArrayID(type = 'slurm')
arrayID <- 14L

# run the simulation (replications reduced per row, but same in total)
runArraySimulation(design=Design5, replications=10,
                   generate=Generate, analyse=Analyse,
                   summarise=Summarise, iseed=iseed,
                   filename='mylongsim', arrayID=arrayID)

res <- readRDS('mylongsim-14.rds')
res
SimResults(res) # condition and replication count stored

SimClean('mylongsim-14.rds')


###
# Emulate the arrayID distribution, storing all results in a 'sim/' folder
# (if 'sim/' does not exist in runArraySimulation() it will be
# created automatically)
dir.create('sim/')

# Emulate distribution to nrow(Design5) = 15 independent job arrays
##  (just used for presentation purposes on local computer)
sapply(1:nrow(Design5), \(arrayID)
     runArraySimulation(design=Design5, replications=10,
          generate=Generate, analyse=Analyse,
          summarise=Summarise, iseed=iseed, arrayID=arrayID,
          filename='condition', dirname='sim', # files: "sim/condition-#.rds"
          control = list(max_time="04:00:00", max_RAM="4GB"))) |> invisible()

#  If necessary, conditions above will manually terminate before
#  4 hours and 4GB of RAM are used, returning any
```

```
#  successfully completed results before the HPC session times
#  out (provided .slurm script specified more than 4 hours)

# list saved files
dir('sim/')

# check that all files saved (warnings will be raised if missing files)
SimCheck('sim/')

condition14 <- readRDS('sim/condition-14.rds')
condition14
SimResults(condition14)

# aggregate simulation results into single file
final <- SimCollect('sim/')
final

# clean simulation directory
SimClean(dirs='sim/')


#############
# same as above, however passing different amounts of information depending
# on the array ID
array2row <- function(arrayID){
  switch(arrayID,
    "1"=1:8,
    "2"=9:14,
    "3"=15)
}

# arrayID 1 does row 1 though 8, arrayID 2 does 9 to 14
array2row(1)
array2row(2)
array2row(3)  # arrayID 3 does 15 only

# emulate remote array distribution with only 3 arrays
sapply(1:3, \(arrayID)
     runArraySimulation(design=Design5, replications=10,
          generate=Generate, analyse=Analyse,
          summarise=Summarise, iseed=iseed, arrayID=arrayID,
          filename='condition', dirname='sim', array2row=array2row)) |> invisible()

# list saved files
dir('sim/')

# Also works if the replication input were a suitable vector (not run)
if(FALSE){
    reps <- c(rep(10, 14), 100)
    cbind(Design5, reps)
    sapply(1:3, \(arrayID)
        runArraySimulation(design=Design5, replications=reps,
          generate=Generate, analyse=Analyse,
```

```
          summarise=Summarise, iseed=iseed, arrayID=arrayID,
          filename='condition', dirname='sim', array2row=array2row)) |> invisible()
}


# note that all row conditions are still stored separately, though note that
#  arrayID is now 2 instead
condition14 <- readRDS('sim/condition-14.rds')
condition14
SimResults(condition14)

# aggregate simulation results into single file
final <- SimCollect('sim/')
final

# clean simulation directory
SimClean(dirs='sim/')


## End(Not run)
```

`runSimulation` *Run a Monte Carlo simulation given conditions and simulation functions*

Description

This function runs a Monte Carlo simulation study given a set of predefined simulation functions, design conditions, and number of replications. Results can be saved as temporary files in case of interruptions and may be restored by re-running `runSimulation`, provided that the respective temp file can be found in the working directory. `runSimulation` supports parallel and cluster computing (with the `parallel` and `future` packages; see also `runArraySimulation` for submitting array jobs to HPC clusters), global and local debugging, error handling (including fail-safe stopping when functions fail too often, even across nodes), provides bootstrap estimates of the sampling variability (optional), and automatic tracking of error and warning messages with their associated `.Random.seed` states. For convenience, all functions available in the R work-space are exported across all nodes so that they are more easily accessible (however, other R objects are not, and therefore must be passed to the `fixed_objects` input to become available across nodes).

Usage

```
runSimulation(
  design,
  replications,
  generate,
  analyse,
  summarise,
  prepare = NULL,
```

```
  fixed_objects = NULL,
  packages = NULL,
  filename = NULL,
  debug = "none",
  load_seed = NULL,
  load_seed_prepare = NULL,
  save = any(replications > 2),
  store_results = TRUE,
  save_results = FALSE,
  parallel = FALSE,
  ncores = parallelly::availableCores(omit = 1L),
  cl = NULL,
  notification = "none",
  notifier = NULL,
  beep = FALSE,
  sound = 1,
  check.globals = FALSE,
  CI = 0.95,
  seed = NULL,
  boot_method = "none",
  boot_draws = 1000L,
  max_errors = 50L,
  resume = TRUE,
  save_details = list(),
  control = list(),
  not_parallel = NULL,
  progress = TRUE,
  verbose = interactive()
)

## S3 method for class 'SimDesign'
summary(object, ...)

## S3 method for class 'SimDesign'
print(x, list2char = TRUE, ...)
```

Arguments

design	a tibble or data.frame object containing the Monte Carlo simulation conditions to be studied, where each row represents a unique condition and each column a factor to be varied. See createDesign for the standard approach to create this simulation design object As an augmentation of the input, the original design object can be passed to expandDesign to systemically repeat the rows of each simulation condition, allowing smaller numbers of replications to be supplied per condition. After this distributed job is complete the function SimCollect can be used to recombine the results to a form commensurate with the original design object
replications	number of independent replications to perform per condition (i.e., each row in

design). Can be a single number, which will be used for each design condition, or an integer vector with length equal to nrow(design). All inputs must be greater than 0, though setting to less than 3 (for initial testing purpose) will disable the save and control$stop_on_fatal flags

generate — user-defined data and parameter generating function (or named list of functions). See Generate for details. Note that this argument may be omitted by the user if they wish to generate the data with the analyse step, but for real-world simulations this is generally not recommended. If multiple generate functions are provided as a list then the list of generate functions are executed in order until the first valid generate function is executed, where the subsequent generation functions are then ignored (see GenerateIf to only apply data generation for specific conditions).

analyse — user-defined analysis function (or named list of functions) that acts on the data generated from Generate (or, if generate was omitted, can be used to generate and analyse the simulated data). See Analyse for details

summarise — optional (but strongly recommended) user-defined summary function from Summarise to be used to compute meta-statistical summary information after all the replications have completed within each design condition. Return of this function, in order of increasing complexity, should be: a named numeric vector or data.frame with one row, a matrix or data.frame with more than one row, and, failing these more atomic types, a named list. When a list is returned the final simulation object will contain a column SUMMARISE containing the summary results for each respective condition

Note that unlike the Generate and Analyse steps, the Summarise portion is not as important to perfectly organize as the results can be summarised later on by using the built-in reSummarise function (provided either store_results = TRUE or save_results = TRUE were included).

Omitting this function will return a tibble with the Design and associated results information for all nrow(Design) * repliations evaluations if the results from each Analyse() call was a one-dimensional vector. For more general objects returned by Analyse() (such as lists), a list containing the results returned from Analyse. This is generally only recommended for didactic purposes because the results will leave out a large amount of information (e.g., try-errors, warning messages, saving files, etc), can witness memory related issues if the Analyse function returns larger objects, and generally is not as flexible internally. However, it may be useful when replications are expensive and ANOVA-based decompositions involving the within-condition replication information are of interest, though of course this can be circumvented by using store_results = TRUE or save_results = TRUE with or without a supplied summarise definition.

Finally, there are keywords that should not be returned from this function, since they will cause a conflict with the aggregated simulation objects. These are currently those listed in capital letters (e.g., ERRORS, WARNINGS, REPLICATIONS, etc), all of which can be avoided if the returned objects are not entirely capitalized (e.g., Errors, errors, ErRoRs, ..., will all avoid conflicts)

prepare — (optional) a function that executes once per simulation condition (i.e., once per row in design) to load or prepare condition-specific objects into fixed_objects

before replications are run. This function should accept `condition` and `fixed_objects` as arguments and return the modified `fixed_objects`.

The primary use case is to load pre-computed objects from disk that were generated offline:

```
prepare <- function(condition, fixed_objects) {
 # Create filename from design parameters
 fname <- paste0('prepare/N', condition$N, '_SD', condition$SD, '.rds')
 fixed_objects$expensive_stuff <- readRDS(fname)
 fixed_objects
}
```

This approach allows you to: (1) pre-generate expensive condition-specific objects prior to running the simulation , (2) save them as individual RDS files, and (3) load them efficiently during the simulation. This is preferable to generating objects within `prepare()` itself because it allows you to inspect the objects, ensures reproducibility, and separates object generation from the simulation workflow.

Note: Objects added to `fixed_objects` in `prepare()` are not stored by `runSimulation()` to conserve memory, as they are typically large. You should maintain your own records of these objects outside the simulation.

RNG Warning: If you generate objects within `prepare()` using random number generation (e.g., `rnorm()`, `runif()`), reproducibility requires explicit RNG state management via `store_Random.seeds=TRUE` and `load_seed_prepare`. Precomputing and saving objects as RDS files is the recommended approach for reproducible simulations.

The function signature should be: `prepare <- function(condition, fixed_objects) { ... return(fixed_objects) }`

Default is `NULL`, in which case no preparation step is performed

`fixed_objects` (optional) an object (usually a named `list`) containing additional user-defined objects that should remain fixed across conditions. This is useful when including large vectors/matrices of population parameters, fixed data information that should be used across all conditions and replications (e.g., including a common design matrix for linear regression models), or simply control constant global elements (e.g., a constant for sample size)

`packages` a character vector of external packages to be used during the simulation (e.g., `c('MASS', 'extraDistr', 'simsem')`). Use this input when running code in parallel to use non-standard functions from additional packages. Note that any previously attached packages explicitly loaded via `library` or `require` will be automatically added to this list, provided that they are visible in the `otherPkgs` element from `sessionInfo`. Alternatively, functions can be called explicitly without attaching the package with the `::` operator (e.g., `extraDistr::rgumbel()`)

`filename` (optional) the name of the `.rds` file to save the final simulation results to. If the extension `.rds` is not included in the file name (e.g. `"mysimulation"` versus `"mysimulation.rds"`) then the `.rds` extension will be automatically added to the file name to ensure the file extension is correct.

Note that if the same file name already exists in the working directly at the time of saving then a new file will be generated instead and a warning will be

thrown. This helps to avoid accidentally overwriting existing files. Default is NULL, indicating no file will be saved by default

debug

a string indicating where to initiate a browser() call for editing and debugging, and pairs particularly well with the load_seed argument for precise debugging. General options are 'none' (default; no debugging), 'error', which starts the debugger when any error in the code is detected in one of three generate-analyse-summarise functions, and 'all', which debugs all the user defined functions regardless of whether an error was thrown or not. Specific options include: 'generate' to debug the data simulation function, 'analyse' to debug the computational function, and 'summarise' to debug the aggregation function.

If the Analyse argument is supplied as a named list of functions then it is also possible to debug the specific function of interest by passing the name of the respective function in the list. For instance, if analyse = list(A1=Analyse.A1, A2=Analyse.A2) then passing debug = 'A1' will debug only the first function in this list, and all remaining analysis functions will be ignored.

For debugging specific rows in the Design input (e.g., when a number of initial rows successfully complete but the kth row fails) the row number can be appended to the standard debug input using a '-' separator. For instance, debugging whenever an error is raised in the second row of Design can be declared via debug = 'error-2'.

Finally, users may place browser calls within the respective functions for debugging at specific lines, which is useful when debugging based on conditional evaluations (e.g., if(this == 'problem') browser()). Note that parallel computation flags will automatically be disabled when a browser() is detected or when a debugging argument other than 'none' is supplied

load_seed

used to replicate an exact simulation state, which is primarily useful for debugging purposes. Input can be a character object indicating which file to load from when the .Random.seeds have be saved (after a call with save_seeds = TRUE), or an integer vector indicating the actual .Random.seed values (e.g., extracted after using store_seeds). E.g., load_seed = 'design-row-2/seed-1' will load the first seed in the second row of the design input, or explicitly passing the elements from .Random.seed (see SimExtract to extract the seeds associated explicitly with errors during the simulation, where each column represents a unique seed). If the input is a character vector then it is important NOT to modify the design input object, otherwise the path may not point to the correct saved location, while if the input is an integer vector (or single column tbl object) then it WILL be important to modify the design input in order to load this exact seed for the corresponding design row. Default is NULL

load_seed_prepare

similar to load_seed, but specifically for debugging the prepare function. Used to replicate the exact RNG state when prepare is called for a given condition. Accepts the same input formats as load_seed: a character string path (e.g., 'design-row-2/prepare-seed'), an integer vector containing the .Random.seed state, or a tibble/data.frame with seed values. This is particularly useful when prepare encounters an error and you need to reproduce the exact state. The prepare error seed can be extracted using SimExtract(res, 'prepare_error_seed'). Default is NULL

save — logical; save the temporary simulation state to the hard-drive? This is useful for simulations which require an extended amount of time, though for shorter simulations can be disabled to slightly improve computational efficiency. When TRUE, which is the default when evaluating replications > 2, a temp file will be created in the working directory which allows the simulation state to be saved and recovered (in case of power outages, crashes, etc). As well, triggering this flag will save any fatal .Random.seed states when conditions unexpectedly crash (where each seed is stored row-wise in an external .rds file), which provides a much easier mechanism to debug issues (see load_seed for details). Upon completion, this temp file will be removed.

To recover your simulation at the last known location (having patched the issues in the previous execution code) simply re-run the code you used to initially define the simulation and the external file will automatically be detected and read-in. Default is TRUE when replications > 10 and FALSE otherwise

store_results — logical; store the complete tables of simulation results in the returned object? This is TRUE default, though if RAM anticipated to be an issue see save_results instead. Note that if the Design object is omitted from the call to runSimulation(), or the number of rows in Design is exactly 1, then this argument is automatically set to TRUE as RAM storage is no longer an issue.

To extract these results pass the returned object to either SimResults or SimExtract with what = 'results', which will return a named list of all the simulation results for each condition if nrow(Design) > 1; otherwise, if nrow(Design) == 1 or Design was missing the results object will be stored as-is

save_results — logical; save the results returned from Analyse to external .rds files located in the defined save_results_dirname directory/folder? Use this if you would like to keep track of the individual parameters returned from the analysis function. Each saved object will contain a list of three elements containing the condition (row from design), results (as a list or matrix), and try-errors. See SimResults for an example of how to read these .rds files back into R after the simulation is complete. Default is FALSE.

WARNING: saving results to your hard-drive can fill up space very quickly for larger simulations. Be sure to test this option using a smaller number of replications before the full Monte Carlo simulation is performed. See also reSummarise for applying summarise functions from saved simulation results

parallel — logical; use parallel processing from the parallel package over each unique condition? This distributes the independent replications across the defined nodes, and is repeated for each row condition in the design input.

Alternatively, if the future package approach is desired then passing parallel = 'future' to runSimulation() will use the defined plan for execution. This allows for greater flexibility when specifying the general computing plan (e.g., plan(multisession)) for parallel computing on the same machine, plan(future.batchtools::batch
or plan(future.batchtools::batchtools_slurm) for common MPI/Slurm schedulers, etc). However, it is the responsibility of the user to use plan(sequential) to reset the computing plan when the jobs are completed

ncores — number of cores to be used in parallel execution (ignored if using the future package approach). Default uses all available minus 1

cl — cluster object defined by makeCluster used to run code in parallel (ignored if using the future package approach). If NULL and parallel = TRUE, a local cluster object will be defined which selects the maximum number cores available and will be stopped when the simulation is complete. Note that supplying a cl object will automatically set the parallel argument to TRUE. Define and supply this cluster object yourself whenever you have multiple nodes and can link them together manually

If the future package has been attached prior to executing runSimulation() then the associated plan() will be followed instead

notification — an optional character vector input that can be used to send notifications with information about execution time and recorded errors and warnings. Pass one of the following supported options: 'none' (default; send no notification), 'condition' to send a notification after each condition has completed, or 'complete' to send a notification only when the simulation has finished. When notification is set to 'condition' or 'complete', the notifier argument must be supplied with a valid notifier object (or a list of notifier objects).

notifier — A notifier object (or a list of notifier objects, allowing for multiple notification methods) required when notification is not set to "none". See listAvailableNotifiers for a list of available notifiers and how to use them.

Example usage:

```
telegram_notifier <- new_TelegramNotifier(bot_token = "123456:ABC-xyz",
chat_id = "987654321")
runSimulation(..., notification = "condition", notifier = telegram_notifier)
```

Using multiple notifiers:

```
pushbullet_notifier <- new_PushbulletNotifier()
runSimulation(..., notification = "complete", notifier = list(telegram_notifier,
pushbullet_notifier))
```

See the R/notifications.R file for reference on implementing a custom notifier.

beep — logical; call the beepr package when the simulation is completed?

sound — sound argument passed to beepr::beep()

check.globals — logical; should the functions be inspected for potential global variable usage? Using global object definitions will raise issues with parallel processing, and therefore any global object to be use in the simulation should either be defined within the code itself of included in the fixed_objects input. Setting this value to TRUE will return a character vector of potentially problematic objects/functions that appear global, the latter of which can be ignored. Careful inspection of this list may prove useful in tracking down object export issues

CI — bootstrap confidence interval level (default is 95%)

seed — a vector or list of integers to be used for reproducibility. The length of the vector must be equal the number of rows in design. If the input is a vector then set.seed or clusterSetRNGStream for each condition will be called, respectively. If a list is provided then these numbers must have been generated from genSeeds. The list approach ensures random number generation independence across conditions and replications, while the vector input ensures independence

within the replications per conditions but not necessarily across conditions. Default randomly generates seeds within the range 1 to 2147483647 for each condition via genSeeds

boot_method method for performing non-parametric bootstrap confidence intervals for the respective meta-statistics computed by the Summarise function. Can be 'basic' for the empirical bootstrap CI, 'percentile' for percentile CIs, 'norm' for normal approximations CIs, or 'studentized' for Studentized CIs (should only be used for simulations with lower replications due to its computational intensity). Alternatively, CIs can be constructed using the argument 'CLT', which computes the intervals according to the large-sample standard error approximation $SD(results)/\sqrt{R}$. Default is 'none', which performs no CI computations

boot_draws number of non-parametric bootstrap draws to sample for the summarise function after the generate-analyse replications are collected. Default is 1000

max_errors the simulation will terminate when more than this number of consecutive errors are thrown in any given condition, causing the simulation to continue to the next unique design condition. This is included to avoid getting stuck in infinite redraws, and to indicate that something fatally problematic is going wrong in the generate-analyse phases. Default is 50

resume logical; if a temporary SimDesign file is detected should the simulation resume from this location? Keeping this TRUE is generally recommended, however this should be disabled if using runSimulation within runSimulation to avoid reading improper save states. Alternatively, if an integer is supplied then the simulation will continue at the associated row location in design (e.g., resume=10). This is useful to overwrite a previously evaluate element in the temporary files that was detected to contain fatal errors that require re-evaluation without discarding the originally valid rows in the simulation

save_details a list pertaining to information regarding how and where files should be saved when the save or save_results flags are triggered.

safe logical; trigger whether safe-saving should be performed. When TRUE files will never be overwritten accidentally, and where appropriate the program will either stop or generate new files with unique names. Default is TRUE

compname name of the computer running the simulation. Normally this doesn't need to be modified, but in the event that a manual node breaks down while running a simulation the results from the temp files may be resumed on another computer by changing the name of the node to match the broken computer. Default is the result of evaluating unname(Sys.info()['nodename'])

out_rootdir root directory to save all files to. Default uses the current working directory

save_results_dirname a string indicating the name of the folder to save result objects to when save_results = TRUE. If a directory/folder does not exist in the current working directory then a unique one will be created automatically. Default is 'SimDesign-results_' with the associated compname appended if no filename is defined, otherwise the filename is used to replace 'SimDesign' in the string

save_results_filename a string indicating the name file to store, where the Design row ID will be appended to ensure uniqueness across rows. Specifying this input will disable any checking for the uniqueness of the file folder, thereby allowing independent runSimulation calls to write to the same save_results_dirname. Useful when the files should all be stored in the same working directory, however the rows of Design are evaluated in isolation (e.g., for HPC structures that allow asynchronous file storage). WARNING: the uniqueness of the file names are not checked using this approach, therefore please ensure that each generated name will be unique a priori, such as naming the file based on the supplied row condition information

save_seeds_dirname a string indicating the name of the folder to save .Random.seed objects to when save_seeds = TRUE. If a directory/folder does not exist in the current working directory then one will be created automatically. Default is 'SimDesign-seeds_' with the associated compname appended if no filename is defined, otherwise the filename is used to replace 'SimDesign' in the string

tmpfilename string indicating the temporary file name to save provisional information to. If not specified the following will be used: paste0('SIMDESIGN-TEMPFILE_', compname, '.rds')

control a list for extra information flags for controlling less commonly used features. These include

stop_on_fatal logical (default is FALSE); should the simulation be terminated immediately when the maximum number of consecutive errors (max_errors) is reached? If FALSE, the simulation will continue as though errors did not occur, however a column FATAL_TERMINATION will be included in the resulting object indicating the final error message observed, and NA placeholders will be placed in all other row-elements. Default is FALSE, though is automatically set to TRUE when replications < 3 for the purpose of debugging

warnings_as_errors logical (default is FALSE); treat warning messages as error messages during the simulation? Default is FALSE, therefore warnings are only collected and not used to restart the data generation step, and the seeds associated with the warning message conditions are not stored within the final simulation object.

Note that this argument is generally intended for debugging/early planning stages when designing a simulation experiment. If specific warnings are known to be problematic and should be treated as errors then please use manageWarnings instead

logging a character vector indicating whether the execution times for the generate and analyse functions should be logged. Values can be 'store' to store the times (and later extract using SimExtract) or 'verbose' to both store and print the times via cat for stdout piping (useful on cluster computing)

save_seeds logical; save the .Random.seed states prior to performing each replication into plain text files located in the defined save_seeds_dirname directory/folder? Use this if you would like to keep track of every simulation state within each replication and design condition. This can be used

to completely replicate any cell in the simulation if need be. As well, see the load_seed input to load a given .Random.seed to exactly replicate the generated data and analysis state (mostly useful for debugging). When TRUE, temporary files will also be saved to the working directory (in the same way as when save = TRUE). Additionally, if a prepare function is provided, the RNG state before prepare() execution is saved to 'design-row-X/prepare-seed'. Default is FALSE

Note, however, that this option is not typically necessary or recommended since the .Random.seed states for simulation replications that throw errors during the execution are automatically stored within the final simulation object, and can be extracted and investigated using SimExtract. Hence, this option is only of interest when *all* of the replications must be reproducible (which occurs very rarely), otherwise the defaults to runSimulation should be sufficient

store_Random.seeds logical; store the complete .Random.seed states for each simulation replicate? Default is FALSE as this can take up a great deal of unnecessary RAM (see related save_seeds), however this may be useful when used with runArraySimulation. To extract use SimExtract(..., what = 'stored_Random.seeds'). Additionally, if a prepare function is provided, the RNG state before prepare() execution is stored and can be extracted with SimExtract(..., what = 'prepare_seeds')

store_warning_seeds logical (default is FALSE); in addition to storing the .Random.seed states whenever error messages are raised, also store the .Random.seed states when warnings are raised? This is disabled by default since warnings are generally less problematic than errors, and because many more warnings messages may be raised throughout the simulation (potentially causing RAM related issues when constructing the final simulation object as any given simulation replicate could generate numerous warnings, and storing the seeds states could add up quickly).

Set this to TRUE when replicating warning messages is important, however be aware that too many warnings messages raised during the simulation implementation could cause RAM related issues.

include_replication_index **or** include_reps logical (default is FALSE); should a REPLICATION element be added to the condition object when performing the simulation to track which specific replication experiment is being evaluated? This is useful when, for instance, attempting to run external software programs (e.g., Mplus) that require saving temporary data sets to the hard-drive (see the Wiki for examples)

try_all_analyse logical; when analyse is a list, should every generated data set be analyzed by each function definition in the analyse list? Default is TRUE.

Note that this TRUE default can be computationally demanding when some analysis functions require more computational resources than others, and the data should be discarded early as an invalid candidate (e.g., estimating a model via maximum-likelihood in on analyze component, while estimating a model using MCMC estimation on another). Hence, the main benefit of using FALSE instead is that the data set may be rejected earlier, where easier/faster to estimate analyse definitions should be placed ear-

lier in the list as the functions are evaluated in sequence (e.g., `Analyse = list(MLE=MLE_definition, MCMC=MCMC_definition)`)

`allow_na` logical (default is `FALSE`); should `NA`s be allowed in the analyse step as a valid result from the simulation analysis?

`allow_nan` logical (default is `FALSE`); should `NaN`s be allowed in the analyse step as a valid result from the simulation analysis?

`type` default type of cluster to create for the `cl` object if not supplied. For Windows OS this defaults to `"PSOCK"`, otherwise `"SOCK"` is selected (suitable for Linux and Mac OSX). This is ignored if the user specifies their own `cl` object

`print_RAM` logical (default is `TRUE`); print the amount of RAM used throughout the simulation? Set to `FALSE` if unnecessary or if the call to `gc` is unnecessarily time consuming

`global_fun_level` determines how many levels to search until global environment frame is located. Default is 2, though for `runArraySimulation` this is set to 3. Use 3 or more whenever `runSimulation` is used within the context of another function

`max_time` Similar to `runArraySimulation`, specifies the (approximate) maximum time that the simulation is allowed to be executed. Default sets no time limit. See `timeFormater` for the input specifications; otherwise, can be specified as a `numeric` input reflecting the maximum time in seconds. Note that when `parallel = TRUE` the `max_time` can only be checked on a per condition basis.

`max_RAM` Similar to `runArraySimulation`, specifies the (approximate) maximum RAM that the simulation is allowed to occupy. However, unlike the implementation in `runArraySimulation` is evaluated on a per condition basis, where `max_RAM` is only evaluated after every row in the `design` object has been completed (hence, is notably more approximate as it has the potential to overshoot by a wider margin). Default sets no RAM limit. See `runArraySimulation` for the input specifications.

`not_parallel` integer vector indicating which rows in the `design` object should not be run in parallel. This is useful when some version of `parallel` is used, however the overhead that tags along with parallel processing results in higher processing times than simply running the simulation with one core.

`progress` logical; display a progress bar (using the `pbapply` package) for each simulation condition? In interactive sessions, shows a timer-based progress bar. In non-interactive sessions (e.g., HPC cluster jobs), displays text-based progress updates that are visible in log files. This is useful when simulations conditions take a long time to run (see also the `notification` argument). Default is `TRUE`

`verbose` logical; print messages to the R console? Default is `TRUE` when in interactive mode

`object` SimDesign object returned from `runSimulation`

`...` additional arguments

`x` SimDesign object returned from `runSimulation`

`list2char` logical; for `tibble` object re-evaluate list elements as character vectors for better printing of the levels? Note that this does not change the original classes of the object, just how they are printed. Default is TRUE

Details

For an in-depth tutorial of the package please refer to Chalmers and Adkins (2020; doi:10.20982/tqmp.16.4.p248). For an earlier didactic presentation of the package refer to Sigal and Chalmers (2016; doi:10.1080/10691898.2016.1246953). Finally, see the associated wiki on Github (https://github.com/philchalmers/SimDesign/wiki) for tutorial material, examples, and applications of SimDesign to real-world simulation experiments, as well as the various vignette files associated with the package.

The strategy for organizing the Monte Carlo simulation work-flow is to

1) Define a suitable Design object (a tibble or data.frame) containing fixed conditional information about the Monte Carlo simulations. Each row or this design object pertains to a unique set of simulation to study, while each column the simulation factor under investigation (e.g., sample size, distribution types, etc). This is often expedited by using the createDesign function, and if necessary the argument subset can be used to remove redundant or non-applicable rows

2) Define the three step functions to generate the data (Generate; see also https://CRAN.R-project.org/view=Distributions for a list of distributions in R), analyse the generated data by computing the respective parameter estimates, detection rates, etc (Analyse), and finally summarise the results across the total number of replications (Summarise).

3) Pass the design object and three defined R functions to runSimulation, and declare the number of replications to perform with the replications input. This function will return a suitable tibble object with the complete simulation results and execution details

4) Analyze the output from runSimulation, possibly using ANOVA techniques (SimAnova) and generating suitable plots and tables

Expressing the above more succinctly, the functions to be called have the following form, with the exact functional arguments listed:

```
Design <- createDesign(...)

Generate <- function(condition, fixed_objects) {...}

Analyse <- function(condition, dat, fixed_objects) {...}

Summarise <- function(condition, results, fixed_objects) {...}

res <- runSimulation(design=Design, replications, generate=Generate, analyse=Analyse, summarise=Summar
```

The condition object above represents a single row from the design object, indicating a unique Monte Carlo simulation condition. The condition object also contains two additional elements to help track the simulation's state: an ID variable, indicating the respective row number in the design object, and a REPLICATION element indicating the replication iteration number (an integer value between 1 and replication). This setup allows users to easily locate the rth replication (e.g., REPLICATION == 500) within the jth row in the simulation design (e.g., ID == 2). The REPLICATION input is also useful when temporarily saving files to the hard-drive when calling external command line utilities (see examples on the wiki).

For a template-based version of the work-flow, which is often useful when initially defining a simulation, use the SimFunctions function. This function will write a template simulation to one/two files so that modifying the required functions and objects can begin immediately. This means that

users can focus on their Monte Carlo simulation details right away rather than worrying about the repetitive administrative code-work required to organize the simulation's execution flow.

Finally, examples, presentation files, and tutorials can be found on the package wiki located at https://github.com/philchalmers/SimDesign/wiki.

Value

a tibble from the dplyr package (also of class 'SimDesign') with the original design conditions in the left-most columns, simulation results in the middle columns, and additional information in the right-most columns (see below).

The right-most column information for each condition are: REPLICATIONS to indicate the number of Monte Carlo replications, SIM_TIME to indicate how long (in seconds) it took to complete all the Monte Carlo replications for each respective design condition, RAM_USED amount of RAM that was in use at the time of completing each simulation condition, COMPLETED to indicate the date in which the given simulation condition completed, SEED for the integer values in the seed argument (if applicable; not relevant when L'Ecuyer-CMRG method used), and, if applicable, ERRORS and WARNINGS which contain counts for the number of error or warning messages that were caught (if no errors/warnings were observed these columns will be omitted). Note that to extract the specific error and warnings messages see SimExtract. Finally, if boot_method was a valid input other than 'none' then the final right-most columns will contain the labels BOOT_ followed by the name of the associated meta-statistic defined in summarise() and and bootstrapped confidence interval location for the meta-statistics.

Saving data, results, seeds, and the simulation state

To conserve RAM, temporary objects (such as data generated across conditions and replications) are discarded; however, these can be saved to the hard-disk by passing the appropriate flags. For longer simulations it is recommended to use the save_results flag to write the analysis results to the hard-drive.

The use of the store_seeds or the save_seeds options can be evoked to save R's .Random.seed state to allow for complete reproducibility of each replication within each condition. These individual .Random.seed terms can then be read in with the load_seed input to reproduce the exact simulation state at any given replication. Most often though, saving the complete list of seeds is unnecessary as problematic seeds are automatically stored in the final simulation object to allow for easier replicability of potentially problematic errors (which incidentally can be extracted using SimExtract(res, 'error_seeds') and passed to the load_seed argument). Finally, providing a vector of seeds is also possible to ensure that each simulation condition is macro reproducible under the single/multi-core method selected.

Finally, when the Monte Carlo simulation is complete it is recommended to write the results to a hard-drive for safe keeping, particularly with the filename argument provided (for reasons that are more obvious in the parallel computation descriptions below). Using the filename argument supplied is safer than using, for instance, saveRDS directly because files will never accidentally be overwritten, and instead a new file name will be created when a conflict arises; this type of implementation safety is prevalent in many locations in the package to help avoid unrecoverable (yet surprisingly common) mistakes during the process of designing and executing Monte Carlo simulations.

Resuming temporary results

In the event of a computer crash, power outage, etc, if `save = TRUE` was used (the default) then the original code used to execute `runSimulation()` need only be re-run to resume the simulation. The saved temp file will be read into the function automatically, and the simulation will continue one the condition where it left off before the simulation state was terminated. If users wish to remove this temporary simulation state entirely so as to start anew then simply pass `SimClean(temp = TRUE)` in the R console to remove any previously saved temporary objects.

A note on parallel computing

When running simulations in parallel (either with `parallel = TRUE` or when using the `future` approach with a `plan()` other than sequential) R objects defined in the global environment will generally *not* be visible across nodes. Hence, you may see errors such as `Error: object 'something' not found` if you try to use an object that is defined in the work space but is not passed to `runSimulation`. To avoid this type or error, simply pass additional objects to the `fixed_objects` input (usually it's convenient to supply a named list of these objects). Fortunately, however, *custom functions defined in the global environment are exported across nodes automatically*. This makes it convenient when writing code because custom functions will always be available across nodes if they are visible in the R work space. As well, note the `packages` input to declare packages which must be loaded via `library()` in order to make specific non-standard R functions available across nodes.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

`SimFunctions`, `createDesign`, `Generate`, `Analyse`, `Summarise`, `SimExtract`, `reSummarise`, `SimClean`, `SimAnova`, `SimResults`, `SimCollect`, `Attach`, `AnalyseIf`, `SimShiny`, `manageWarnings`, `runArraySimulation`

Examples

```
#-------------------------------------------------------------------------
# Example 1: Sampling distribution of mean

# This example demonstrate some of the simpler uses of SimDesign,
# particularly for classroom settings. The only factor varied in this simulation
# is sample size.

# skeleton functions to be saved and edited
SimFunctions()
```

```
#### Step 1 --- Define your conditions under study and create design data.frame

Design <- createDesign(N = c(10, 20, 30))


#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 2 --- Define generate, analyse, and summarise functions

# help(Generate)
Generate <- function(condition, fixed_objects) {
    dat <- with(condition, rnorm(N, 10, 5)) # distributed N(10, 5)
    dat
}

# help(Analyse)
Analyse <- function(condition, dat, fixed_objects) {
    ret <- c(mean=mean(dat)) # mean of the sample data vector
    ret
}

# help(Summarise)
Summarise <- function(condition, results, fixed_objects) {
    # mean and SD summary of the sample means
    ret <- c(mu=mean(results$mean), SE=sd(results$mean))
    ret
}


#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 3 --- Collect results by looping over the rows in design

# run the simulation in testing mode (replications = 2)
Final <- runSimulation(design=Design, replications=2,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
Final
(results <- SimResults(Final))
results |> dplyr::group_by(N) |> descript()

# reproduce exact simulation
Final_rep <- runSimulation(design=Design, replications=2, seed=Final$SEED,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
Final_rep
(results <- SimResults(Final_rep))
results |> dplyr::group_by(N) |> descript()

## Not run:
# run with more standard number of replications
Final <- runSimulation(design=Design, replications=1000,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
Final
(results <- SimResults(Final))
results |> dplyr::group_by(N) |> descript()
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~
#### Extras
# compare SEs estimates to the true SEs from the formula sigma/sqrt(N)
5 / sqrt(Design$N)

# To store the results from the analyse function either
#   a) omit a definition of summarise() to return all results,
#   b) use store_results = TRUE (default) to store results internally and later
#      extract with SimResults(), or
#  c) pass save_results = TRUE to runSimulation() and read the results in with SimResults()
#
#   Note that method c) should be adopted for larger simulations, particularly
#   if RAM storage could be an issue and error/warning message information is important.

# a) approach
res <- runSimulation(design=Design, replications=100,
                     generate=Generate, analyse=Analyse)
res

# b) approach (store_results = TRUE by default)
res <- runSimulation(design=Design, replications=100,
                     generate=Generate, analyse=Analyse, summarise=Summarise)
res
SimResults(res)

# c) approach
Final <- runSimulation(design=Design, replications=100, save_results=TRUE,
                       generate=Generate, analyse=Analyse, summarise=Summarise)

# read-in all conditions (can be memory heavy)
res <- SimResults(Final)
res
head(res[[1]]$results)

# just first condition
SimResults(Final, which=1)

# obtain empirical bootstrapped CIs during an initial run
# the simulation was completed (necessarily requires save_results = TRUE)
res <- runSimulation(design=Design, replications=1000, boot_method = 'basic',
                     generate=Generate, analyse=Analyse, summarise=Summarise)
res

# alternative bootstrapped CIs that uses saved results via reSummarise().
# Default directory save to:
dirname <- paste0('SimDesign-results_', unname(Sys.info()['nodename']), "/")
res <- reSummarise(summarise=Summarise, dir=dirname, boot_method = 'basic')
res

# remove the saved results from the hard-drive if you no longer want them
SimClean(results = TRUE)
```

```
## End(Not run)


#--------------------------------------------------------------------------------
# Example 2: t-test and Welch test when varying sample size, group sizes, and SDs

# skeleton functions to be saved and edited
SimFunctions()

## Not run:
# in real-world simulations it's often better/easier to save
# these functions directly to your hard-drive with
SimFunctions('my-simulation')

## End(Not run)

#### Step 1 --- Define your conditions under study and create design data.frame

Design <- createDesign(sample_size = c(30, 60, 90, 120),
                       group_size_ratio = c(1, 4, 8),
                       standard_deviation_ratio = c(.5, 1, 2))
Design

#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 2 --- Define generate, analyse, and summarise functions

Generate <- function(condition, fixed_objects) {
    N <- condition$sample_size      # could use Attach() to make objects available
    grs <- condition$group_size_ratio
    sd <- condition$standard_deviation_ratio
    if(grs < 1){
        N2 <- N / (1/grs + 1)
        N1 <- N - N2
    } else {
        N1 <- N / (grs + 1)
        N2 <- N - N1
    }
    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)), DV = c(group1, group2))
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    welch <- t.test(DV ~ group, dat)$p.value
    independent <- t.test(DV ~ group, dat, var.equal=TRUE)$p.value

    # In this function the p values for the t-tests are returned,
    #  and make sure to name each element, for future reference
    ret <- nc(welch, independent)
    ret
}
```

```
Summarise <- function(condition, results, fixed_objects) {
    #find results of interest here (e.g., alpha < .1, .05, .01)
    ret <- EDR(results, alpha = .05)
    ret
}


#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 3 --- Collect results by looping over the rows in design

# first, test to see if it works
res <- runSimulation(design=Design, replications=2,
                     generate=Generate, analyse=Analyse, summarise=Summarise)
res

## Not run:
# complete run with 1000 replications per condition
res <- runSimulation(design=Design, replications=1000, parallel=TRUE,
                     generate=Generate, analyse=Analyse, summarise=Summarise)
res
View(res)

## save final results to a file upon completion, and play a beep when done
runSimulation(design=Design, replications=1000, parallel=TRUE, filename = 'mysim',
              generate=Generate, analyse=Analyse, summarise=Summarise, beep=TRUE)

## same as above, but send a notification via Pushbullet upon completion
library(RPushbullet) # read-in default JSON file
pushbullet_notifier <- new_PushbulletNotifier(verbose_issues = TRUE)
runSimulation(design=Design, replications=1000, parallel=TRUE, filename = 'mysim',
              generate=Generate, analyse=Analyse, summarise=Summarise,
              notification = 'complete', notifier = pushbullet_notifier)

## Submit as RStudio job (requires job package and active RStudio session)
job::job({
  res <- runSimulation(design=Design, replications=100,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
}, title='t-test simulation')
res  # object res returned to console when completed

## Debug the generate function. See ?browser for help on debugging
##   Type help to see available commands (e.g., n, c, where, ...),
##   ls() to see what has been defined, and type Q to quit the debugger
runSimulation(design=Design, replications=1000,
              generate=Generate, analyse=Analyse, summarise=Summarise,
              parallel=TRUE, debug='generate')

## Alternatively, place a browser() within the desired function line to
##   jump to a specific location
Summarise <- function(condition, results, fixed_objects) {
    #find results of interest here (e.g., alpha < .1, .05, .01)
    browser()
```

```
    ret <- EDR(results[,nms], alpha = .05)
    ret
}

## The following debugs the analyse function for the
## second row of the Design input
runSimulation(design=Design, replications=1000,
              generate=Generate, analyse=Analyse, summarise=Summarise,
              parallel=TRUE, debug='analyse-2')


## End(Not run)


####################################

## Not run:
## Network linked via ssh (two way ssh key-paired connection must be
## possible between master and slave nodes)
##
## Define IP addresses, including primary IP
primary <- '192.168.2.20'
IPs <- list(
    list(host=primary, user='phil', ncore=8),
    list(host='192.168.2.17', user='phil', ncore=8)
)
spec <- lapply(IPs, function(IP)
                  rep(list(list(host=IP$host, user=IP$user)), IP$ncore))
spec <- unlist(spec, recursive=FALSE)

cl <- parallel::makeCluster(type='PSOCK', master=primary, spec=spec)
res <- runSimulation(design=Design, replications=1000, parallel = TRUE,
                     generate=Generate, analyse=Analyse, summarise=Summarise, cl=cl)


## Using parallel='future' to allow the future framework to be used instead
library(future) # future structure to be used internally
plan(multisession) # specify different plan (default is sequential)

res <- runSimulation(design=Design, replications=100, parallel='future',
                     generate=Generate, analyse=Analyse, summarise=Summarise)
head(res)

# The progressr package is used for progress reporting with futures. To redefine
#  use progressr::handlers() (see below)
library(progressr)
with_progress(res <- runSimulation(design=Design, replications=100, parallel='future',
                     generate=Generate, analyse=Analyse, summarise=Summarise))
head(res)

# re-define progressr's bar (below requires cli)
handlers(handler_pbcol(
   adjust = 1.0,
```

```
   complete = function(s) cli::bg_red(cli::col_black(s)),
   incomplete = function(s) cli::bg_cyan(cli::col_black(s))
))

with_progress(res <- runSimulation(design=Design, replications=100, parallel='future',
                     generate=Generate, analyse=Analyse, summarise=Summarise))

# reset future computing plan when complete (good practice)
plan(sequential)


## End(Not run)

####################################

###### Post-analysis: Analyze the results via functions like lm() or SimAnova(), and create
###### tables(dplyr) or plots (ggplot2) to help visualize the results.
###### This is where you get to be a data analyst!


library(dplyr)
res %>% summarise(mean(welch), mean(independent))
res %>% group_by(standard_deviation_ratio, group_size_ratio) %>%
   summarise(mean(welch), mean(independent))

# quick ANOVA analysis method with all two-way interactions
SimAnova( ~ (sample_size + group_size_ratio + standard_deviation_ratio)^2, res,
  rates = TRUE)

# or more specific ANOVAs
SimAnova(independent ~ (group_size_ratio + standard_deviation_ratio)^2,
    res, rates = TRUE)

# make some plots
library(ggplot2)
library(tidyr)
dd <- res %>%
   select(group_size_ratio, standard_deviation_ratio, welch, independent) %>%
   pivot_longer(cols=c('welch', 'independent'), names_to = 'stats')
dd

ggplot(dd, aes(factor(group_size_ratio), value)) + geom_boxplot() +
    geom_abline(intercept=0.05, slope=0, col = 'red') +
    geom_abline(intercept=0.075, slope=0, col = 'red', linetype='dotted') +
    geom_abline(intercept=0.025, slope=0, col = 'red', linetype='dotted') +
    facet_wrap(~stats)

ggplot(dd, aes(factor(group_size_ratio), value, fill = factor(standard_deviation_ratio))) +
    geom_boxplot() + geom_abline(intercept=0.05, slope=0, col = 'red') +
    geom_abline(intercept=0.075, slope=0, col = 'red', linetype='dotted') +
    geom_abline(intercept=0.025, slope=0, col = 'red', linetype='dotted') +
    facet_grid(stats~standard_deviation_ratio) +
    theme(legend.position = 'none')
```

```
#-------------------------------------------------------------------------------
# Example with prepare() function - Loading pre-computed objects

## Not run:

# Step 1: Pre-generate expensive objects offline (can be parallelized)
Design <- createDesign(N = c(10, 20), rho = c(0.3, 0.7))

# Create directory for storing prepared objects
dir.create('prepared_objects', showWarnings = FALSE)

# Generate and save correlation matrices for each condition
for(i in 1:nrow(Design)) {
  cond <- Design[i, ]

  # Generate correlation matrix
  corr_matrix <- matrix(cond$rho, nrow=cond$N, ncol=cond$N)
  diag(corr_matrix) <- 1

  # Create filename based on design parameters
  fname <- paste0('prepared_objects/N', cond$N, '_rho', cond$rho, '.rds')

  # Save to disk
  saveRDS(corr_matrix, file = fname)
}

# Step 2: Use prepare() to load these objects during simulation
prepare <- function(condition, fixed_objects) {
  # Create matching filename
  fname <- paste0('prepared_objects/N', condition$N,
                  '_rho', condition$rho, '.rds')

  # Load the pre-computed correlation matrix
  fixed_objects$corr_matrix <- readRDS(fname)

  return(fixed_objects)
}

generate <- function(condition, fixed_objects) {
  # Use the loaded correlation matrix to generate multivariate data
  dat <- MASS::mvrnorm(n = 50,
                       mu = rep(0, condition$N),
                       Sigma = fixed_objects$corr_matrix)
  data.frame(dat)
}

analyse <- function(condition, dat, fixed_objects) {
  # Calculate mean correlation in generated data
  obs_corr <- cor(dat)
  c(mean_corr = mean(obs_corr[lower.tri(obs_corr)]))
}
```

```
summarise <- function(condition, results, fixed_objects) {
  c(mean_corr = mean(results$mean_corr))
}

# Run simulation - prepare() loads objects efficiently
results <- runSimulation(design = Design,
                         replications = 2,
                         prepare = prepare,
                         generate = generate,
                         analyse = analyse,
                         summarise = summarise,
                         verbose = FALSE)

results

# Cleanup
SimClean(dirs='prepared_objects')


## End(Not run)
```

`rValeMaurelli` *Generate non-normal data with Vale & Maurelli's (1983) method*

Description

Generate multivariate non-normal distributions using the third-order polynomial method described by Vale & Maurelli (1983). If only a single variable is generated then this function is equivalent to the method described by Fleishman (1978).

Usage

```
rValeMaurelli(
  n,
  mean = rep(0, nrow(sigma)),
  sigma = diag(length(mean)),
  skew = rep(0, nrow(sigma)),
  kurt = rep(0, nrow(sigma))
)
```

Arguments

`n`	number of samples to draw
`mean`	a vector of k elements for the mean of the variables
`sigma`	desired k x k covariance matrix between bivariate non-normal variables
`skew`	a vector of k elements for the skewness of the variables
`kurt`	a vector of k elements for the kurtosis of the variables

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Fleishman, A. I. (1978). A method for simulating non-normal distributions. *Psychometrika, 43*, 521-532.

Vale, C. & Maurelli, V. (1983). Simulating multivariate nonnormal distributions. *Psychometrika, 48*(3), 465-471.

Examples

```
set.seed(1)

# univariate with skew
nonnormal <- rValeMaurelli(10000, mean=10, sigma=5, skew=1, kurt=3)
descript(nonnormal)

# multivariate with skew and kurtosis
n <- 10000
r12 <- .4
r13 <- .9
r23 <- .1
cor <- matrix(c(1,r12,r13,r12,1,r23,r13,r23,1),3,3)
sk <- c(1.5,1.5,0.5)
ku <- c(3.75,3.5,0.5)

nonnormal <- rValeMaurelli(n, sigma=cor, skew=sk, kurt=ku)
cor(nonnormal) |> round(3)
descript(nonnormal)
```

Serlin2000 *Empirical detection robustness method suggested by Serlin (2000)*

Description

Hypothesis test to determine whether an observed empirical detection rate, coupled with a given robustness interval, statistically differs from the population value. Uses the methods described by Serlin (2000) as well to generate critical values (similar to confidence intervals, but define a fixed window of robustness). Critical values may be computed without performing the simulation experiment (hence, can be obtained a priori).

Usage

```
Serlin2000(p, alpha, delta, R, CI = 0.95)
```

Arguments

p	(optional) a vector containing the empirical detection rate(s) to be tested. Omitting this input will compute only the CV1 and CV2 values, while including this input will perform a one-sided hypothesis test for robustness
alpha	Type I error rate (e.g., often set to .05)
delta	(optional) symmetric robustness interval around alpha (e.g., a value of .01 when alpha = .05 would test the robustness window .04-.06)
R	number of replications used in the simulation
CI	confidence interval for alpha as a proportion. Default of 0.95 indicates a 95% interval

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Serlin, R. C. (2000). Testing for Robustness in Monte Carlo Studies. *Psychological Methods, 5*, 230-240.

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
# Cochran's criteria at alpha = .05 (i.e., 0.5 +- .01), assuming N = 2000
Serlin2000(p = .051, alpha = .05, delta = .01, R = 2000)

# Bradley's liberal criteria given p = .06 and .076, assuming N = 1000
Serlin2000(p = .060, alpha = .05, delta = .025, R = 1000)
Serlin2000(p = .076, alpha = .05, delta = .025, R = 1000)

# multiple p-values
Serlin2000(p = c(.05, .06, .07), alpha = .05, delta = .025, R = 1000)

# CV values computed before simulation performed
Serlin2000(alpha = .05, R = 2500)
```

SFA *Surrogate Function Approximation via the Generalized Linear Model*

Description

Given a simulation that was executed with runSimulation, potentially with the argument store_results = TRUE to store the unsummarised analysis results, fit a surrogate function approximation (SFA) model to the results and (optionally) perform a root-solving step to solve a target quantity. See Schoemann et al. (2014) for details.

Usage

```
SFA(
  results,
  formula,
  family = "binomial",
  b = NULL,
  design = NULL,
  CI = 0.95,
  interval = NULL,
  ...
)

## S3 method for class 'SFA'
print(x, ...)
```

Arguments

results	data returned from runSimulation. This can be the original results object or the extracted results stored when using store_results = TRUE included to store the analysis results.
formula	formula to specify for the regression model
family	character vector indicating the family of GLMs to use (see family)
b	(optional) Target quantity to use for root solving given the fitted surrogate function (e.g., find sample size associated with SFA implied power of .80)
design	(optional) data.frame object containing all the information relevant for the surrogate model (passed to newdata in predict) with an NA value in the variable to be solved
CI	advertised confidence interval of SFA prediction around solved target
interval	interval to be passed to uniroot if not specified then the lowest and highest values from results for the respective variable will be used
...	additional arguments to pass to glm
x	an object of class SFA

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Schoemann, A. M., Miller, P., Pornprasertmanit, S., and Wu, W. (2014). Using Monte Carlo simulations to determine power and sample size for planned missing designs. *International Journal of Behavioral Development, SAGE Publications, 38*, 471-479.

See Also

runSimulation, SimSolve

Examples

```
## Not run:

# create long Design object to fit surrogate over
Design <- createDesign(N = 100:500,
                       d = .2)
Design

#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 2 --- Define generate, analyse, and summarise functions

Generate <- function(condition, fixed_objects) {
    Attach(condition)
    group1 <- rnorm(N)
    group2 <- rnorm(N, mean=d)
    dat <- data.frame(group = gl(2, N, labels=c('G1', 'G2')),
                      DV = c(group1, group2))
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    p <- c(p = t.test(DV ~ group, dat, var.equal=TRUE)$p.value)
    p
}

Summarise <- function(condition, results, fixed_objects) {
    ret <- EDR(results, alpha = .05)
    ret
}

#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 3 --- Estimate power over N

# Use small number of replications given range of sample sizes
```

```
## note that due to the lower replications disabling the
## RAM printing will help reduce overhead

sim <- runSimulation(design=Design, replications=10,
                     generate=Generate, analyse=Analyse,
                     summarise=Summarise, store_results=TRUE, save=FALSE,
                     progress=FALSE, control=list(print_RAM=FALSE))
sim

# total of 4010 replication
sum(sim$REPLICATIONS)

# use the unsummarised results for the SFA, and include p.values < alpha
sim_results <- SimResults(sim)
sim_results <- within(sim_results, sig <- p < .05)
sim_results

# fitted model
sfa <- SFA(sim_results, formula = sig ~ N)
sfa
summary(sfa)

# plot the observed and SFA expected values
plot(p ~ N, sim, las=1, pch=16, main='Rejection rates with R=10')
pred <- predict(sfa, type = 'response')
lines(sim_results$N, pred, col='red', lty=2)

# fitted model + root-solved solution given f(.) = b,
#   where b = target power of .8
design <- data.frame(N=NA, d=.2)
sfa.root <- SFA(sim_results, formula = sig ~ N,
                b=.8, design=design)
sfa.root

# true root
pwr::pwr.t.test(power=.8, d=.2)


################
# example with smaller range but higher precision
Design <- createDesign(N = 375:425,
                       d = .2)
Design

sim2 <- runSimulation(design=Design, replications=100,
                     generate=Generate, analyse=Analyse,
                     summarise=Summarise, store_results=TRUE, save=FALSE,
                     progress=FALSE, control=list(print_RAM=FALSE))
sim2
sum(sim2$REPLICATIONS) # more replications in total

# use the unsummarised results for the SFA, and include p.values < alpha
sim_results <- SimResults(sim2)
```

```
sim_results <- within(sim_results, sig <- p < .05)
sim_results

# fitted model
sfa <- SFA(sim_results, formula = sig ~ N)
sfa
summary(sfa)

# plot the observed and SFA expected values
plot(p ~ N, sim2, las=1, pch=16, main='Rejection rates with R=100')
pred <- predict(sfa, type = 'response')
lines(sim_results$N, pred, col='red', lty=2)

# fitted model + root-solved solution given f(.) = b,
#   where b = target power of .8
design <- data.frame(N=NA, d=.2)
sfa.root <- SFA(sim_results, formula = sig ~ N,
                b=.8, design=design, interval=c(100, 500))
sfa.root

# true root
pwr::pwr.t.test(power=.8, d=.2)

####################
# vary multiple parameters (e.g., sample size + effect size) to fit
# multi-parameter surrogate

Design <- createDesign(N = seq(from=10, to=500, by=10),
                       d = seq(from=.1, to=.5, by=.1))
Design

sim3 <- runSimulation(design=Design, replications=50,
                      generate=Generate, analyse=Analyse,
                      summarise=Summarise, store_results=TRUE, save=FALSE,
                      progress=FALSE, control=list(print_RAM=FALSE))
sim3
sum(sim3$REPLICATIONS)

# use the unsummarised results for the SFA, and include p.values < alpha
sim_results <- SimResults(sim3)
sim_results <- within(sim_results, sig <- p < .05)
sim_results

# additive effects (logit(sig) ~ N + d)
sfa0 <- SFA(sim_results, formula = sig ~ N+d)
sfa0

# multiplicative effects (logit(sig) ~ N + d + N:d)
sfa <- SFA(sim_results, formula = sig ~ N*d)
sfa

# multiplicative better fit (sample size interacts with effect size)
anova(sfa0, sfa, test = "LRT")
```

```
summary(sfa)

# plot the observed and SFA expected values
library(ggplot2)
sim3$pred <- predict(sfa, type = 'response', newdata=sim3)
ggplot(sim3, aes(N, p, color = factor(d))) +
  geom_point() + geom_line(aes(y=pred)) +
  facet_wrap(~factor(d))

# fitted model + root-solved solution given f(.) = b,
#   where b = target power of .8
design <- data.frame(N=NA, d=.2)
sfa.root <- SFA(sim_results, formula = sig ~ N * d,
                b=.8, design=design, interval=c(100, 500))
sfa.root

# true root
pwr::pwr.t.test(power=.8, d=.2)

# root prediction where d *not* used in original data
design <- data.frame(N=NA, d=.25)
sfa.root <- SFA(sim_results, formula = sig ~ N * d,
                b=.8, design=design, interval=c(100, 500))
sfa.root

# true root
pwr::pwr.t.test(power=.8, d=.25)


## End(Not run)
```

SimAnova *Function for decomposing the simulation into ANOVA-based effect sizes*

Description

Given the results from a simulation with runSimulation form an ANOVA table (without p-values) with effect sizes based on the eta-squared statistic. These results provide approximate indications of observable simulation effects, therefore these ANOVA-based results are generally useful as exploratory rather than inferential tools.

Usage

```
SimAnova(formula, dat, subset = NULL, rates = TRUE)
```

Arguments

formula	an R formula generally of a form suitable for lm or aov. However, if the dependent variable (left size of the equation) is omitted then all the dependent variables in the simulation will be used and the result will return a list of analyses
dat	an object returned from runSimulation of class 'SimDesign'
subset	an optional argument to be passed to subset with the same name. Used to subset the results object while preserving the associated attributes
rates	logical; does the dependent variable consist of rates (e.g., returned from ECR or EDR)? Default is TRUE, which will use the logit of the DV to help stabilize the proportion-based summary statistics when computing the parameters and effect sizes

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
data(BF_sim)

# all results (not usually good to mix Power and Type I results together)
SimAnova(alpha.05.F ~ (groups_equal + distribution)^2, BF_sim)

# only use anova for Type I error conditions
SimAnova(alpha.05.F ~ (groups_equal + distribution)^2, BF_sim, subset = var_ratio == 1)

# run all DVs at once using the same formula
SimAnova(~ groups_equal * distribution, BF_sim, subset = var_ratio == 1)
```

SimCheck *Check for missing files in array simulations*

Description

Given the saved files from a runArraySimulation remote evaluation check whether all .rds files have been saved. If missing the missing row condition numbers will be returned.

Usage

```
SimCheck(dir = NULL, files = NULL, min = 1L, max = NULL)
```

Arguments

dir	character vector input indicating the directory containing the .rds files (see files)
files	vector of file names referring to the saved simulation files. E.g. c('mysim-1.rds', 'mysim-2.rds', ...)
min	minimum number after the '-' deliminator. Default is 1
max	maximum number after the '-' deliminator. If not specified is extracted from the attributes in the first file

Value

returns an invisible list of indices of empty, missing and empty-and-missing row conditions. If no missing then an empty list is returned

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

See Also

runArraySimulation, SimCollect

Examples

```
## Not run:

# if files are in mysimfiles/ directory
SimCheck('mysimfiles')

# specifying files explicility
setwd('mysimfiles/')
SimCheck(files=dir())


## End(Not run)
```

SimClean *Removes/cleans files and folders that have been saved*

Description

This function is mainly used in pilot studies where results and datasets have been temporarily saved by runSimulation but should be removed before beginning the full Monte Carlo simulation (e.g., remove files and folders which contained bugs/biased results).

Usage

```
SimClean(
  ...,
  dirs = NULL,
  temp = TRUE,
  results = FALSE,
  seeds = FALSE,
  save_details = list()
)
```

Arguments

...	one or more character objects indicating which files to remove. Used to remove .rds files which were saved with saveRDS or when using the save and filename inputs to runSimulation
dirs	a character vector indicating which directories to remove
temp	logical; remove the temporary file saved when passing save = TRUE?
results	logical; remove the .rds results files saved when passing save_results = TRUE?
seeds	logical; remove the seed files saved when passing save_seeds = TRUE?
save_details	a list pertaining to information about how and where files were saved (see the corresponding list in runSimulation)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
## Not run:

# remove file called 'results.rds'
SimClean('results.rds')

# remove default temp file
SimClean()

# remove customized saved-results directory called 'mydir'
SimClean(results = TRUE, save_details = list(save_results_dirname = 'mydir'))


## End(Not run)
```

SimCollect *Collapse separate simulation files into a single result*

Description

This function collects and aggregates the results from SimDesign's runSimulation into a single objects suitable for post-analyses, or combines all the saved results directories and combines them into one. This is useful when results are run piece-wise on one node (e.g., 500 replications in one batch, 500 again at a later date, though be careful about the set.seed use as the random numbers will tend to correlate the more it is used) or run independently across different nodes/computing cores (e.g., see runArraySimulation.

Usage

```
SimCollect(
  dir = NULL,
  files = NULL,
  filename = NULL,
  simobj = NULL,
  select = NULL,
  check.only = FALSE,
  target.reps = NULL,
  warning_details = FALSE,
  error_details = TRUE,
  gc = FALSE
)

aggregate_simulations(...)
```

Arguments

dir	a character vector pointing to the directory name containing the .rds files. All .rds files in this directory will be used after first checking their status with SimCheck. For greater specificity use the files argument
files	a character vector containing the names of the simulation's final .rds files.
filename	(optional) name of .rds file to save aggregate simulation file to. If not specified then the results will only be returned in the R console.
simobj	an object returned from runSimulation that was initially passed a design object expanded by expandDesign. This allows the results to be aggregated across the simulation according to the expanded design patter. However, note that the majority of the technical information will only be available in the original simulation object returned from runSimulation
select	a character vector indicating columns to variables to select from the SimExtract(what='results') information. This is mainly useful when RAM is an issue given simulations with many stored estimates. Default includes the results objects in their entirety, though to omit all internally stored simulation results pass the character 'NONE'. To investigate the stored warnings and error messages in isolation pass 'WARNINGS' or 'ERRORS', respectively
check.only	logical; for larger simulations file sets, such as those generated by runArraySimulation, return the design conditions that do no satisfy the target.reps and throw warning if files are unexpectedly missing (the latter criterion is reported from SimCheck, which can be used to obtain the conditions associated with the missing files)
target.reps	(optional) number of replications to check against to evaluate whether the simulation files returned the desired number of replications. If missing, the highest detected value from the collected set of replication information will be used
warning_details	logical; include the aggregate of the warnings to be extracted via SimExtract?
error_details	logical; include the aggregate of the errors to be extracted via SimExtract?
gc	logical; explicitly call R's garbage collector gc? May help when memory is severely constrained during the file read-ins. Otherwise, the select argument should be used to take more memory-friendly subsets
...	not used

Value

returns a data.frame/tibble with the (weighted) average/aggregate of the simulation results

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation, runArraySimulation, SimCheck

Examples

```
## Not run:

setwd('my_working_directory')

## run simulations to save the .rds files (or move them to the working directory)
# seeds1 <- genSeeds(design)
# seeds2 <- genSeeds(design, old.seeds=seeds1)
# ret1 <- runSimulation(design, ..., seed=seeds1, filename='file1')
# ret2 <- runSimulation(design, ..., seed=seeds2, filename='file2')

# saves to the hard-drive and stores in workspace
final <- SimCollect(files = c('file1.rds', 'file2.rds'))
final

# If filename not included, can be extracted from results
# files <- c(SimExtract(ret1, 'filename'), SimExtract(ret2, 'filename'))
# final <- SimCollect(files = files)


################################################
# Example where each row condition is repeated, evaluated independently,
# and later collapsed into a single analysis object

# Each condition repeated four times (hence, replications
# should be set to desired.reps/4)
Design <- createDesign(mu = c(0,5),
                       N  = c(30, 60))
Design

# assume the N=60 takes longer, and should be spread out across more arrays
Design_long <- expandDesign(Design, c(2,2,4,4))
Design_long

replications <- c(rep(50, 4), rep(25,8))
data.frame(Design_long, replications)

#-----------------------------------------------------------------
```

```
Generate <- function(condition, fixed_objects) {
    dat <- with(condition, rnorm(N, mean=mu))
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    ret <- c(mean=mean(dat), SD=sd(dat))
    ret
}

Summarise <- function(condition, results, fixed_objects) {
    ret <- colMeans(results)
    ret
}

#---------------------------------------------------------------------

# create directory to store all final simulation files
dir.create('sim_files/')

iseed <- genSeeds()

# distribute jobs independently
sapply(1:nrow(Design_long), \(i) {
  runArraySimulation(design=Design_long, replications=replications,
                generate=Generate, analyse=Analyse, summarise=Summarise,
                arrayID=i, dirname='sim_files/', filename='job', iseed=iseed)
}) |> invisible()

# check for any missed files
SimCheck(dir='sim_files/')

# check that all replications satisfy target
SimCollect('sim_files/', check.only = TRUE)


# specify files explicitly
SimCollect(files = list.files(path='sim_files/', pattern="*.rds", full.names=TRUE),
   check.only = TRUE)

# this would have been returned were the target.rep supposed to be 1000
SimCollect('sim_files/', check.only = TRUE, target.reps=1000)

# aggregate into single object
sim <- SimCollect('sim_files/')
sim

# view list of error messages (if there were any raised)
SimCollect('sim_files/', select = 'ERRORS')

SimClean(dir='sim_files/')

######################################################################
```

```
# Similar to the above, however implemented using the less flexible
# and more memory intensive runSimulation() approach

# objects borrowed from above as the logic is the same
data.frame(Design_long, replications) |> head()

iseed <- genSeeds()

# seed must be of length 1
long_final <- runSimulation(Design_long, replications=replications,
                generate=Generate, analyse=Analyse, summarise=Summarise,
                seed=iseed)
long_final

# aggregate simulation
final <- SimCollect(simobj=long_final)
final


## End(Not run)
```

SimDesign *Structure for Organizing Monte Carlo Simulation Designs*

Description

Structure for Organizing Monte Carlo Simulation Designs

Details

Provides tools to help organize Monte Carlo simulations in R. The package controls the structure and back-end of Monte Carlo simulations by utilizing a general generate-analyse-summarise strategy. The functions provided control common simulation issues such as re-simulating non-convergent results, support parallel back-end computations with proper random number generation within each simulation condition, save and restore temporary files, aggregate results across independent nodes, and provide native support for debugging. The primary function for organizing the simulations is `runSimulation`, while for array jobs submitting to HPC clusters (e.g., SLURM) see `runArraySimulation` and the associated package vignettes.

For an in-depth tutorial of the package please refer to Chalmers and Adkins (2020; doi:10.20982/tqmp.16.4.p248). For an earlier didactic presentation of the package users can refer to Sigal and Chalmers (2016; doi:10.1080/10691898.2016.1246953). Finally, see the associated wiki on Github (`https://github.com/philchalmers/SimDesign/wiki`) for other tutorial material, examples, and applications of `SimDesign` to real-world simulations.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

SimExtract *Function to extract extra information from SimDesign objects*

Description

Function used to extract any error or warnings messages, the seeds associated with any error or warning messages, and any analysis results that were stored in the final simulation object.

Usage

```
SimExtract(object, what, fuzzy = TRUE, append = TRUE)
```

Arguments

object	object returned from runSimulation
what	character vector indicating what information to extract, written in agnostic casing (e.g., 'ERRORS' and 'errors' are equivalent). Possible inputs include

'errors' to return a tibble object containing counts of any error messages

'warnings' to return a data.frame object containing counts of any warning messages

'seeds' for the specified random number generation seeds

'Random.seeds' for the complete list of .Random.seed states across replications (only stored when runSimulation(..., control = list(store_Random.seeds=TRUE)))

'log_times' for the per replication generate/analyse execution times (recorded in seconds)

'error_seeds' **and** 'warning_seeds' to extract the associated .Random.seed values associated with the ERROR/WARNING messages

'prepare_seeds' to extract the .Random.seed states captured before prepare() was called for each condition

'prepare_error_seed' to extract the .Random.seed state when prepare() encountered an error (useful for debugging with load_seed_prepare)

'results' to extract the simulation results if the option store_results was passed to runSimulation,

'filename' **and** 'save_results_dirname' for extracting the saved file/directory name information (if used), 'functions' to extract the defined functions used in the experiment

'design' to extract the original design object

Note that 'warning_seeds' are not stored automatically in simulations and require passing store_warning_seeds = TRUE to runSimulation.

fuzzy — logical; use fuzzy string matching to reduce effectively identical messages? For example, when attempting to invert a matrix the error message *"System is computationally singular: reciprocal condition number = 1.92747e-17"* and *"System is computationally singular: reciprocal condition number = 2.15321e-16"* are effectively the same, and likely should be reported in the same columns of the extracted output

append — logical; append the design conditions when extracting error/warning messages?

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

Examples

```
## Not run:

Generate <- function(condition, fixed_objects) {
    int <- sample(1:10, 1)
    if(int > 5) warning('GENERATE WARNING: int greater than 5')
    if(int == 1) stop('GENERATE ERROR: integer is 1')
    rnorm(5)
}

Analyse <- function(condition, dat, fixed_objects) {
    int <- sample(1:10, 1)
    if(int > 5) warning('ANALYSE WARNING: int greater than 5')
    if(int == 1) stop('ANALYSE ERROR: int is 1')
    c(ret = 1)
}

Summarise <- function(condition, results, fixed_objects) {
    mean(results)
}

res <- runSimulation(replications = 100, seed=1234,
                     generate=Generate, analyse=Analyse, summarise=Summarise)
res

SimExtract(res, what = 'errors')
SimExtract(res, what = 'warnings')
```

```
seeds <- SimExtract(res, what = 'error_seeds')
seeds[,1:3]

# replicate a specific error for debugging (type Q to exit debugger)
res <- runSimulation(replications = 100, load_seed=seeds[,1], debug='analyse',
                     generate=Generate, analyse=Analyse, summarise=Summarise)



## End(Not run)
```

SimFunctions *Template-based generation of the Generate-Analyse-Summarise functions*

Description

This function prints template versions of the required Design and Generate-Analyse-Summarise functions for SimDesign to run simulations. Templated output comes complete with the correct inputs, class of outputs, and optional comments to help with the initial definitions. Use this at the start of your Monte Carlo simulation study. Following the definition of the SimDesign template file please refer to detailed the information in runSimulation for how to edit this template to make a working simulation study.

Usage

```
SimFunctions(
  filename = NULL,
  dir = getwd(),
  save_structure = "single",
  extra_file = FALSE,
  nAnalyses = 1,
  nGenerate = 1,
  summarise = TRUE,
  comments = FALSE,
  openFiles = TRUE,
  clip = FALSE,
  spin_header = TRUE,
  SimSolve = FALSE
)
```

Arguments

filename	a character vector indicating whether the output should be saved to two respective files containing the simulation design and the functional components, respectively. Using this option is generally the recommended approach when beginning to write a Monte Carlo simulation

dir	the directory to write the files to. Default is the working directory
save_structure	character indicating the number of files to break the simulation code into when filename is included (default is 'single' for one file). When save_structure = 'double' the output is saved to two separate files containing the functions and design definitions, and when save_structure = 'all' the generate, analyse, summarise, and execution code are all saved into separate files. The purpose of this structure is because multiple structured files often makes organization and debugging slightly easier for larger Monte Carlo simulations, though, in principle, all files could be stored in a single R script
extra_file	logical; should an extra file be saved containing user-defined functions or objects? Default is FALSE
nAnalyses	number of analysis functions to create (default is 1). Increasing the value of this argument when independent analysis are being performed allows function definitions to be better partitioned and potentially more modular
nGenerate	number of generate functions to create (default is 1). Increase the value of this argument when when the data generation functions are very different and should be isolated from each other (otherwise, if there is much in common between the generate steps, the default of 1 should be preferred). Otherwise, if nGenerate == 0 then no generate function will be provided and instead this data-generation step can be defined in the analysis function(s) (only recommended for smaller simulations)
summarise	include summarise function? Default is TRUE
comments	logical; include helpful comments? Default is FALSE
openFiles	logical; after files have been generated, open them in your text editor (e.g., if Rstudio is running the scripts will open in a new tab)?
clip	logical; use the 'clipr' package to copy the simulation template to the OS clipboard?
spin_header	logical; include a basic knitr::spin header to allow the simulation to be knitted? Default is TRUE. For those less familiar with spin documents see https://bookdown.org/yihui/rm for further details
SimSolve	logical; should the template be generated that is intended for a SimSolve implementation? Default is FALSE

Details

The recommended approach to organizing Monte Carlo simulation files is to first save the template generated by this function to the hard-drive by passing a suitable filename argument (which, if users are interacting with R via the RStudio IDE, will also open the template file after it has been saved). For larger simulations, two separate files could also be used (achieved by changing out.files), and may be easier for debugging/sourcing the simulation code; however, this is a matter of preference and does not change any functionality in the package.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
SimFunctions()
SimFunctions(comments = TRUE) #with helpful comments

## Not run:

# write output files to a single file with comments
SimFunctions('mysim', comments = TRUE)

# Multiple analysis functions for optional partitioning
SimFunctions(nAnalyses = 2)
SimFunctions(nAnalyses = 3)

# Multiple analysis + generate functions
SimFunctions(nAnalyses = 2, nGenerate=2)

# save multiple files for the purpose of designing larger simulations
#  (also include extra_file for user-defined objects/functions)
SimFunctions('myBigSim', save_structure = 'all',
   nAnalyses = 3, nGenerate=2, extra_file = TRUE)



## End(Not run)
```

SimResults *Function to read in saved simulation results*

Description

If runSimulation was passed the flag save_results = TRUE then the row results corresponding to the design object will be stored to a suitable sub-directory as individual .rds files. While users could use readRDS directly to read these files in themselves, this convenience function will read the desired rows in automatically given the returned object from the simulation. Can be used to read in 1 or more .rds files at once (if more than 1 file is read in then the result will be stored in a list).

Usage

```
SimResults(obj, which, prefix = "results-row", wd = getwd(), rbind = FALSE)
```

Arguments

obj	object returned from runSimulation where save_results = TRUE or store_results was used. If the former then the remaining function arguments can be useful for reading in specific files. Alternatively, the object can be from the Spower package, evaluated using either Spower() or SpowerBatch()
which	a numeric vector indicating which rows should be read in. If missing, all rows will be read in
prefix	character indicating prefix used for stored files
wd	working directory; default is found with getwd.
rbind	logical; should the results be combined by row or returned as a list? Only applicable when the supplied obj was obtained from the function Spower::SpowerBatch()

Value

the returned result is either a nested list (when length(which) > 1) or a single list (when length(which) == 1) containing the simulation results. Each read-in result refers to a list of 4 elements:

condition the associate row (ID) and conditions from the respective design object

results the object with returned from the analyse function, potentially simplified into a matrix or data.frame

errors a table containing the message and number of errors that caused the generate-analyse steps to be rerun. These should be inspected carefully as they could indicate validity issues with the simulation that should be noted

warnings a table containing the message and number of non-fatal warnings which arose from the analyse step. These should be inspected carefully as they could indicate validity issues with the simulation that should be noted

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

descript

Examples

```
## Not run:

# store results (default behaviour)
sim <- runSimulation(..., store_results = TRUE)
SimResults(sim)

# store results to drive if RAM issues are present
obj <- runSimulation(..., save_results = TRUE)

# row 1 results
row1 <- SimResults(obj, 1)

# rows 1:5, stored in a named list
rows_1to5 <- SimResults(obj, 1:5)

# all results
rows_all <- SimResults(obj)


## End(Not run)
```

SimShiny *Generate a basic Monte Carlo simulation GUI template*

Description

This function generates suitable stand-alone code from the shiny package to create simple web-interfaces for performing single condition Monte Carlo simulations. The template generated is relatively minimalistic, but allows the user to quickly and easily edit the saved files to customize the associated shiny elements as they see fit.

Usage

```
SimShiny(filename = NULL, dir = getwd(), design, ...)
```

Arguments

filename	an optional name of a text file to save the server and UI components (e.g., 'mysimGUI.R'). If omitted, the code will be printed to the R console instead
dir	the directory to write the files to. Default is the working directory
design	design object from runSimulation
...	arguments to be passed to runSimulation. Note that the design object is not used directly, and instead provides options to be selected in the GUI

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

runSimulation

Examples

```
## Not run:

Design <- createDesign(sample_size = c(30, 60, 90, 120),
                       group_size_ratio = c(1, 4, 8),
                       standard_deviation_ratio = c(.5, 1, 2))

Generate <- function(condition, fixed_objects) {
    N <- condition$sample_size
    grs <- condition$group_size_ratio
    sd <- condition$standard_deviation_ratio
    if(grs < 1){
        N2 <- N / (1/grs + 1)
        N1 <- N - N2
    } else {
        N1 <- N / (grs + 1)
        N2 <- N - N1
    }
    group1 <- rnorm(N1)
    group2 <- rnorm(N2, sd=sd)
    dat <- data.frame(group = c(rep('g1', N1), rep('g2', N2)), DV = c(group1, group2))
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    welch <- t.test(DV ~ group, dat)
    ind <- t.test(DV ~ group, dat, var.equal=TRUE)

    # In this function the p values for the t-tests are returned,
    #  and make sure to name each element, for future reference
    ret <- c(welch = welch$p.value, independent = ind$p.value)
    ret
}

Summarise <- function(condition, results, fixed_objects) {
    #find results of interest here (e.g., alpha < .1, .05, .01)
    ret <- EDR(results, alpha = .05)
    ret
}
```

```
# test that it works
# Final <- runSimulation(design=Design, replications=5,
#                        generate=Generate, analyse=Analyse, summarise=Summarise)

# print code to console
SimShiny(design=Design, generate=Generate, analyse=Analyse,
         summarise=Summarise)

# save shiny code to file
SimShiny('app.R', design=Design, generate=Generate, analyse=Analyse,
         summarise=Summarise)

# run the application
shiny::runApp()
shiny::runApp(launch.browser = TRUE) # in web-browser


## End(Not run)
```

SimSolve *One Dimensional Root (Zero) Finding in Simulation Experiments*

Description

Function provides a stochastic root-finding approach to solve specific quantities in simulation experiments (e.g., solving for a specific sample size to meet a target power rate) using the Probablistic Bisection Algorithm with Bolstering and Interpolations (ProBABLI; Chalmers, 2024). The structure follows the three functional steps outlined in `runSimulation`, however portions of the `design` input are taken as variables to be estimated rather than fixed, where an additional constant `b` is required in order to solve the root equation `f(x) - b = 0`.

Usage

```
SimSolve(
  design,
  interval,
  b,
  generate,
  analyse,
  summarise,
  replications = list(burnin.iter = 15L, burnin.reps = 50L, max.reps = 500L,
    min.total.reps = 9000L, increase.by = 10L),
  integer = TRUE,
  formula = y ~ poly(x, 2),
  family = "binomial",
  parallel = FALSE,
  cl = NULL,
```

```
  save = TRUE,
  resume = TRUE,
  method = "ProBABLI",
  wait.time = NULL,
  ncores = parallelly::availableCores(omit = 1L),
  type = ifelse(.Platform$OS.type == "windows", "PSOCK", "FORK"),
  maxiter = 100L,
  check.interval = TRUE,
  predCI = 0.95,
  predCI.tol = NULL,
  lastSolve = NULL,
  verbose = interactive(),
  control = list(),
  ...
)

## S3 method for class 'SimSolve'
summary(object, tab.only = FALSE, reps.cutoff = 300, ...)

## S3 method for class 'SimSolve'
plot(x, y, ...)
```

Arguments

design	a tibble or data.frame object containing the Monte Carlo simulation conditions to be studied, where each row represents a unique condition and each column a factor to be varied (see createDesign). However, exactly one column of this object in each row must be specified with NA placeholders to indicate that the missing value should be estimated via the select stochastic optimizer
interval	a vector of length two, or matrix with nrow(design) rows and two columns, containing the end-points of the interval to be searched per row condition. If a vector then the interval will be used for all rows in the supplied design object when passed to the PBA engine
b	a single constant used to solve the root equation f(x) - b = 0
generate	generate function. See runSimulation
analyse	analysis function. See runSimulation
summarise	summary function that returns a single number corresponding to the function evaluation f(x) in the equation f(x) = b to be solved as a root f(x) - b = 0. Unlike in the standard runSimulation() definitions this input is required. For further information on this function specification, see runSimulation
replications	a named list or vector indicating the number of replication to use for each design condition per PBA iteration. By default the input is a list with the arguments burnin.iter = 15L, specifying the number of burn-in iterations to used, burnin.reps = 50L to indicate how many replications to use in each burn-in iteration, max.reps = 500L to prevent the replications from increasing higher than this number, min.total.reps = 9000L to avoid termination when very few replications have been explored (lower bound of the replication budget),

and increase.by = 10L to indicate how many replications to increase per iteration after the burn-in stage. Default can overwritten by explicit definition (e.g., replications = list(increase.by = 25L)).

Vector inputs can specify the exact replications for each respective iteration. As a general rule, early iterations should be relatively low for initial searches to avoid unnecessary computations when locating the approximate location of the root, while the number of replications should gradually increase after this burn-in to reduce the sampling variability.

integer — logical; should the values of the root be considered integer or numeric? If TRUE then bolstered directional decisions will be made in the PBA function based on the collected sampling history

formula — regression formula to use when interpolate = TRUE. Default fits an orthogonal polynomial of degree 2

family — family argument passed to glm. By default the 'binomial' family is used, as this function defaults to power analysis setups where isolated results passed to summarise will return 0/1s, however other families should be used if summarise returns something else (e.g., if solving for a particular standard error then a 'gaussian' family would be more appropriate).

Note that if individual results from the analyse steps should not be used (i.e., only the aggregate from summarise is meaningful) then set control = list(summarise.reg_data = TRUE) to override the default behaviour, thereby using only the aggregate information and weights

parallel — for parallel computing for slower simulation experiments (see runSimulation for details)

cl — see runSimulation

save — logical; store temporary file in case of crashes. If detected in the working directory will automatically be loaded to resume (see runSimulation for similar behaviour)

resume — logical; if a temporary SimDesign file is detected should the simulation resume from this location? Keeping this TRUE is generally recommended, however this should be disabled if using SimSolve within runSimulation to avoid reading improper save states

method — optimizer method to use. Default is the stochastic root-finder 'ProBABLI', but can also be the deterministic options 'Brent' (which uses the function uniroot) or 'bisection' for the classical bisection method. If using deterministic root-finders then replications must either equal a single constant to reflect the number of replication to use per deterministic iteration or be a vector of length maxiter to indicate the replications to use per iteration

wait.time — (optional) argument passed to PBA to indicate the time to wait (specified in minutes if a numeric vector is passed) per row in the Design object rather than using pre-determined termination criteria based on the estimates. For example, if three three conditions were defined in Design, and wait.time="5", then the total search time till terminate after 15 minutes regardless of independently specified termination criteria in control. See timeFormater for alternative specifications

ncores — see runSimulation

type — type of cluster (see makeCluster) or plotting type to use.

If type used in plot then can be 'density' to plot the density of the iteration history after the burn-in stage, 'iterations' for a bubble plot with inverse replication weights. If not specified then the default PBA plots are provided (see PBA)

maxiter — the maximum number of iterations (default 100) except when wait.time is specified (automatically increased to 3000 to avoid early termination)

check.interval — logical; should an initial check be made to determine whether f(interval[1L]) and f(interval[2L]) have opposite signs? If FALSE, the specified interval is assumed to contain a root, where f(interval[1]) < 0 and f(interval[2] > 0. Default is TRUE

predCI — advertised confidence interval probability for final model-based prediction of target b given the root input estimate. Returned as an element in the summary() list output

predCI.tol — (optional) rather than relying on the changes between successive estimates (default), if the predicting CI is consistently within this supplied tolerance range then the search will be terminated. This provides termination behaviour based on the predicted precision of the root solutions rather than their stability history, and therefore can be used to obtain estimates with a particular level of advertised accuracy. For example, when solving for a sample size value (N) if the solution associated with b = .80 requires that the advertised 95 is consistently between [.795, .805] then predCI.tol = .01 should be used to reflect this tolerance range

lastSolve — stub for Spower package; not to be used by front-end users

verbose — logical; print information to the console?

control — a list of the algorithm control parameters. If not specified, the defaults described below are used.

- tol tolerance criteria for early termination (.1 for integer = TRUE searches; .00025 for non-integer searches
- rel.tol relative tolerance criteria for early termination (default .0001)
- k.success number of consecutive tolerance successes given rel.tol and tol criteria (default is 3)
- bolster logical; should the PBA evaluations use bolstering based on previous evaluations? Default is TRUE, though only applicable when integer = TRUE
- interpolate.R number of replications to collect prior to performing the interpolation step (default is 3000 after accounting for data exclusion from burnin.iter). Setting this to 0 will disable any interpolation computations
- include_reps logical; include a column in the condition elements to indicate how many replications are currently being evaluated? Mainly useful when further tuning within each ProBABLI iteration is desirable (e.g., for increasing/decreasing bootstrap draws as the search progresses). Default is FALSE

summarise.reg_data logical; should the aggregate results from Summarise (along with its associated weights) be used for the interpolation steps, or the raw data from the Analyse step? Set this to TRUE when the individual results from Analyse give less meaningful information

... additional arguments to be pasted to PBA

object object of class 'SimSolve'

tab.only logical; print only the (reduced) table of estimates?

reps.cutoff integer indicating the rows to omit from the output if the number of replications are less than this value

x object of class 'SimSolve'

y design row to plot. If omitted defaults to 1

Details

Root finding is performed using a progressively bolstered version of the probabilistic bisection algorithm (PBA) to find the associated root given the noisy simulation objective function evaluations. Information is collected throughout the search to make more accurate predictions about the associated root via interpolation. If interpolations fail, then the last iteration of the PBA search is returned as the best guess.

For greater advertised accuracy with ProBABLI, termination criteria can be based on the width of the advertised predicting interval (via predCI.tol) or by specifying how long the investigator is willing to wait for the final estimates (via wait.time, where longer wait times lead to progressively better accuracy in the final estimates).

Value

the filled-in design object containing the associated lower and upper interval estimates from the stochastic optimization

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

Chalmers, R. P. (2024). Solving Variables with Monte Carlo Simulation Experiments: A Stochastic Root-Solving Approach. Psychological Methods. DOI: 10.1037/met0000689

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

See Also

SFA

Examples

```
## Not run:

#########################
## A Priori Power Analysis
#########################

# GOAL: Find specific sample size in each group for independent t-test
# corresponding to a power rate of .8
#
# For ease of the setup, assume the groups are the same size, and the mean
# difference corresponds to Cohen's d values of .2, .5, and .8
# This example can be solved numerically using the pwr package (see below),
# though the following simulation setup is far more general and can be
# used for any generate-analyse combination of interest

# SimFunctions(SimSolve=TRUE)

#### Step 1 --- Define your conditions under study and create design data.frame.
#### However, use NA placeholder for sample size as it must be solved,
#### and add desired power rate to object

Design <- createDesign(N = NA,
                       d = c(.2, .5, .8),
                       sig.level = .05)
Design    # solve for NA's

#~~~~~~~~~~~~~~~~~~~~~~~
#### Step 2 --- Define generate, analyse, and summarise functions

Generate <- function(condition, fixed_objects) {
    Attach(condition)
    group1 <- rnorm(N)
    group2 <- rnorm(N, mean=d)
    dat <- data.frame(group = gl(2, N, labels=c('G1', 'G2')),
                      DV = c(group1, group2))
    dat
}

Analyse <- function(condition, dat, fixed_objects) {
    p <- t.test(DV ~ group, dat, var.equal=TRUE)$p.value
    p
}

Summarise <- function(condition, results, fixed_objects) {
    # Must return a single number corresponding to f(x) in the
    # root equation f(x) = b

    ret <- c(power = EDR(results, alpha = condition$sig.level))
    ret
}
```

```
#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 3 --- Optimize N over the rows in design

### (For debugging) may want to see if simulation code works as intended first
### for some given set of inputs
# runSimulation(design=createDesign(N=100, d=.8, sig.level=.05),
#               replications=10, generate=Generate, analyse=Analyse,
#               summarise=Summarise)

# Initial search between N = [10,500] for each row using the default
   # integer solver (integer = TRUE). In this example, b = target power
solved <- SimSolve(design=Design, b=.8, interval=c(10, 500),
                generate=Generate, analyse=Analyse,
                summarise=Summarise)
solved
summary(solved)
plot(solved, 1)
plot(solved, 2)
plot(solved, 3)

# also can plot median history and estimate precision
plot(solved, 1, type = 'history')
plot(solved, 1, type = 'density')
plot(solved, 1, type = 'iterations')

# verify with true power from pwr package
library(pwr)
pwr.t.test(d=.2, power = .8) # sig.level/alpha = .05 by default
pwr.t.test(d=.5, power = .8)
pwr.t.test(d=.8, power = .8)

# use estimated N results to see how close power was
N <- solved$N
pwr.t.test(d=.2, n=N[1])
pwr.t.test(d=.5, n=N[2])
pwr.t.test(d=.8, n=N[3])

# with rounding
N <- ceiling(solved$N)
pwr.t.test(d=.2, n=N[1])
pwr.t.test(d=.5, n=N[2])
pwr.t.test(d=.8, n=N[3])

### failing analytic formula, confirm results with more precise
###  simulation via runSimulation()
###  (not required, if accuracy is important then ProBABLI should be run longer)
# csolved <- solved
# csolved$N <- ceiling(solved$N)
# confirm <- runSimulation(design=csolved, replications=10000, parallel=TRUE,
#                          generate=Generate, analyse=Analyse,
#                          summarise=Summarise)
# confirm
```

```
# Similarly, terminate if the prediction interval is consistently predicted
#   to be between [.795, .805]. Note that maxiter increased as well
solved_predCI <- SimSolve(design=Design, b=.8, interval=c(10, 500),
                     generate=Generate, analyse=Analyse, summarise=Summarise,
                     maxiter=200, predCI.tol=.01)
solved_predCI
summary(solved_predCI) # note that predCI.b are all within [.795, .805]

N <- solved_predCI$N
pwr.t.test(d=.2, n=N[1])
pwr.t.test(d=.5, n=N[2])
pwr.t.test(d=.8, n=N[3])

# Alternatively, and often more realistically, wait.time can be used
# to specify how long the user is willing to wait for a final estimate.
# Solutions involving more iterations will be more accurate,
# and therefore it is recommended to run the ProBABLI root-solver as long
# the analyst can tolerate if the most accurate estimates are desired.
# Below executes the simulation for 2 minutes per condition

solved_2min <- SimSolve(design=Design[1, ], b=.8, interval=c(10, 500),
                generate=Generate, analyse=Analyse, summarise=Summarise,
                wait.time="2")
solved_2min
summary(solved_2min)

# use estimated N results to see how close power was
N <- solved_2min$N
pwr.t.test(d=.2, n=N[1])


#-----------------------------------------------

#######################
## Sensitivity Analysis
#######################

# GOAL: solve effect size d given sample size and power inputs (inputs
# for root no longer required to be an integer)

# Generate-Analyse-Summarise functions identical to above, however
# Design input includes NA for d element
Design <- createDesign(N = c(100, 50, 25),
                       d = NA,
                       sig.level = .05)
Design    # solve for NA's

#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 2 --- Define generate, analyse, and summarise functions (same as above)

#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 3 --- Optimize d over the rows in design
# search between d = [.1, 2] for each row
```

```
# In this example, b = target power
# note that integer = FALSE to allow smooth updates of d
solved <- SimSolve(design=Design, b = .8, interval=c(.1, 2),
                   generate=Generate, analyse=Analyse,
                   summarise=Summarise, integer=FALSE)
solved
summary(solved)
plot(solved, 1)
plot(solved, 2)
plot(solved, 3)

# plot median history and estimate precision
plot(solved, 1, type = 'history')
plot(solved, 1, type = 'density')
plot(solved, 1, type = 'iterations')

# verify with true power from pwr package
library(pwr)
pwr.t.test(n=100, power = .8)
pwr.t.test(n=50, power = .8)
pwr.t.test(n=25, power = .8)

# use estimated d results to see how close power was
pwr.t.test(n=100, d = solved$d[1])
pwr.t.test(n=50, d = solved$d[2])
pwr.t.test(n=25, d = solved$d[3])

### failing analytic formula, confirm results with more precise
###  simulation via runSimulation() (not required; if accuracy is important
###  PROBABLI should just be run longer)
# confirm <- runSimulation(design=solved, replications=10000, parallel=TRUE,
#                          generate=Generate, analyse=Analyse,
#                          summarise=Summarise)
# confirm


#-----------------------------------------------

#####################
## Criterion Analysis
#####################

# GOAL: solve Type I error rate (alpha) given sample size, effect size, and
# power inputs (inputs for root no longer required to be an integer). Only useful
# when Type I error is less important than achieving the desired 1-beta (power)

Design <- createDesign(N = 50,
                       d = c(.2, .5, .8),
                       sig.level = NA)
Design    # solve for NA's

# all other function definitions same as above
```

```
# search for alpha within [.0001, .8]
solved <- SimSolve(design=Design, b = .8, interval=c(.0001, .8),
                   generate=Generate, analyse=Analyse,
                   summarise=Summarise, integer=FALSE)
solved
summary(solved)
plot(solved, 1)
plot(solved, 2)
plot(solved, 3)

# plot median history and estimate precision
plot(solved, 1, type = 'history')
plot(solved, 1, type = 'density')
plot(solved, 1, type = 'iterations')

# verify with true power from pwr package
library(pwr)
pwr.t.test(n=50, power = .8, d = .2, sig.level=NULL)
pwr.t.test(n=50, power = .8, d = .5, sig.level=NULL)
pwr.t.test(n=50, power = .8, d = .8, sig.level=NULL)

# use estimated alpha results to see how close power was
pwr.t.test(n=50, d = .2, sig.level=solved$sig.level[1])
pwr.t.test(n=50, d = .5, sig.level=solved$sig.level[2])
pwr.t.test(n=50, d = .8, sig.level=solved$sig.level[3])

### failing analytic formula, confirm results with more precise
###  simulation via runSimulation() (not required; if accuracy is important
###  PROBABLI should just be run longer)
# confirm <- runSimulation(design=solved, replications=10000, parallel=TRUE,
#                          generate=Generate, analyse=Analyse,
#                          summarise=Summarise)
# confirm


## End(Not run)
```

`Summarise` *Summarise simulated data using various population comparison statistics*

Description

This collapses the simulation results within each condition to composite estimates such as RMSE, bias, Type I error rates, coverage rates, etc. See the `See Also` section below for useful functions to be used within `Summarise`.

Usage

```
Summarise(condition, results, fixed_objects)
```

Arguments

condition	a single row from the design input from runSimulation (as a data.frame), indicating the simulation conditions
results	a tibble data frame (if Analyse returned a named numeric vector of any length) or a list (if Analyse returned a list or multi-rowed data.frame) containing the analysis results from Analyse, where each cell is stored in a unique row/list element
fixed_objects	object passed down from runSimulation

Value

for best results should return a named numeric vector or data.frame with the desired meta-simulation results. Named list objects can also be returned, however the subsequent results must be extracted via SimExtract

References

Chalmers, R. P., & Adkins, M. C. (2020). Writing Effective and Reliable Monte Carlo Simulations with the SimDesign Package. The Quantitative Methods for Psychology, 16(4), 248-280. doi:10.20982/tqmp.16.4.p248

Sigal, M. J., & Chalmers, R. P. (2016). Play it again: Teaching statistics with Monte Carlo simulation. Journal of Statistics Education, 24(3), 136-156. doi:10.1080/10691898.2016.1246953

See Also

bias, RMSE, RE, EDR, ECR, MAE, SimExtract

Examples

```
## Not run:

summarise <- function(condition, results, fixed_objects) {

    #find results of interest here (alpha < .1, .05, .01)
    lessthan.05 <- EDR(results, alpha = .05)

    # return the results that will be appended to the design input
    ret <- c(lessthan.05=lessthan.05)
    ret
}


## End(Not run)
```

`timeFormater` *Format time string to suitable numeric output*

Description

Format time input string into suitable numeric output metric (e.g., seconds). Input follows the `SBATCH` utility specifications. Accepted time formats include `"minutes"`, `"minutes:seconds"`, `"hours:minutes:seconds"`, `"days-hours"`, `"days-hours:minutes"` and `"days-hours:minutes:seconds"`. Alternatively, function can be used to convert numeric input to SBATCH format.

Usage

```
timeFormater(time, output = "sec", input = "min", sround = floor)
```

Arguments

`time`	a character string to be formatted. If a numeric vector is supplied then this will be interpreted as minutes due to character coercion.
`output`	type of numeric output to convert time into. Currently supported are `'sec'` for seconds (default), `'min'` for minutes, `'hour'`, and `'day'`. Alternatively, if `time` were numeric then setting `output` to `'SBATCH'` will return a suitable SBATCH format.
`input`	if supplied `time` is a numeric, indicates what the value represents. Default assumes the input is in minutes (see `output` for supported values)
`sround`	function used to round last seconds computation

Details

For example, `time = "60"` indicates a maximum time of 60 minutes, `time = "03:00:00"` a maximum time of 3 hours, `time = "4-12"` a maximum of 4 days and 12 hours, and `time = "2-02:30:00"` a maximum of 2 days, 2 hours and 30 minutes.

Examples

```
# Test cases (outputs in seconds)
timeFormater("4-12")        # day-hours
timeFormater("4-12:15")     # day-hours:minutes
timeFormater("4-12:15:30")  # day-hours:minutes:seconds

timeFormater("30")          # minutes
timeFormater("30:30")       # minutes:seconds
timeFormater("4:30:30")     # hours:minutes:seconds

# output in hours
timeFormater("4-12", output = 'hour')
timeFormater("4-12:15", output = 'hour')
timeFormater("4-12:15:30", output = 'hour')
```

```
timeFormater("30", output = 'hour')
timeFormater("30:30", output = 'hour')
timeFormater("4:30:30", output = 'hour')

# numeric input is understood as minutes
timeFormater(42)               # seconds
timeFormater(42, output='min') # minutes

# convert numeric inputs to SBATCH format
timeFormater(60, output='SBATCH')
timeFormater(3, output='SBATCH', input='day')
timeFormater(7000, output='SBATCH', input='sec')
timeFormater(100000, output='SBATCH', input='sec')

# rounding seconds
timeFormater(1.55555, output='SBATCH', input='sec') # floor default
timeFormater(1.55555, output='SBATCH', input='sec', sround=ceiling)
timeFormater(1.55555, output='SBATCH', input='sec', sround=\(x) round(x, 3))
```

Index