Package ‘SimDesign’

August 9, 2022

Title Structure for Organizing Monte Carlo Simulation Designs

Version 2.9

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, and implicitly supports parallel processing.
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 (>= 3.5.0)

Imports foreach, methods, parallel, dplyr, sessioninfo, beepr, pbapply (>= 1.3-0), RPushbullet, stats

Suggests knitr, ggplot2, tidyr, purrr, shiny, doMPI, copula, extraDistr, renv, rmarkdown

License GPL (>= 2)

ByteCompile yes

LazyData true

URL `https://github.com/philchalmers/SimDesign`,
`https://github.com/philchalmers/SimDesign/wiki`

RoxygenNote 7.2.1

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2022-08-09 16:10:02 UTC

R topics documented:

add_missing . . . 3
aggregate_simulations . . . 5
Analyse . . . 6
AnalyseIf . . . 8
Attach . . . 10
BF_sim . . . 12
BF_sim_alternative . . . 13
bias . . . 13
boot_predict . . . 16
CC . . . 18
createDesign . . . 19
ECR . . . 21
EDR . . . 23
Generate . . . 24
IRMSE . . . 26
MAE . . . 28
MSRSE . . . 30
quiet . . . 32
RAB . . . 33
rbind.SimDesign . . . 34
RD . . . 35
RE . . . 37
rejectionSampling . . . 38
reSummarise . . . 41
rHeadrick . . . 44
rint . . . 46
rinvWishart . . . 48
rmgh . . . 49
RMSE . . . 50
rmvnorm . . . 52
rmvt . . . 53
RSE . . . 55
rtruncate . . . 56
runSimulation . . . 58
rValeMaurelli . . . 72
Serlin2000 . . . 74
SimAnova . . . 75
SimCheck . . . 76
SimClean . . . 77
SimDesign . . . 79
SimExtract . . . 79
SimFunctions . . . 81

SimResults 83
SimShiny 84
Summarise 86

Index **88**

`add_missing` *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

```
add_missing(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 <- add_missing(y), 10)

## 50% missing with default FUN
head(ymiss <- add_missing(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 <- add_missing(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 <- add_missing(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 <- add_missing(y, fun=fun)
tail(cbind(y, ymiss), 10)


## End(Not run)
```

---

aggregate_simulations *Collapse separate simulation files into a single result*

Description

This function 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 piecewise on one node (e.g., 500 replications in one batch, 500 again at a later date) or run independently across different nodes/computers that are not on the same network.

Usage

```
aggregate_simulations(
  files = NULL,
  file_name = "SimDesign_aggregate.rds",
  dirs = NULL,
  results_dirname = "SimDesign_aggregate_results"
)
```

Arguments

`files`	a `character` vector containing the names of the simulation's final `.rds` files
`file_name`	name of .rds file to save aggregate simulation file to. Default is `'SimDesign_aggregate.rds'`
`dirs`	a `character` vector containing the names of the `save_results` directories to be aggregated. A new folder will be created and placed in the `results_dirname` output folder
`results_dirname`	the new directory to place the aggregated results files

Value

if `files` is used the function returns a `data.frame` with the (weighted) average of the simulation results. Otherwise, if `dirs` is used, the function returns NULL

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:

setwd('my_working_directory')

## run simulations to save the .rds files (or move them to the working directory)
# runSimulation(..., filename='file1')
# runSimulation(..., filename='file2')

# saves to the hard-drive and stores in workspace
final <- aggregate_simulations(files = c('file1.rds', 'file2.rds'))
final

# aggregate saved results for .rds files and results directories
# runSimulation(..., save_results = TRUE, save_details = list(save_results_dirname = 'dir1'))
# runSimulation(..., save_results = TRUE, save_details = list(save_results_dirname = 'dir2'))

# place new saved results in 'SimDesign_results/' by default
aggregate_simulations(files = c('file1.rds', 'file2.rds'),
                      dirs = c('dir1', 'dir2'))



## 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 = NULL)
```

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

Examples

```
## Not run:

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

   # 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 = NULL) {
    welch <- t.test(DV ~ group, dat)
    ret <- c(p=welch$p.value)
    ret
}

analysis_ind <- function(condition, dat, fixed_objects = NULL) {
    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 a 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, data = NULL)
```

Arguments

`x`	logical statement to evaluate. If the statement evaluates to `TRUE` then the remainder of the defined function will be evaluated
`data`	(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 = NULL) {
  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 = NULL) {
  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 = NULL) {
  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 = NULL) {
  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)
```

Arguments

`...` a comma separated list of `data.frame` or `tibble` objects containing 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]]`

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

```
## Not run:

# does not use Attach()
Generate <- function(condition, fixed_objects = NULL) {
    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))
    dat
}

# similar to above, but using the Attach() function instead of indexing
Generate <- function(condition, fixed_objects = NULL) {
```

```
    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
}

## End(Not run)
```

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",
  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), `'abs_relative'` computes the relative bias but the absolute values of the parameters are used in the denominator rather than the (potentially) signed input values, and `'standardized'` computes the standardized bias estimate (standard bias divided by the standard deviation of the sample estimates)
- `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')

dev <- samp - pop
bias(dev)

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

# 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))

# 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)
```

`boot_predict` *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

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

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 = NULL) {
    dat <- rnorm(100, 0, condition$sigma)
    dat
}

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

Summarise <- function(condition, results, fixed_objects = NULL) {
    ret <- c(mu_bias = bias(results[,1], 0),
             mu_coverage = ECR(results[,2:3], parameter = 0))
    ret
}

## Not run:
# boot_predict supports only one condition at a time
out <- boot_predict(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)
```

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)
```

---

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, tibble = TRUE, stringsAsFactors = FALSE)

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

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
`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
`x`	object returned by `createDesign`
`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

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

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


## End(Not run)
```

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,
  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

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)

# single vector input
CI <- c(-1, 1)
ECR(CI, 0)
ECR(CI, 2)
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 rate for Type I errors and Power*

Description

Computes the detection 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 nominal detection rate to be studied (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

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)
```

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 = NULL)
```

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

add_missing, Attach, rmgh, rValeMaurelli, rHeadrick

Examples

```
## Not run:

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

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


## 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)
```

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", 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))
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)
```

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

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

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

quiet *Suppress function messages and Concatenate and Print (cat)*

Description

This function is used to suppress information printed from external functions that make internal use of `link{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 `suppressWarnings`, though use this function carefully as some warnings can be meaningful and unexpected.

Usage

```
quiet(..., messages = FALSE, cat = FALSE)
```

Arguments

`...`	the functional expression to be evaluated
`messages`	logical; suppress all messages?
`cat`	logical; suppress all concatenate and print calls from `cat`?

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

```
myfun <- function(x){
   message('This function is rather chatty')
   cat("It even prints in different output forms!\n")
   message('And even at different....')
   cat("...times!\n")
   x
}

out <- myfun(1)
out

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

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 = NULL) {
    dat <- with(condition, rnorm(N, 10, sd=SD))
```

```
    dat
}

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

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

Arguments

summarise	a summarise function to apply to the read-in files. See runSimulation for details
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%)

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 = NULL) {
    dat <- with(condition, rnorm(N, 10, 5)) # distributed N(10, 5)
    dat
}

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

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

Summarise <- function(condition, results, fixed_objects = NULL){
    apply(results, 2, mean)
}

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

Summarise2 <- function(condition, results, fixed_objects = NULL){
    ret <- c(mean_ests=apply(results, 2, mean),
             SE=apply(results, 2, sd))
    ret
}

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

SimClean('simresults/')


## End(Not run)

###
```

```
# similar to above, but using objects defined in workspace
results <- runSimulation(design=Design, replications=50,
                         generate=Generate, analyse=Analyse)
str(results)

Summarise <- function(condition, results, fixed_objects = NULL){
    ret <- c(mean_ests=apply(results, 2, mean),
             SE=apply(results, 2, sd))
    ret
}

res <- reSummarise(Summarise, results=results, Design=Design)
res

res <- reSummarise(Summarise, results=results, boot_method = 'basic')
res

###
# Also similar, but storing the results within the summarised simulation

Summarise <- function(condition, results, fixed_objects = NULL){
    ret <- c(mean_ests=apply(results, 2, mean),
             SE=apply(results, 2, sd))
    ret
}

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

# internal results stored
results <- SimExtract(res, what = 'results')
str(results)

# pass SimDesign object to results
res <- reSummarise(Summarise, results=res, boot_method = 'basic')
res
```

`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)
# psych::describe(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",
  MSE = FALSE,
  percent = FALSE,
  unname = FALSE
)

RMSD(
  estimate,
  parameter = NULL,
  type = "RMSE",
```

```
    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
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])
```

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 <- apply(par_ests, 2, sd))

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)
```

---

runSimulation *Run a Monte Carlo simulation given a data.frame of 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, 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 and their associated `.Random.seed` states. For convenience, all functions available in the R work-space are exported across all computational 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). 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.

Usage

```
runSimulation(
  design,
  replications,
  generate,
  analyse,
  summarise,
  fixed_objects = NULL,
  packages = NULL,
  filename = NULL,
  debug = "none",
  load_seed = NULL,
  save_results = FALSE,
  parallel = FALSE,
  ncores = parallel::detectCores(),
  cl = NULL,
```

```
  notification = "none",
  beep = FALSE,
  sound = 1,
  CI = 0.95,
  seed = NULL,
  boot_method = "none",
  boot_draws = 1000L,
  max_errors = 50L,
  save_seeds = FALSE,
  save = TRUE,
  store_results = FALSE,
  save_details = list(),
  extra_options = list(),
  progress = TRUE,
  verbose = TRUE
)

## 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

`replications` number of independent replications to perform per condition (i.e., each row in `design`). Must be greater than 0

`generate` user-defined data and parameter generating function. 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

`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 analyses the simulated data). See `Analyse` for details

`summarise` optional (but highly 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.

Omitting this function will return a list of `data.frames` (or a single `data.frame`, if only one row in `design` is supplied) or, for more general objects (such as `lists`), a `list` containing the results returned form `Analyse`. Alternatively, the value `NA` can be passed to let the generate-analyse-summarise process to run as usual, where the summarise components are instead included only as a placeholder. Omitting this input is only recommended for didactic purposes because it leaves out a large amount of information (e.g., try-errors, warning messages, saving files, etc), can witness memory related issues, and generally is not as flexible internally.

If users do not wish to supply a summarise function then it is is recommended to pass `NA` to this argument while also supplying passing `save_results = TRUE` to save the results to the hard-drive during the simulation. This provides a more RAM friendly alternative to storing all the Generate-Analyse results in the working environment, where the Analysis results can be summarised at a later time

`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 `parallel = TRUE` or `MPI = TRUE` to use non-standard functions from additional packages, otherwise the functions must be made available by using explicit `library` or `require` calls within the provided simulation functions. 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.

Alternatively, 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., `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`
`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?
`ncores`	number of cores to be used in parallel execution. Default uses all available
`cl`	cluster object defined by `makeCluster` used to run code in parallel. 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 to chain together (note in this case that you must use either the "MPI" or "PSOCK" clusters)
`notification`	an optional character vector input that can be used to send Pushbullet notifications from a configured computer. This reports information such as the total execution time, the condition completed, and error/warning messages recorded. This arguments assumes that users have already A) registered for a Pushbullet account, B) installed the application on their mobile device and computer, and C) created an associated JSON file of the form `~/.rpushbullet.json` using `RPushbullet::pbSetup()`). To utilize the `RPushbullet` in `SimDesign` first call `library(RPushbullet` before running `runSimulation()` to read-in the default JSON file. Next, 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.
`beep`	logical; call the `beepr` package when the simulation is completed?

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

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

seed
: a vector of integers to be used for reproducibility. The length of the vector must be equal the number of rows in `design`. This argument calls `set.seed` or `clusterSetRNGStream` for each condition, respectively, but will not be run when `MPI = TRUE`. Default randomly generates seeds within the range 1 to 2147483647 for each condition.

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

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`). 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

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`, 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

store_results
: logical; store the complete tables of simulation results in the returned object? This is FALSE by default to help avoid RAM issues (see save_results as a more suitable alternative). However, 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 will no longer be an issue.

  To extract these results pass the returned object to SimExtract(..., 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_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
  - 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

extra_options
: a list for extra information flags no commonly used. These can be

  - 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
  - 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

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 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 earlier 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 NAs be allowed in the analyse step as a valid result from the simulation analysis?

allow_nan logical (default is FALSE); should NaNs 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 no 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

MPI logical (default is FALSE); use the foreach package in a form usable by MPI to run simulation in parallel on a cluster?

progress	logical; display a progress bar (using the pbapply package) for each simulation condition? This is useful when simulations conditions take a long time to run (see also the notifications argument). Default is TRUE
verbose	logical; print messages to the R console? Default is TRUE
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

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 = NULL) {...}
Analyse <- function(condition, dat, fixed_objects = NULL) {...}
Summarise <- function(condition, results, fixed_objects = NULL) {...}
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 `r`th replication (e.g., `REPLICATION == 500`) within the `j`th 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, COMPLETED to indicate the date in which the given simulation condition completed, SEED for the integer values in the seed argument, 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 save_seeds option 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, save_seeds is less useful since 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 `MPI = TRUE`) 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 workspace 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 workspace. 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`, `aggregate_simulations`, `Attach`, `AnalyseIf`, `SimShiny`

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 = NULL) {
    dat <- with(condition, rnorm(N, 10, 5)) # distributed N(10, 5)
    dat
}

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

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


#~~~~~~~~~~~~~~~~~~~~~~~~
#### Step 3 --- Collect results by looping over the rows in design

# run the simulation
Final <- runSimulation(design=Design, replications=1000,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
Final

# reproduce exact simulation
Final_rep <- runSimulation(design=Design, replications=1000, seed=Final$SEED,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
Final_rep


#~~~~~~~~~~~~~~~~~~~~~~~~
#### Extras
## Not run:
# 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 of summarise(), or
#   b) pass save_results = TRUE to runSimulation() and read the results in with SimResults()
# Note that the latter method should be adopted for longer simulations

# e.g., the a) approach
res <- runSimulation(design=Design, replications=1000,
                     generate=Generate, analyse=Analyse)
str(res)
head(res[[1]])

# or b) approach
Final <- runSimulation(design=Design, replications=1000, save_results=TRUE,
                       generate=Generate, analyse=Analyse, summarise=Summarise)
```

```
res <- SimResults(Final)
str(res)
head(res[[1]]$results)


# 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 = NULL) {
    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 = NULL) {
    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 = NULL) {
    #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=5,
                     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
runSimulation(design=Design, replications=1000, parallel=TRUE, filename = 'mysim',
              generate=Generate, analyse=Analyse, summarise=Summarise,
              notification = 'complete')
```

```
## 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 = NULL) {
    #find results of interest here (e.g., alpha < .1, .05, .01)
    browser()
    ret <- EDR(results[,nms], alpha = .05)
    ret
}

runSimulation(design=Design, replications=1000,
              generate=Generate, analyse=Analyse, summarise=Summarise,
              parallel=TRUE)



## EXTRA: To run the simulation on a MPI cluster, use the following setup (not run)
# library(doMPI)
# cl <- startMPIcluster()
# registerDoMPI(cl)
# Final <- runSimulation(design=Design, replications=1000, MPI=TRUE,
#                        generate=Generate, analyse=Analyse, summarise=Summarise)
# saveRDS(Final, 'mysim.rds')
# closeCluster(cl)
# mpi.quit()


## Similarly, run simulation on a 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)

#~~~~~~~~~~~~~~~~~~~~~~~~
###### 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')


## 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)
# psych::describe(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)
# psych::describe(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)
```

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 the status of the simulation's temporary results*

Description

This function reads the temporary file saved by runSimulation by collapsing the information into a suitable (albeit temporary) object of class 'SimDesign'. This is useful when taking a quick-peak at how the early simulation results are performing (useful long running simulation results with many rows in the Design object). Returns a tibble-based data.frame object (tbl_df).

Usage

```
SimCheck(file)
```

Arguments

file	the temp file currently saving the simulation state. If missing the file is assumed to be in the current working directory, and start with the name 'SIMDESIGN-TEMPFILE'

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:

# explicit
temp_results <- SimCheck(file = 'SIMDESIGN-TEMPFILE_mycomp.rds')
temp_results

# works if file is in the current working directory
temp_results <- SimCheck()
temp_results


## 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)
```

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 and MPI distributed computations, 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`. 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)
```

Arguments

- `object` — object returned from `runSimulation`
- `what` — character indicating what information to extract. 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, `'error_seeds'` and `'warning_seeds'` to extract the associated `.Random.seed` values associated with the ERROR/WARNING messages, `'results'` to extract the simulation results if the option `store_results` was passed to `runSimulation`, and `'summarise'` if the `Summarise` definition returned a named `list` rather than a named numeric vector.

  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

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 = NULL) {
    int <- sample(1:10, 1)
    if(int > 5) warning('GENERATE WARNING: int greater than 5')
    if(int == 1) stop('GENERATE WARNING: integer is 1')
    rnorm(5)
}

Analyse <- function(condition, dat, fixed_objects = NULL) {
    int <- sample(1:10, 1)
    if(int > 5) warning('ANALYSE WARNING: int greater than 5')
    if(int == 1) stop('ANALYSE WARNING: int is 1')
    c(ret = 1)
}
```

```
Summarise <- function(condition, results, fixed_objects = NULL) {
    mean(results)
}

res <- runSimulation(replications = 100, seed=1234, verbose=FALSE,
                     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(),
  comments = FALSE,
  singlefile = TRUE,
  summarise = TRUE,
  generate = TRUE,
  nAnalyses = 1,
  openFiles = TRUE
)
```

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
comments	logical; include helpful comments? Default is FALSE
singlefile	logical; when filename is included, put output in one files? When FALSE the output is saved to two separate files containing the functions and design definitions. The two-file format often makes organization and debugging slightly easier, especially for larger Monte Carlo simulations. Default is TRUE
summarise	include summarise function? Default is TRUE
generate	include generate function? Default is TRUE
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
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)?

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 passing singlefile = FALSE), 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)


## 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(results, which, wd = getwd())
```

Arguments

results	object returned from runSimulation where save_results = TRUE was used
which	a numeric vector indicating which rows should be read in. If missing, all rows will be read in
wd	working directory; default is found with getwd.

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

Examples

```
## Not run:

results <- runSimulation(..., save_results = TRUE)

# row 1 results
row1 <- SimResults(results, 1)

# rows 1:5, stored in a named list
rows_1to5 <- SimResults(results, 1:5)

# all results
rows_all <- SimResults(results)


## 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 = NULL) {
    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 = NULL) {
    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 = NULL) {
    #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, verbose=FALSE)

# save shiny code to file
SimShiny('app.R', design=Design, generate=Generate, analyse=Analyse,
         summarise=Summarise, verbose=FALSE)

# run the application
shiny::runApp()
shiny::runApp(launch.browser = TRUE) # in web-browser


## 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 = NULL)
```

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 = NULL) {

    #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)
```

Index

∗ **data**
BF_sim, 12
BF_sim_alternative, 13
∗ **package**
SimDesign, 79

abs, *14*
add_missing, 3, *25*
aggregate_simulations, 5, *67*
Analyse, 6, *9*, *24*, *59*, *61*, *65*, *67*, *87*
AnalyseIf, *8*, 8, *67*
aov, *75*
as.vector, *36*
Attach, 10, *25*, *67*
attach, *10*

BF_sim, 12, *13*
BF_sim_alternative, *12*, 13
bias, 13, *33*, *51*, *87*
boot_predict, 16
browser, *60*

cat, *32*
CC, 18
clusterSetRNGStream, *62*
cor, *19*
createDesign, 19, *20*, *59*, *65*, *67*

ECR, 21, *24*, *75*, *87*
EDR, *22*, 23, *75*, *87*
expand.grid, *20*

Generate, *7*, *11*, 24, *59*, *65*, *67*
getwd, *83*

integrate, *27*
IRMSE, 26

library, *60*
lm, *17*, *75*

MAE, 28, *87*
makeCluster, *61*
MSRSE, 30

optimize, *39*

print.Design *(*createDesign*)*, 19
print.SimDesign *(*runSimulation*)*, 58

quiet, 32

RAB, 33
rbind.SimDesign, 34
RD, 35
RE, 37, *87*
readRDS, *83*
rejectionSampling, 38
require, *60*
reSummarise, 41, *61*, *67*
rHeadrick, *24, 25*, 44
rint, 46
rinvWishart, 48
rmgh, *24, 25*, 49
RMSD *(*RMSE*)*, 50
RMSE, *15*, *27*, *37*, 50, *87*
rmvnorm, 52
rmvt, 53
RSE, 55
rtruncate, 56
runSimulation, *5–7*, *9*, *11*, *16*, *20*, *24*, *34*, *42*, *48*, *53, 54*, *57*, 58, *64*, *75–83*, *85*, *87*
rValeMaurelli, *24, 25*, 72

sample, *46*
sample.int, *46*
saveRDS, *66*, *78*
Serlin2000, 74
set.seed, *62*
SimAnova, *65*, *67*, 75
SimCheck, 76
SimClean, *67*, 77

SimDesign, 79
SimExtract, *61*, *62*, *66*, *67*, 79, *87*
SimFunctions, *65*, *67*, 81
SimResults, *61*, *67*, 83
SimShiny, *67*, 84
stop, *7*, *8*, *25*
subset, *75*
Summarise, *55*, *59*, *65*, *67*, *80*, 86
summary.SimDesign (runSimulation), 58
suppressWarnings, *32*

try, *7*, *25*

unname, *14*, *18*, *22*, *23*, *29*, *30*, *33*, *36*, *37*, *51*, *55*

Vectorize, *38*, *39*