Package ‘SimDesign’

July 17, 2018

Title Structure for Organizing Monte Carlo Simulation Designs

Version 1.11

Description Provides tools to help safely and efficiently 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. For a pedagogical introduction to the package refer to Sigal and Chalmers (2016) <doi:10.1080/10691898.2016.1246953>.

VignetteBuilder knitr

Depends R (>= 3.2.1)

Imports foreach, methods, parallel,plyr, pbapply (>= 1.3-0), stats

Suggests knitr, dplyr, ggplot2, shiny, doMPI, copula, extraDistr

License GPL (>= 2)

ByteCompile yes

LazyData true

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

RoxygenNote 6.0.1

NeedsCompilation no

Author Phil Chalmers [aut, cre],
      Matthew Sigal [ctb],
      Ogreden Oguzhan [ctb]

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

Repository CRAN

Date/Publication 2018-07-17 04:30:03 UTC
R topics documented:

add_missing .................................................. 3
aggregate_simulations ..................................... 5
Analyse .......................................................... 6
Attach ............................................................. 8
BF_sim ............................................................ 9
BF_sim_alternative ......................................... 10
bias ............................................................... 11
boot_predict .................................................... 12
ECR .............................................................. 14
EDR .............................................................. 16
Generate .......................................................... 17
IRMSE ............................................................ 19
MAE .............................................................. 21
MSRSE ........................................................... 22
quiet .............................................................. 24
RD ................................................................. 25
RE ................................................................. 26
rejectionSampling .......................................... 27
rHeadrick ...................................................... 29
rint ............................................................... 31
rinvWishart ..................................................... 32
rmgh .............................................................. 33
RMSE ............................................................ 35
rmvnorm .......................................................... 36
rmvt ............................................................... 37
rtruncate ........................................................ 39
runSimulation .................................................. 40
rValeMaurelli .................................................. 53
Serlin2000 ...................................................... 54
SimAnova ....................................................... 55
SimBoot .......................................................... 57
SimClean ........................................................ 58
SimDesign ....................................................... 59
SimFunctions ................................................... 60
SimResults ...................................................... 61
SimShiny ........................................................ 63
subset.SimDesign ........................................... 64
Summarise ....................................................... 66

Index ............................................................ 68
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 NAs). 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


Examples

```r
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)
  pllogis(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)
```
aggregate_simulations  

Collapse separate simulation files into a single result

Description

This function grabs all .rds files in the working directory and aggregates them into a single data.frame object or combines all the saved results directories and combines them into one. This is generally useful when results are run piecewise on one node or run independently across different nodes/computers which are not on the same network.

Usage

aggregate_simulations(files = NULL, dirs = NULL, results_dirname = "SimDesign_aggregate_results")

Arguments

files  
a character vector containing the names of the simulation files. If NULL, all files in the working directory ending in .rds will be used

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


See Also

runSimulation
Examples

```r
## 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')

final <- aggregate_simulations()
saveRDS(final, 'my_final_simulation.rds')

# aggregate saved results
# 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/' directory by default
aggregate_simulations(dirs = c('dir1', 'dir2'))

## End(Not run)
```

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. If a suitable `generate` function was not supplied then this function can be used to generate and analyse the Monte Carlo data (though in general this setup is not recommended for larger simulations).

Usage

```r
Analyze(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


See Also

stop

Examples

## Not run:

myanalyse <- 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)
}

## End(Not run)
Description

This function accepts the condition object used to indicate the design conditions and makes the variable names available in the environment from which it is called. 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(condition, check = TRUE)

Arguments

condition a data.frame containing the condition names
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

Details

The behavior 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.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References


See Also

runSimulation, Generate

Examples

## Not run:

# does not use Attach()
mygenerate <- function(condition, fixed_objects = NULL){
  N1 <- condition$sample_sizes_group
Example simulation from Brown and Forsythe (1974)

**BF_sim**

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 on the wiki ([https://github.com/philchalmers/SimDesign/wiki](https://github.com/philchalmers/SimDesign/wiki)).

**Author(s)**

Phil Chalmers <philip.chalmers@gmail.com>

**References**


BF_sim_alternative

Examples

```r
## 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](https://github.com/philchalmers/SimDesign/wiki)).

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References


Examples

```r
## 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")

Arguments
- **estimate**: a numeric vector or matrix/data.frame 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.
- **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), and 'standardized' computes the standardized bias estimate (standard bias divided by the standard deviation of the sample estimates).

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

Author(s)
Phil Chalmers <rphilip.chalmers@gmail.com>

References

See Also
RMSE
Examples

```r
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)
```

**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

```r
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


Examples

```r
set.seed(4321)
Design <- expand.grid(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
}

# 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

---

**ECR**

*Compute the empirical coverage rate for Type I errors and Power*

---

**Description**

Computes the detection rate for determining empirical Type I error and power rates using information from the 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)

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. Alternatively, 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 \(1 - \text{sum}(\text{ECR(CIs, parameter, tails=TRUE)})\)

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References


See Also

EDR

Examples

```r
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)

**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.
Generate

Author(s)
Phil Chalmers <rphilip.chalmers@gmail.com>

References

See Also
ECR

Examples

```r
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)
```

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
Value

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

References


See Also

add_missing,Attach,rmgh.rValeMaurelli.rHeadrick

Examples

```r
## Not run:

mygenerate <- 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
mygenerate <- 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
}

mygenerate2 <- function(condition, fixed_objects = NULL){
  mu <- sample(c(-1,0,1), 1)
  dat <- rnorm(100, mu)
  dat    #return simple vector (discard mu information)
}
```
mygenerate3 <- 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**

```r
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

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

where \( g(\theta, \ldots) \) 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 <philip.chalmers@gmail.com>

References


See Also

RMSE

Examples

# logistic regression function with one slope and intercept

# 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, ...)


MAE

Computes 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")

Arguments

- **estimate**: a numeric vector or matrix/data.frame 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.
- **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 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 'NMSE_sd' for the normalized MAE by the standard deviation (MAE / sd(estimate)).

Value

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References


See Also

RMSE
Examples

```r
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 = 'NMAE_SD')

# 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**

```r
MSRSE(SE, SD)
```

**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
Details

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

\[
MSRSE = \frac{E(\text{SE}(\psi)^2)}{SD(\psi)^2} = \frac{1/R \ast \sum_{r=1}^{R} \text{SE}(\psi_r)^2}{SD(\psi)^2} - 1
\]

where \(\text{SE}(\psi_r)\) represents the estimate of the standard error at the \(r\)th 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 \(\psi\).

Value

returns a vector of ratios indicating the relative performance of the standard error estimates to the observed parameter standard deviation. Values less than 0 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 0 were smaller than the standard deviation of the parameters (i.e., more liberal SEs)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References


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)
Description

This function is used to suppress information printed from external functions that make internal use of `message` and `cat`, which provide information in interactive R sessions. For simulations, the session is not interactive, and therefore this type of output should be suppressed. For similar behavior for suppressing warning messages see `suppressWarnings`, though use this function carefully as some warnings can be meaningful and unexpected.

Usage

```r
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


Examples

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

out <- myfun(1)
out

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

Description
Computes the relative difference statistic of the form \((\text{est} - \text{pop})/\text{pop}\), which is equivalent to the form \(\text{est}/\text{pop} - 1\). If matrices are supplied then an equivalent matrix variant will be used of the form \((\text{est} - \text{pop}) * \text{solve}(\text{pop})\). Values closer to 0 indicate better relative parameter recovery.

Usage
RD(est, pop, as.vector = TRUE)

Arguments
- est: a numeric vector or matrix containing the parameter estimates
- pop: a numeric vector or matrix containing the true parameter values. Must be of the same dimensions as est
- as.vector: logical; always wrap the result in a as.vector function before returning?

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

Examples
```r
# 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)
```
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)`

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?

Value

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

Author(s)

Phil Chalmers <philip.chalmers@gmail.com>

References


Examples

```r
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))

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

This function supports the rejection sampling (i.e., accept-reject) approach to drawing values from seemingly difficult, and potentially non-normed, probability density functions by sampling values from a more manageable proxy distribution. This function is optimized to work efficiently when the defined functions are vectorized; otherwise, the accept-reject algorithm will loop over candidate sample-draws in isolation.

Usage

```r
cleanup::cleancache()

rejectionSampling(n, df, dg, rg, M = NULL, returnM = FALSE, vectorized = TRUE)
```

Arguments

- `n`: number of samples to draw
- `df`: the desired (potentially un-normed) probability density function to draw samples from. Must be in the form of a function with a single input corresponding to the values sampled from `rg`
- `dg`: the proxy (potentially un-normed) probability 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`
- `rg`: the proxy random number generation function, associated with `dg`, used to draw samples from in lieu of drawing samples from `df`. 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)
- `M`: the upper-bound of the ratio of probability density functions to help minimize the number of discarded draws. By default, `M` is computed internally by finding the maximum of the ratio `df(x) / dg(x)` within the range `[0, 1]`. When both `df` and `dg` are true probability density functions (i.e., integrate to 1) then the acceptance probability is equal to `1/M`
- `returnM`: logical; return the value of `M` located from the internal optimization results?
- `vectorized`: logical; have the input function been vectorized (i.e., do they support a vector of input values rather than only a single sample)? This can be disabled, however it’s recommended to redefine the input functions to be vectorized instead since these are more efficient when `n` is large or `1/M` is small
Details

The accept-reject algorithm is a flexible approach to obtaining i.i.d.'s from a difficult to sample from probability density function (pdf) where either the transformation method fails or inverse of the cumulative distribution function is too 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), and accepts the draws if they are likely within the proposed pdf. 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 <philip.chalmers@gmail.com>

References


Examples

```r
# 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)
df <- function(x) dbeta(x, shape1 = 2.7, shape2 = 6.3)
dg <- function(x) dunif(x, min = 0, max = 1)
rg <- function(n) runif(n, min = 0, max = 1)

dat <- rejectionSampling(10000, df=df, dg=dg, rg=rg)
hist(dat, 100)
hist(rbeta(10000, 2.7, 6.3), 100) # compare

# when df and dg both integrate to 1, acceptance probability = 1/M
rejectionSampling(df=df, dg=dg, rg=rg, returnM=TRUE)

# user supplied M. Here, M = 4, indicating 25% acceptance rate
dat2 <- rejectionSampling(10000, df=df, dg=dg, rg=rg, M=4)
hist(dat2, 100)

# generate using better support function (here, Y ~ beta(2,6))
dg <- function(x) dbeta(x, shape1 = 2, shape2 = 6)
rg <- function(n) rbeta(n, shape1 = 2, shape2 = 6)
rejectionSampling(10000, df=df, dg=dg, rg=rg, returnM=TRUE) # more efficient
dat <- rejectionSampling(10000, df=df, dg=dg, rg=rg)
hist(dat, 100)
```

---------------------------------------------
```r
# sample from wonky (and non-normed) pdf, like below
df <- 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, df(y), type = 'l', main = "pdf to sample")

# choose dg/rg functions that have support within the range [-inf, inf]
rg <- function(n) rnorm(n, sd=2)
dg <- function(x) dnorm(x, sd=2)
dat <- rejectionSampling(10000, df=df, dg=dg, rg=rg)
hist(dat, 100, prob=TRUE)
lines(density(dat), col = 'red')

# same as above, but df not vectorized (much slower)
df2 <- function(x){
  ret <- if(x <= .5) dnorm(x)
  else if(x > .5) dnorm(x) + dchisq(x, df = 2)
  ret
}
system.time(dat2 <-
  rejectionSampling(100000, df=df2, dg=dg, rg=rg, vectorized=FALSE))
system.time(dat <-
  rejectionSampling(100000, df=df, dg=dg, rg=rg))

#---------------------------------------------------------------
# multivariate distribution
df <- function(x) prod(c(dnorm(x[1]) + dchisq(x[1], df = 5),
  dnorm(x[2], -1, 2)))
rg <- function(n) c(rnorm(n, sd=3), rnorm(n, sd=3))
dg <- function(x) prod(c(dnorm(x[1], sd=3), dnorm(x[1], sd=3)))

dat <- rejectionSampling(5000, df=df, dg=dg, rg=rg, M=10)
hist(dat[,1], 30)
hist(dat[,2], 30)
plot(dat)
```

---

**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

```r
code: rHeadrick

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(seed =
    NULL, 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 <rphil.chalmers@gmail.com>

References


Examples

```r
## 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**

```r
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

Examples

```r
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

da 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


See Also

runSimulation

Examples

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

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

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


Examples

```r
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, 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**

```r
RMSE(estimate, parameter = NULL, type = "RMSE", MSE = FALSE)
```

**Arguments**

- **estimate**
  - a numeric vector or matrix/data.frame 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.

- **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 \( \sqrt{\text{mean}(\text{estimate}^2)} \) 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))), ‘NRMSE_SD’ for the normalized RMSE with the standard deviation (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

**Value**

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

**Author(s)**

Phil Chalmers <rphilip.chalmers@gmail.com>

**References**

See Also

bias
MAE

Examples

```r
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 = 'NRMSE_SD')
RMSE(samp, pop, type = 'CV')
RMSE(samp, pop, type = 'RMSLE')

# 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

```r
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


See Also

`runSimulation`

Examples

```r
# 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 noncentrality 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


See Also

`runSimulation`

Examples

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

```
rtruncateln, rfunL, rangeL, \ldots, 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 required for this function are passed through the argument \ldots
- `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
- \ldots 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 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
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 tracking of error and warning messages. For convenience, all functions available
in the R workspace 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 a didactic presentation of the package refer to Sigal and Chalmers (2016; doi: 10.1080/10691898.2016.1246953), and 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.

Usage

```runSimulation(design, replications, generate, analyse, summarise,
  fixed_objects = NULL, packages = NULL, bootSE = FALSE,
  boot_draws = 1000L, filename = "SimDesign-results",
  seed = rint(nrow(design), min = 1L, max = 2147483647L), save = FALSE,
  save_results = FALSE, store_results = FALSE, warnings_as_errors = FALSE,
  save_seeds = FALSE, load_seed = NULL, parallel = FALSE,
  ncores = parallel::detectCores(), cl = NULL, MPI = FALSE,
  max_errors = 50L, as.factor = TRUE, save_generate_data = FALSE,
  save_details = list(), edit = "none", progress = TRUE, verbose = TRUE)
```

```r
# S3 method for class 'SimDesign'
print(x, drop.extras = FALSE, drop.design = FALSE,
  format.time = TRUE, ...)
```

```r
# S3 method for class 'SimDesign'
head(x, ...)
```

```r
# S3 method for class 'SimDesign'
tail(x, ...)
```

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

```r
extract_results(object)
```

```r
extract_error_seeds(object)
```

```r
# S3 method for class 'SimDesign'
as.data.frame(x, ...)
```

Arguments

design a 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

replications number of replication 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 computation function which acts on the data generated from `Generate` (or, if `generate` was omitted, both generates and analyses the simulated data). See `Analyse` for details.

- **summarise**: optional (but highly recommended) user-defined summary function to be used after all the replications have completed within each design condition. Omitting this function will return a list of matrices (or a single matrix, if only one row in design is supplied) or, for more general objects (such as lists), a list containing the results returned from `Analyse`. Omitting this function is only recommended for didactic purposes because it leaves out a large amount of information (e.g., try-errors, warning messages, etc), can witness memory related issues, and generally is not as flexible internally. See the `save_results` option for a more RAM friendly alternative to storing all the Generate-Analyse results in the working environment.

- **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 long fixed vectors/matrices of population parameters, data that should be used across all conditions and replications (e.g., including a fixed design matrix for linear regression), or simply control constant global elements such as 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()`).

- **bootSE**: logical; perform a non-parametric bootstrap to compute bootstrap standard error estimates for the respective meta-statistics computed by the `summarise` function? When `TRUE`, bootstrap samples for each row in `Design` will be obtained after the `generate-analyse` steps have obtain the simulation results to be summarised so that standard errors for each statistic can be computed. To compute large-sample confidence intervals given the bootstrap SE estimates see `SimBoot`. This option is useful to approximate how accurate the resulting meta-statistic estimates were, particularly if the number of replications was relatively low (e.g., less than 5000). If users prefer to obtain alternative bootstrap estimates then consider passing `save_results = TRUE`, reading the generate-analyse data into R via `SimResults`, and performing the bootstrap manually with function found in the external `boot` package.

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

- **filename**: (optional) the name of the `.rds` file to save the final simulation results to when `save = TRUE`. 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 `'SimDesign-results'`.
**seed**

A vector of integers to be used for reproducibility. The length of the vector must be equal to 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.

**save**

Logical; save the simulation state and final results to the hard-drive? This is useful for simulations which require an extended amount of time. 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). To recover your simulation at the last known location simply re-run the code you used to initially define the simulation and the external file will automatically be detected and read-in. Upon completion, the final results will be saved to the working directory, and the temp file will be removed. Default is `FALSE`.

**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 analyses. 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. When `TRUE`, a temp file will be used to track the simulation state (in case of power outages, crashes, etc). When `TRUE`, temporary files will also be saved to the working directory (in the same way as when `save = TRUE`). See `SimResults` for an example of how to read these `.rds` files back into R after the simulation is complete. Default is `FALSE`.

**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). To extract these results pass the returned object to `extract_results`, which will return a named list of all the simulation results for each condition.

**warnings_as_errors**

Logical; treat warning messages as errors during the simulation? Default is `FALSE`, therefore warnings are only collected and not used to restart the data generation step.

**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 the simulation state within each replication and design condition. Primarily, this is useful for completely replicating any cell in the simulation if need be, especially when tracking down hard-to-find errors and bugs. 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`.

**load_seed**

A character object indicating which file to load from when the `.Random.seed` have been 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 626 long elements from `.Random.seed` (see `extract_error_seed()` 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 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.

**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 stop the cluster when the simulation is complete. Note that supplying a `cl` object will automatically set the parallel argument to TRUE.

**MPI** logical; use the `foreach` package in a form usable by MPI to run simulation in parallel on a cluster? Default is FALSE.

**max_errors** the simulation will terminate when more than this number of consecutive errors are thrown in any given condition. The purpose of this is to indicate that something fatally problematic is likely going wrong in the generate-analyse phases and should be inspected. Default is 50.

**as.factor** logical; coerce the input design elements into factors when the simulation is complete? If the columns inputs are numeric then these will be treated as ordered. Default is TRUE.

**save_generate_data** logical; save the data returned from `Generate` to external `.rds` files located in the defined `save_generate_data_dirname` directory/folder? When TRUE, temporary files will also be saved to the working directory (in the same way as when `save = TRUE`). Default is FALSE.

WARNING: saving data 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. It is generally recommended to leave this argument as FALSE because saving datasets will often consume a needless amount of disk space, and by-and-large saving data is not required for simulations.

**save_details** a list pertaining to information regarding how and where files should be saved when the save, save_results, or save_generate_data 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.

tmpfilename the name of the temporary .rds file when any of the save flags are used. This file will be read-in if it is in the working directory and the simulation will continue at the last point this file was saved (useful in case of power outages or broken nodes). Finally, this file will be deleted when the simulation is complete. Default is the system name (compname) appended to 'SIMDESIGN-TEMPFILE_'.

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.

save_generate_data_dirname a string indicating the name of the folder to save data objects to when save_generate_data = TRUE. If a directory/folder does not exist in the current working directory then one will be created automatically. Within this folder nested directories will be created associated with each row in design. Default is 'SimDesign-generate-data_' with the compname appended.

edit a string indicating where to initiate a browser() call for editing and debugging. General options are 'none' (default) and 'all', which are used to disable debugging and to debug all the user defined functions, respectively. Specific options include: 'generate' to edit the data simulation function, 'analyse' to edit the computational function, and 'summarise' to edit the aggregation function.

Alternatively, users may place browser calls within the respective functions for debugging at specific lines (note: parallel computation flags will automatically be disabled when a browser() is detected).

progress logical; display a progress bar for each simulation condition? This is useful when simulations conditions take a long time to run. Uses the pbapply package to display the progress. Default is FALSE.

verbose logical; print messages to the R console? Default is TRUE.

x SimDesign object returned from runSimulation.

drop.extras logical; don’t print information about warnings, errors, simulation time, and replications? Default is FALSE.

drop.design logical; don’t include information about the (potentially factorized) simulation design? This may be useful if you wish to cbind() the original design data.frame to the simulation results instead of using the auto-factorized version. Default is FALSE.

format.time logical; format SIM_TIME into a day/hour/min/sec character vector? Default is TRUE.
... additional arguments
object SimDesign object returned from runSimulation

Details

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

1) Define a suitable design data.frame object containing fixed conditional information about the Monte Carlo simulations. This is often expedited by using the expand.grid function, and if necessary using the subset function 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). Note that these functions can be automatically generated by using the SimFunctions function.

3) Pass the above objects to the runSimulation function, and declare the number of replications to perform with the replications input. This function will accept a design data.frame object and will return a suitable data.frame object with the simulation results

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

More succinctly, the functions to be called follow the following form with the exact inputs required by the SimDesign package

Design <- expand.grid(...)  
Generate <- function(condition, fixed_objects = NULL) {...}  
Analyse <- function(condition, dat, fixed_objects = NULL) {...}  
Summarise <- function(condition, results, fixed_objects = NULL) {...}  
results <- runSimulation(design=Design, replications, generate=Generate, analyse=Analyse, summarise=Summarise)

The condition object 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. Mainly, these are included to help with debugging, where users can easily locate the rth replication (e.g., REPLICATION == 500) within the jth row in the simulation design (e.g., ID == 2). The REPLICATION input is also useful when temporarily saving files to the hard-drive when calling external command line utilities.

For a skeleton version of the work-flow, which is often useful when initially defining a simulation, see SimFunctions. This function will write template simulation code to one/two files so that modifying the required functions and objects can begin immediately with minimal error. This means that you can focus on your Monte Carlo simulation immediately rather than worrying about the administrative code-work required to organize the simulation work-flow.

Additional information for each condition are also contained in the data.frame object returned by runSimulation: 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, columns containing the number of replications which had to be re-run due to errors (where the error messages represent the names of the columns prefixed with a ERROR: string), and columns containing the number of warnings prefixed with a WARNING: string. Finally, if bootSE = TRUE was included then the final right-most columns will contain the labels BOOT_SE. followed by the name of the associated meta-statistic defined in summarise()

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

Value

a data.frame (also of class 'SimDesign') with the original design conditions in the left-most columns, simulation results and ERROR/WARNING’s (if applicable) in the middle columns, and additional information (such as REPLICATIONS, SIM_TIME, COMPLETED, and SEED) in the right-most columns.

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 save = TRUE to temporarily save the simulation state, and to use the save_results flag to write the analysis results to the hard-disc.

The generated data can be saved by passing save_generate_data = TRUE, however it is often more memory efficient to use the save_seeds option instead to only save R’s .Random.seed state instead (still allowing for complete reproducibility); individual .Random.seed terms may also be read in with the load_seed input to reproduce the exact simulation state at any given replication. Finally, providing a vector of seeds is also possible to ensure that each simulation condition is completely reproducible under the single/multi-core method selected.

The load_seed input will also accept an integer vector corresponding to the exact .Random.seed state. This is helpful because SimDesign also tracks these seeds for simulation conditions that threw errors, where these values can be extracted via the extract_error_seeds() function. The column names indicate the respective design row (first number), the order in which the errors were thrown (second number), and finally the error message string (coerced to a proper data.frame column name). After this data.frame object is extracted, individual columns can be passed to load_seed to replicate the exact error issue that appeared (note that the design object must be indexed manually to ensure that the correct design conditions is paired with this exact .Random.seed state).

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 save and filename arguments provided (for reasons that are more obvious in the parallel computation descriptions below). Using the filename argument (along with save = TRUE) supplied is much safer than using something like 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 safety is prevalent in many aspects of the package and helps to avoid many unrecoverable (yet surprisingly common) mistakes.
Resuming temporary results

In the event of a computer crash, power outage, etc, if save = TRUE was used 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.

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


See Also

Generate, Analyse, Summarise, SimFunctions, SimClean, SimAnova, SimResults, SimBoot, aggregate_simulations, Attach, SimShiny

Examples

```r
# 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 <- data.frame(N = c(10, 20, 30))
```
runSimulation

```r
#------------------------
### 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

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

# or b) approach
Final <- runSimulation(design=Design, replications=1000, save_results=TRUE,
```

---

The code defines functions for generating, analysing, and summarising data, and then uses these functions to simulate data and store results.
runSimulation

```r
generate=Generate, analyse=Analyse, summarise=Summarise)

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

# remove the saved results from the hard-drive if you no longer want them
SimClean(results = TRUE)

# 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 <- expand.grid(sample_size = c(30, 60, 90, 120),
                        group.size_ratio = c(1, 4, 8),
                        standard.deviation_ratio = c(.5, 1, 2))
dim(Design)
head(Design)

#### Step 2 --- Define generate, analyse, and summarise functions

Generate <- function(condition, fixed.objects = NULL){
  N <- condition$sample.size       # alternatively, could use Attach() to make objects available
grs <- condition$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){
```
runSimulation

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$p.value, independent = ind$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
Final <- runSimulation(design=Design, replications=5, store_results=TRUE,
generate=Generate, analyse=Analyse, summarise=Summarise)
head(Final)

## Not run:
# complete run with 1000 replications per condition
Final <- runSimulation(design=Design, replications=1000, parallel=TRUE,
generate=Generate, analyse=Analyse, summarise=Summarise)
head(Final, digits = 3)
View(Final)

## save final results to a file upon completion (not run)
runSimulation(design=Design, replications=1000, parallel=TRUE, save=TRUE, filename = 'mysim',
genarate=Generate, analyse=Analyse, summarise=Summarise)

## 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,
genarate=Generate, analyse=Analyse, summarise=Summarise,
parallel=TRUE, edit='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)
  ret <- EDR(results[,nms], alpha = .05)
  browser()
  ret
}
runSimulation <- function(design, replications = 1000, parallel = TRUE) {
  generate = Generate, analyse = Analyse, summarise = Summarise,
  ...
}

## EXTRA: To run the simulation on a MPI cluster, use the following setup on each node (not run)
# library(doMPI)
# cl <- startMPIcluster()
# registerDoMPI(cl)
# Final <- runSimulation(design=Design, replications=1000, MPI=TRUE, save=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))
# )
# cl <- parallel::makeCluster(type='PSOCK', master=primary, spec=spec)
# Final <- runSimulation(design=Design, replications=1000, parallel = TRUE, save=TRUE,
#   generate=Generate, analyse=Analyse, summarise=Summarise, cl=cl)

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

library(dplyr)
Final2 <- tbl_df(Final)
Final2 %>% summarise(mean(welch), mean(independent))

# quick ANOVA analysis method with all two-way interactions
SimAnova(~ (sample_size + group_size_ratio + standard_deviation_ratio)^2, Final)

# or more specific ANOVAs
SimAnova(independent ~ (group_size_ratio + standard_deviation_ratio)^2, Final)
# make some plots
library(ggplot2)
library(reshape2)

welch_ind <- Final[,c('group_size_ratio', "standard_deviation_ratio", 
    "welch", "independent")]
dd <- melt(welch_ind, id.vars = names(welch_ind)[1:2])

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(~variable)

## 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**

```r
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


Examples

```r
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,r13,1,r23,r13,1,r23,1),3,3)
sk <- c(1.5,1.5,0.5)
ku <- c(3.75,3.75,0.5)
nonnormal <- rValeMaurelli(n, sigma=cor, skew=sk, kurt=ku)
# cor(nonnormal)
# psych::describe(nonnormal)
```

---


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

```r
Serlin2000(p, alpha, delta, R, CI = 0.95)
```
**SimAnova**

**Function for decomposing the simulation into ANOVA-based effect sizes**

**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**


**Examples**

```r
# Cochran's criteria at alpha = .05 (i.e., 0.5 +/- .01), assuming N = 2000
Serlin2000(p = .05, alpha = .05, delta = .01, R = 2000)

# Bradley's liberal criteria given p = .06 and .075, assuming N = 1000
Serlin2000(p = .06, alpha = .05, delta = .025, R = 1000)
Serlin2000(p = .075, 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)
```
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

```r
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 side 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


Examples

```r
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)
```
SimBoot

Function to present bootstrap standard errors estimates for Monte Carlo simulation meta-statistics

Description

This function generates bootstrap confidence intervals for the meta-statistics called within the `summarise` function with `runSimulation` that included the argument `bootSE = TRUE`.

Usage

```r
SimBoot(results, CI = 0.99)
```

Arguments

- `results` object returned from `runSimulation` where `bootSE = TRUE` was used
- `CI` desired confidence interval level for each meta-statistic using the bootstrap SE estimate. Default is .99, which constructs a 99% confidence interval

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References


Examples

```r
## Not run:
#SimFunctions()

Design <- data.frame(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){
  CIs <- t.test(dat)$conf.int # t-based CIs
  xbar <- mean(dat) # mean of the sample data vector
  ret <- c(mean=xbar, lowerCI=CIs[1], upperCI=CIs[2])
  ret
}

Summarise <- function(condition, results, fixed_objects = NULL){
```
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, generate_data = FALSE, results = FALSE,
         seeds = FALSE, temp = 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
- `generate_data`: logical; remove the .rds data-set files saved when passing `save_generate_data = 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`?
- `temp`: logical; remove the temporary file saved when passing `save = TRUE`?
- `save_details`: a list pertaining to information about how and where files were saved (see the corresponding list in `runSimulation`)
**SimDesign**

**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 a didactic presentation of the package refer to Sigal and Chalmers (2016; doi: 10.1080/10691898.2016.1246953), and 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.
SimFunctions

Author(s)
Phil Chalmers <rphilip.chalmers@gmail.com>

References

SimFunctions

**Skeleton functions for simulations**

Description
This function prints skeleton versions of the required SimDesign functions to run simulations, 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. The recommended approach is to save the template to the hard-drive by passing a suitable file name. However, for larger simulations, as well as when using the RStudio, two separate files will often be easier for debugging/sourcing the simulation code (achieved by passing singlefile = FALSE). For a didactic presentation of the package refer to Sigal and Chalmers (2016; doi: 10.1080/10691898.2016.1246953), and see the associated wiki on Github ([https://github.com/philchalmers/SimDesign/wiki](https://github.com/philchalmers/SimDesign/wiki)) for other tutorial material, examples, and applications of SimDesign to real-world simulations.

Usage
SimFunctions(filename = NULL, dir = getwd(), comments = FALSE, singlefile = TRUE, summarise = TRUE, generate = 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

Author(s)
Phil Chalmers <rphilip.chalmers@gmail.com>
SimResults

References


Examples

SimFunctions()
SimFunctions(comments = TRUE) #with helpful comments

## Not run:

# write output to two files (recommended for larger MCSs)
SimFunctions('mysim', singlefile = FALSE)

# write output files to a single file with comments
SimFunctions('mysim', comments = TRUE)

## End(Not run)

---

SimResults  

**Function to read in saved simulation results**

Description

If `runSimulation` was passed the flag `save_results = TRUE` then the row results corresponding to the design object will be stored to a suitable sub-directory as individual `.rds` files. While users could use `readRDS` directly to read these files in themselves, this convenience function will read the desired rows in automatically given the returned object from the simulation. Can be used to read in 1 or more `.rds` files at once (if more than 1 file is read in then the result will be stored in a list).

Usage

`SimResults(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`. 
**SimResults**

**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**


**Examples**

```r
## 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


See Also

runSimulation

Examples

## Not run:

Design <- expand.grid(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)
subset.SimDesign

Description
subset.SimDesign is a default method for subsetting a data.frame of class SimDesign. This is a modification of the base R subset command to maintain the extra attributes produced during a simulation.

Usage
## S3 method for class 'SimDesign'
subset(x, ...)

Arguments
x
A data.frame object, of class SimDesign

... Further arguments to be passed to subset

Value
A data.frame/SimDesign class object.

Author(s)
Phil Chalmers <rphilip.chalmers@gmail.com>

References

See Also
SimDesign

Examples
## Not run:
data("BF_sim")
x <- subset(BF_sim, select = 1:6)
attributes(x)
head(x)

x1 <- subset(BF_sim, select = c(1,2,4,5,10))
attributes(x1)

x2 <- subset(BF_sim, select = var_ratio:alpha.05.Jacknife)
attributes(x2)

x3 <- subset(BF_sim, var_ratio == 1)
dim(BF_sim)
dim(x3)

## End(Not run)
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 data.frame (if Analyse returned a numeric vector) 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

must return a named numeric vector or data.frame with the desired meta-simulation results

References


See Also

`bias`, `RMSE`, `RE`, `EDR`, `ECR`, `MAE`

Examples

```r
## Not run:

mysummarise <- 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(less than 0.05 = less than 0.05)
ret
}

## End(Not run)
Index

*Topic **data**
  BF_sim, 9
  BF_sim_alternative, 10
*Topic **package**
  SimDesign, 59

add_missing, 3, 18
aggregate_simulations, 5, 48
Analyse, 6, 17, 42, 43, 46, 48, 66
aov, 56
as.data.frame.SimDesign
  (runSimulation), 40
as.vector, 25
Attach, 8, 18, 48
attach, 8

BF_sim, 9, 10
BF_sim_alternative, 9, 10
bias, 11, 36, 66
boot_predict, 12
browser, 45

cat, 24
clusterSetRNGStream, 43

ECR, 14, 17, 56, 66
EDR, 15, 16, 56, 66
expand.grid, 46
extract_error_seeds (runSimulation), 40
extract_results, 43
extract_results (runSimulation), 40

Generate, 6–8, 17, 41, 42, 44, 46, 48
getwd, 61

head.SimDesign (runSimulation), 40

integrate, 19
IRMSE, 19

library, 42

lm, 13, 56
MAE, 21, 66
makeCluster, 44
MSRSE, 22

print.SimDesign (runSimulation), 40
quiet, 24
RD, 25
RE, 26, 66
readRDS, 61
rejectionSampling, 27
require, 42
rHeadrick, 17, 18, 29
rint, 31
rinvWishart, 32
rmgh, 17, 18, 33
RMSE, 11, 20, 26, 35, 66
rmvnorm, 36
rmvt, 37
rtruncate, 39
runSimulation, 5, 6, 8, 13, 17, 33, 37, 38, 40, 44, 45, 46, 56–59, 61, 63, 66
rvaleMaurelli, 17, 18, 53

sample, 31
sample.int, 31
saveRDS, 47, 58
Serlin2000, 54
set.seed, 43
SimAnova, 46, 48, 55
SimBoot, 42, 48, 57
SimClean, 48, 58
SimDesign, 59, 65
SimDesign-package (SimDesign), 59
SimFunctions, 46, 48, 60
SimResults, 42, 43, 48, 61
SimShiny, 48, 63
stop, 7

68
subset, 46, 56, 65
subset.SimDesign, 64
Summarise, 46, 48, 66
summary.SimDesign(runSimulation), 40
suppressWarnings, 24

tail.SimDesign(runSimulation), 40
try, 7