Package 'SimDesign'

January 26, 2018

Title Structure for Organizing Monte Carlo Simulation Designs

Version 1.9

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]

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

Repository CRAN

Date/Publication 2018-01-26 21:33:53 UTC

R topics documented:

`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

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

Examples

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

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

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

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

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

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

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

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

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

`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

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

See Also

runSimulation

Examples

```
## Not run:

setwd('my_working_directory')

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

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)
```

`Analyse` *Compute estimates and statistics*

Description

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

Usage

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

Arguments

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

Details

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

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

Value

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

References

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

See Also

stop

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)
```

Attach *Attach the simulation conditions for easier reference*

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

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

See Also

runSimulation, Generate

Examples

```
## Not run:

# does not use Attach()
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))
    dat
}

# 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
}

## End(Not run)
```

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

Description

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

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

Examples

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

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

## End(Not run)
```

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

Description

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

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

Examples

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

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

## End(Not run)
```

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

Description

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

Usage

```
bias(estimate, parameter = NULL, type = "bias")
```

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. 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

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

See Also

RMSE

Examples

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

dev <- samp - pop
bias(dev)

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

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

```
bias(mat, parameter = 2, type = 'relative')
bias(mat, parameter = 2, type = 'standardized')

# 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

```
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

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

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

Examples

```
set.seed(4321)
Design <- 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. Alternative, a `numeric` vector object with length equal to the number of rows as `CIs` (use to compare sets of parameters at once)
`tails`	logical; when TRUE returns a vector of length 2 to indicate the proportion of times the parameter was lower or higher than the supplied interval, respectively. This is mainly only useful when the coverage region is not expected to be symmetric, and therefore is generally not required. Note that `1 - sum(ECR(CIs, parameter, tails=TRUE))`

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

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

See Also

EDR

Examples

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

ECR(CIs, 0)
ECR(CIs, 0, tails = TRUE)

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

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

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

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

```
head(CIs)
ECR(CIs, parameter)
ECR(CIs, parameter, tails=TRUE)
ECR(CIs, parameter, CI_width=TRUE)

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

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

Description

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

Usage

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

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

ECR

Examples

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

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

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

`Generate` *Generate data*

Description

Generate data from a single row in the `design` input (see `runSimulation`). R contains numerous approaches to generate data, some of which are contained in the base package, as well as in `SimDesign` (e.g., `rmgh`, `rValeMaurelli`, `rHeadrick`). However the majority can be found in external packages. See CRAN's list of possible distributions here: https://CRAN.R-project.org/view=Distributions

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

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

See Also

add_missing, Attach, rmgh, rValeMaurelli, rHeadrick

Examples

```
## Not run:

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)
```

MAE *Compute the mean absolute error*

Description

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

Usage

```
MAE(estimate, parameter = NULL, type = "MAE")
```

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. 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

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

See Also

RMSE

Examples

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

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

RD *Compute the relative difference*

Description

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

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

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

Examples

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

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

RE *Compute the relative efficiency of multiple estimators*

Description

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

Usage

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

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 <rphilip.chalmers@gmail.com>

References

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

Examples

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

RE(c(RMSE1, RMSE2))

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

---

rejectionSampling *Rejection sampling (i.e., accept-reject method) to draw samples from difficult probability density functions*

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

```
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 <rphilip.chalmers@gmail.com>

References

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

Examples

```
# Generate 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)

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

```
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 <rphilip.chalmers@gmail.com>

References

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

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

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

Examples

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

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

nonnormal <- rHeadrick(N, mean, Sigma, skewness, kurtosis)
# cor(nonnormal)
# psych::describe(nonnormal)

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

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

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

## End(Not run)
```

rint *Generate integer values within specified range*

Description

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

Usage

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

Arguments

`n`	number of samples to draw
`min`	lower limit of the distribution. Must be finite
`max`	upper limit of the distribution. Must be finite
`replace`	should sampling be with replacement?
`prob`	a vector of probability weights for obtaining the elements of the vector being sampled

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

Examples

```
set.seed(1)

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

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

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

```
system.time(x2 <- sample(100:1e8, 1000, replace = TRUE))

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

`rinvWishart` *Generate data with the inverse Wishart distribution*

Description

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

Usage

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

Arguments

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

Value

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

`runSimulation`

Examples

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

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

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

Description

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

Usage

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

Arguments

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

Examples

```
set.seed(1)

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

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

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

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

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

Description

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

Usage

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

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. If `NULL` then it will be assumed that the `estimate` input is in a deviation form (therefore `sqrt(mean(estimate^2))` will be returned)

type	type of deviation to compute. Can be 'RMSE' (default) for the root mean square-error, 'NRMSE' for the normalized RMSE (RMSE / (max(estimate) - min(estimate))), '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

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

See Also

bias

MAE

Examples

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

dev <- samp - pop
RMSE(dev)

RMSE(samp, pop, type = 'NRMSE')
RMSE(dev, type = 'NRMSE')
RMSE(dev, pop, type = '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)

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

# parameters of the same size
```

```
parameters <- 1:10
estimates <- parameters + rnorm(10)
RMSE(estimates, parameters)
```

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

Description

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

Usage

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

Arguments

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

Value

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

`runSimulation`

Examples

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

---

rmvt *Generate data with the multivariate t distribution*

Description

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

Usage

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

Arguments

n	number of observations to generate
sigma	positive definite covariance matrix
df	degrees of freedom. `df = 0` and `df = Inf` corresponds to the multivariate normal distribution
delta	the vector of 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

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

See Also

runSimulation

Examples

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

---

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

Description

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

Usage

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

Arguments

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

Details

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

Value

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

runSimulation

Examples

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

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

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

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

runSimulation *Run a Monte Carlo simulation given a data.frame of conditions and simulation functions*

Description

This function runs a Monte Carlo simulation study given a set of predefined simulation functions, design conditions, and number of replications. Results can be saved as temporary files in case of interruptions and may be restored by re-running `runSimulation`, provided that the respective temp file can be found in the working directory. `runSimulation` supports parallel and cluster computing, global and local debugging, error handling (including fail-safe stopping when functions fail too often, even across nodes), provides bootstrap estimates of the sampling variability (optional), and 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

```
runSimulation(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 = FALSE,
  verbose = TRUE)

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

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

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

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

extract_results(object)
```

```
## 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
analyse	user-defined computation function which acts on the data generated from Generate. 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 form Analyse. Ommiting 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 enviroment
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-statatistic 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 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.

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

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 (handy 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.seeds` have be saved (after a call with `save_seeds = TRUE`). E.g., `load_seed = 'design-row-2/seed-1'` will load the first seed in the second row of the `design` input. Note that it is important NOT to modify the `design` input object, otherwise the path may not point to the correct saved location. 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'])`

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
```

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 the to 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.

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.

Cluster computing

SimDesign code may be released to a computing system which supports parallel cluster computations using the industry standard Message Passing Interface (MPI) form. This simply requires that the computers be setup using the usual MPI requirements (typically, running some flavor of Linux, have password-less open-SSH access, IP addresses have been added to the `/etc/hosts` file or `~/.ssh/config`, etc). More generally though, these resources are widely available through professional organizations dedicated to super-computing.

To setup the R code for an MPI cluster one need only add the argument `MPI = TRUE`, wrap the appropriate MPI directives around `runSimulation`, and submit the files using the suitable BASH commands to execute the `mpirun` tool. For example,

```
library(doMPI)

cl <- startMPIcluster()

registerDoMPI(cl)

runSimulation(design=Design, replications=1000, save=TRUE, filename='mysimulation', generate=Genera


closeCluster(cl)

mpi.quit()
```

The necessary `SimDesign` files must be uploaded to the dedicated master node so that a BASH call to `mpirun` can be used to distribute the work across slaves. For instance, if the following BASH command is run on the master node then 16 processes will be summoned (1 master, 15 slaves) across the computers named `localhost`, `slave1`, and `slave2` in the ssh `config` file.

```
mpirun -np 16 -H localhost,slave1,slave2 R --slave -f simulation.R
```

Network computing

If you access have to a set of computers which can be linked via secure-shell (ssh) on the same LAN network then Network computing (a.k.a., a Beowulf cluster) may be a viable and useful option. This approach is similar to MPI computing approach except that it offers more localized control and requires more hands-on administrative access to the master and slave nodes. The setup generally requires that the master node has `SimDesign` installed and the slave/master nodes have all the required R packages pre-installed (Unix utilities such as `dsh` are very useful for this purpose). Finally, the master node must have ssh access to the slave nodes, each slave node must have ssh access with the master node, and a cluster object (`cl`) from the `parallel` package must be defined on the master node.

Setup for network computing is generally more straightforward and controlled than the setup for MPI jobs in that it only requires the specification of a) the respective IP addresses within a defined R script, and b) the user name (if different from the master node's user name. Otherwise, only a) is required). However, on Linux I have found it is also important to include relevant information about the host names and IP addresses in the `/etc/hosts` file on the master and slave nodes, and

to ensure that the selected port (passed to makeCluster) on the master node is not hindered by a firewall.

As an example, using the following code the master node (primary) will spawn 7 slaves and 1 master, while a separate computer on the network with the associated IP address will spawn an additional 6 slaves. Information will be collected on the master node, which is also where the files and objects will be saved using the save inputs (if requested).

```
library(parallel)
primary <- '192.168.2.1'
IPs <- list(list(host=primary, user='myname', ncore=8), list(host='192.168.2.2', user='myname', nco

spec <- lapply(IPs, function(IP) rep(list(list(host=IP$host, user=IP$user)), IP$ncore))

spec <- unlist(spec, recursive=FALSE)
cl <- makeCluster(master=primary, spec=spec)
Final <- runSimulation(..., cl=cl)
stopCluster(cl)
```

The object cl is passed to runSimulation on the master node and the computations are distributed across the respective IP addresses. Finally, it's usually good practice to use stopCluster(cl) when all the simulations are said and done to release the communication between the computers, which is what the above code shows.

Alternatively, if you have provided suitable names for each respective slave node, as well as the master, then you can define the cl object using these instead (rather than supplying the IP addresses in your R script). This requires that the master node has itself and all the slave nodes defined in the /etc/hosts and ~/.ssh/config files, while the slave nodes require themselves and the master node in the same files (only 2 IP addresses required on each slave). Following this setup, and assuming the user name is the same across all nodes, the cl object could instead be defined with

```
library(parallel)
primary <- 'master'
IPs <- list(list(host=primary, ncore=8), list(host='slave', ncore=6))
spec <- lapply(IPs, function(IP) rep(list(list(host=IP$host)), IP$ncore))
spec <- unlist(spec, recursive=FALSE)
cl <- makeCluster(master=primary, spec=spec)
Final <- runSimulation(..., cl=cl)
stopCluster(cl)
```

Or, even more succinctly if all communication elements required are identical to the master node,

```
library(parallel)
primary <- 'master'
spec <- c(rep(primary, 8), rep('slave', 6))
cl <- makeCluster(master=primary, spec=spec)
Final <- runSimulation(..., cl=cl)
stopCluster(cl)
```

Poor man's cluster computing for independent nodes

In the event that you do not have access to a Beowulf-type cluster (described in the section on "Network Computing") but have multiple personal computers then the simulation code can be manually distributed across each independent computer instead. This simply requires passing a smaller value to the `replications` argument on each computer and later aggregating the results using the `aggregate_simulations` function.

For instance, if you have two computers available on different networks and wanted a total of 500 replications you could pass `replications = 300` to one computer and `replications = 200` to the other along with a `filename` argument (or simply saving the final objects as `.rds` files manually after `runSimulation()` has finished). This will create two distinct `.rds` files which can be combined later with the `aggregate_simulations` function. The benefit of this approach over MPI or setting up a Beowulf cluster is that computers need not be linked on the same network, and, should the need arise, the temporary simulation results can be migrated to another computer in case of a complete hardware failure by moving the saved temp files to another node, modifying the suitable `compname` input to `save_details` (or, if the `filename` and `tmpfilename` were modified, matching those files accordingly), and resuming the simulation as normal.

Note that this is also a useful tactic if the MPI or Network computing options require you to submit smaller jobs due to time and resource constraint-related reasons, where fewer replications/nodes should be requested. After all the jobs are completed and saved to their respective files, `aggregate_simulations` can then collapse the files as if the simulations were run all at once. Hence, SimDesign makes submitting smaller jobs to super-computing resources considerably less error prone than managing a number of smaller jobs manually .

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

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

Examples

```
#-------------------------------------------------------------------------
# Example 1: Sampling distribution of mean

# This example demonstrate some of the simpler uses of SimDesign,
# particularly for classroom settings. The only factor varied in this simulation
# is sample size.

# skeleton functions to be saved and edited
SimFunctions()
```

```
#### Step 1 --- Define your conditions under study and create design data.frame

Design <- data.frame(N = c(10, 20, 30))

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

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

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

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


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

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

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


#~~~~~~~~~~~~~~~~~~~~~~~~~
#### Extras
# 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,
                       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){
    welch <- t.test(DV ~ group, dat)
    ind <- t.test(DV ~ group, dat, var.equal=TRUE)

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

Summarise <- function(condition, results, fixed_objects = NULL){
    #find results of interest here (e.g., alpha < .1, .05, .01)
    ret <- EDR(results, alpha = .05)
    ret
}


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

# first, test to see if it works
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',
              generate=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,
              generate=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(design=Design, replications=1000,
              generate=Generate, analyse=Analyse, summarise=Summarise,
              parallel=TRUE)



## EXTRA: To run the simulation on a MPI cluster, use the following setup 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))
# spec <- unlist(spec, recursive=FALSE)
#
# 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)

#~~~~~~~~~~~~~~~~~~~~~~~~
###### Post-analysis: Analyze the results via functions like lm() or SimAnova(), and create
###### tables(dplyr) or plots (ggplot2) to help visualize the results.
###### This is where you get to be a data analyst!

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

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

ggplot(dd, aes(factor(group_size_ratio), value, fill = factor(standard_deviation_ratio))) +
    geom_boxplot() + geom_abline(intercept=0.05, slope=0, col = 'red') +
    geom_abline(intercept=0.075, slope=0, col = 'red', linetype='dotted') +
    geom_abline(intercept=0.025, slope=0, col = 'red', linetype='dotted') +
    facet_grid(variable~standard_deviation_ratio) +
    theme(legend.position = 'none')


## End(Not run)
```

`rValeMaurelli` *Generate non-normal data with Vale & Maurelli's (1983) method*

Description

Generate multivariate non-normal distributions using the third-order polynomial method described by Vale & Maurelli (1983). If only a single variable is generated then this function is equivalent to the method described by Fleishman (1978).

Usage

```
rValeMaurelli(n, mean = rep(0, nrow(sigma)), sigma = diag(length(mean)),
  skew = rep(0, nrow(sigma)), kurt = rep(0, nrow(sigma)))
```

Arguments

`n`	number of samples to draw
`mean`	a vector of k elements for the mean of the variables
`sigma`	desired k x k covariance matrix between bivariate non-normal variables
`skew`	a vector of k elements for the skewness of the variables
`kurt`	a vector of k elements for the kurtosis of the variables

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

Fleishman, A. I. (1978). A method for simulating non-normal distributions. *Psychometrika, 43*, 521-532.

Vale, C. & Maurelli, V. (1983). Simulating multivariate nonnormal distributions. *Psychometrika, 48*(3), 465-471.

Examples

```
set.seed(1)

# univariate with skew
nonnormal <- rValeMaurelli(10000, mean=10, sigma=5, skew=1, kurt=3)
# psych::describe(nonnormal)

# multivariate with skew and kurtosis
n <- 10000
r12 <- .4
r13 <- .9
r23 <- .1
cor <- matrix(c(1,r12,r13,r12,1,r23,r13,r23,1),3,3)
sk <- c(1.5,1.5,0.5)
ku <- c(3.75,3.5,0.5)

nonnormal <- rValeMaurelli(n, sigma=cor, skew=sk, kurt=ku)
# cor(nonnormal)
# psych::describe(nonnormal)
```

Serlin2000 *Empirical detection robustness method suggested by Serlin (2000)*

Description

Hypothesis test to determine whether an observed empirical detection rate, coupled with a given robustness interval, statistically differs from the population value. Uses the methods described by Serlin (2000) as well to generate critical values (similar to confidence intervals, but define a fixed window of robustness). Critical values may be computed without performing the simulation experiment (hence, can be obtained a priori).

Usage

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

Arguments

p	(optional) a vector containing the empirical detection rate(s) to be tested. Omitting this input will compute only the CV1 and CV2 values, while including this input will perform a one-sided hypthesis 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

Serlin, R. C. (2000). Testing for Robustness in Monte Carlo Studies. *Psychological Methods, 5*, 230-240.

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

Examples

```
# Cochran's criteria at alpha = .05 (i.e., 0.5 +- .01), assuming N = 2000
Serlin2000(p = .051, alpha = .05, delta = .01, R = 2000)

# Bradley's liberal criteria given p = .06 and .076, assuming N = 1000
Serlin2000(p = .060, alpha = .05, delta = .025, R = 1000)
Serlin2000(p = .076, alpha = .05, delta = .025, R = 1000)

# multiple p-values
Serlin2000(p = c(.05, .06, .07), alpha = .05, delta = .025, R = 1000)

# CV values computed before simulation performed
Serlin2000(alpha = .05, R = 2500)
```

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

Description

Given the results from a simulation with runSimulation form an ANOVA table (without p-values) with effect sizes based on the eta-squared statistic. These results provide approximate indications of observable simulation effects, therefore these ANOVA-based results are generally useful as exploratory rather than inferential tools.

Usage

```
SimAnova(formula, dat, subset = NULL, rates = TRUE)
```

Arguments

formula	an R formula generally of a form suitable for lm or aov. However, if the dependent variable (left size of the equation) is omitted then all the dependent variables in the simulation will be used and the result will return a list of analyses
dat	an object returned from runSimulation of class 'SimDesign'
subset	an optional argument to be passed to subset with the same name. Used to subset the results object while preserving the associated attributes
rates	logical; does the dependent variable consist of rates (e.g., returned from ECR or EDR)? Default is TRUE, which will use the logit of the DV to help stabilize the proportion-based summary statistics when computing the parameters and effect sizes

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

Examples

```
data(BF_sim)

# all results (not usually good to mix Power and Type I results together)
SimAnova(alpha.05.F ~ (groups_equal + distribution)^2, BF_sim)

# only use anova for Type I error conditions
SimAnova(alpha.05.F ~ (groups_equal + distribution)^2, BF_sim, subset = var_ratio == 1)

# run all DVs at once using the same formula
SimAnova(~ groups_equal * distribution, BF_sim, subset = var_ratio == 1)
```

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

Description

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

Usage

```
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

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

Examples

```
#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){
   ret <- c(mu=mean(results[,1]), SE=sd(results[,1]), # mean and SD summary of the sample means
```

```
                coverage=ECR(results[,2:3], parameter = 10))
    ret
}

res <- runSimulation(design=Design, replications=250, bootSE=TRUE,
                     generate=Generate, analyse=Analyse, summarise=Summarise)

res
SimBoot(res)

# larger R
res2 <- runSimulation(design=Design, replications=2500, bootSE=TRUE,
                     generate=Generate, analyse=Analyse, summarise=Summarise)

# point estimates more accurate, smaller BOOT_SE terms
res2
SimBoot(res2) # more reasonable CI range
```

`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`)

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

runSimulation

Examples

```
## Not run:

# remove file called 'results.rds'
SimClean('results.rds')

# remove default temp file
SimClean(temp = TRUE)

# remove default saved-data directory
SimClean(generate_data = TRUE)

# remove customized saved-results directory called 'mydir'
SimClean(results = TRUE, save_details = list(save_results_dirname = 'mydir'))


## End(Not run)
```

SimDesign *Structure for Organizing Monte Carlo Simulation Designs*

Description

Structure for Organizing Monte Carlo Simulation Designs

Details

Provides tools to help organize Monte Carlo simulations in R. The package controls the structure and back-end of Monte Carlo simulations by utilizing a general generate-analyse-summarise strategy. The functions provided control common simulation issues such as re-simulating non-convergent results, support parallel back-end and MPI distributed computations, save and restore temporary files, aggregate results across independent nodes, and provide native support for debugging. The primary function for organizing the simulations is runSimulation. For 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.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

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) 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)
```

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

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

Examples

```
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 conviencience function will read the desired rows in automatically given the returned object from the simulation. Can be used to read in 1 or more .rds files at once (if more than 1 file is read in then the result will be stored in a list).

Usage

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

Arguments

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

Value

the returned result is either a nested list (when `length(which) > 1`) or a single list (when `length(which) == 1`) containing the simulation results. Each read-in result refers to a list of 4 elements:

`condition` the associate row (ID) and conditions from the respective `design` object

`results` the object with returned from the `analyse` function, potentially simplified into a matrix or data.frame

`errors` a table containing the message and number of errors that caused the generate-analyse steps to be rerun. These should be inspected carefully as they could indicate validity issues with the simulation that should be noted

`warnings` a table containing the message and number of non-fatal warnings which arose from the analyse step. These should be inspected carefully as they could indicate validity issues with the simulation that should be noted

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

Examples

```
## Not run:

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

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

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

# all results
rows_all <- SimResults(results)


## End(Not run)
```

SimShiny *Generate a basic Monte Carlo simulation GUI template*

Description

This function generates suitable stand-alone code from the shiny package to create simple web-interfaces for performing single condition Monte Carlo simulations. The template generated is relatively minimalistic, but allows the user to quickly and easily edit the saved files to customize the associated shiny elements as they see fit.

Usage

```
SimShiny(filename = NULL, dir = getwd(), design, ...)
```

Arguments

filename	an optional name of a text file to save the server and UI components (e.g., 'mysimGUI.R'). If omitted, the code will be printed to the R console instead
dir	the directory to write the files to. Default is the working directory
design	design object from runSimulation
...	arguments to be passed to runSimulation. Note that the design object is not used directly, and instead provides options to be selected in the GUI

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

runSimulation

Examples

```
## Not run:

Design <- 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` *Subset method for SimDesign objects*

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, subset, select, drop = FALSE, ...)
```

Arguments

x	A data.frame object, of class SimDesign.
subset	A logical expression indicating elements or rows to keep.
select	An expression, indicating columns to select from a dataframe.
drop	Additional arguments passed on to [indexing operator.
...	Further arguments to be passed to or from other methods.

Value

A data.frame object.

Author(s)

Phil Chalmers <rphilip.chalmers@gmail.com>

References

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

See Also

SimDesign

Examples

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

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

See Also

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

Examples

```
## 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(lessthan.05=lessthan.05)
    ret
}


## End(Not run)
```

Index