Package 'SimEngine'

January 20, 2025

Type Package

Title A Modular Framework for Statistical Simulations in R

Version 1.4.0

Description An open-source R package for structuring, maintaining, running, and debugging statistical simulations on both local and cluster-based computing environments.See full documentation at <https://avi-kenny.github.io/SimEngine/>.

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.1

VignetteBuilder knitr

Depends magrittr (>= 2.0.3)

Imports dplyr (>= 1.0.10), parallel (>= 4.2.2), pbapply (>= 1.6.0), data.table (>= 1.14.6), rlang (>= 1.0.6), methods (>= 4.2.2), stats (>= 4.0.0), utils (>= 4.2.2), MASS (>= 7.3.50)

Suggests covr (>= 3.6.1), knitr (>= 1.41), rmarkdown (>= 2.19), testthat (>= 3.1.6), tidyr (>= 1.2.1), ggplot2 (>= 3.4.0), sandwich (>= 3.0.2)

URL https://avi-kenny.github.io/SimEngine/

BugReports https://github.com/Avi-Kenny/SimEngine/issues

NeedsCompilation no

Author Avi Kenny [aut, cre],
Charles Wolock [aut]

Maintainer Avi Kenny <avi.kenny@gmail.com>

Repository CRAN

Date/Publication 2024-04-04 03:22:58 UTC

Contents

`batch` *Run a block of code as part of a batch*

Description

This function is useful for sharing data or objects between simulation replicates. Essentially, it allows simulation replicates to be divided into "batches"; all replicates in a given batch will then share a certain set of objects. A common use case for this is a simulation that involves using multiple methods to analyze a shared dataset, and repeating this process over a number of dataset replicates. See the Advanced Functionality vignette for a detailed overview of how this function is used.

Usage

```
batch(code)
```

Arguments

code	A block of code enclosed by curly braces { }; see examples.

Examples

```
sim <- new_sim()
create_data <- function(n, mu) { rnorm(n=n, mean=mu) }
est_mean <- function(dat, type) {
  if (type=="est_mean") { return(mean(dat)) }
  if (type=="est_median") { return(median(dat)) }
}
sim %<>% set_levels(n=c(10,100), mu=c(3,5), est=c("est_mean","est_median"))
sim %<>% set_config(
  num_sim = 2,
  batch_levels = c("n","mu"),
  return_batch_id = TRUE
)
sim %<>% set_script(function() {
  batch({
    dat <- create_data(n=L$n, mu=L$mu)
  })
  mu_hat <- est_mean(dat=dat, type=L$est)
  return(list(
    "mu_hat" = round(mu_hat,2),
    "dat_1" = round(dat[1],2)
  ))
})
sim %<>% run()
sim$results[order(sim$results$batch_id),]
```

---

get_complex *Access internal simulation variables*

Description

Extract complex simulation data from a simulation object

Usage

```
get_complex(sim, sim_uid)
```

Arguments

sim	A simulation object of class `sim_obj`, usually created by `new_sim`
sim_uid	The unique identifier of a single simulation replicate. This corresponds to the `sim_uid` column in `sim$results`.

Value

The value of the complex simulation result data corresponding to the supplied `sim_uid`

Examples

```
sim <- new_sim()
sim %<>% set_levels(n=c(10, 100, 1000))
create_data <- function(n) {
  x <- runif(n)
  y <- 3 + 2*x + rnorm(n)
  return(data.frame("x"=x, "y"=y))
}
sim %<>% set_config(num_sim=2)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  model <- lm(y~x, data=dat)
  return(list(
    "beta0_hat" = model$coefficients[[1]],
    "beta1_hat" = model$coefficients[[2]],
    ".complex" = list(
      "model" = model,
      "cov_mtx" = vcov(model)
    )
  ))
})
sim %<>% run()
c5 <- get_complex(sim, sim_uid=5)
print(summary(c5$model))
print(c5$cov_mtx)
```

js_support *Display information about currently-supported job schedulers*

Description

Run this function to display information about job schedulers that are currently supported for running **SimEngine** simulations on a cluster computing system (CCS).

Usage

```
js_support()
```

Examples

```
js_support()
```

new_sim *Create a new simulation object*

Description

Create a new simulation object. This is typically the first function to be called when running a simulation using **SimEngine**. Most other **SimEngine** functions take a simulation object as their first argument.

Usage

```
new_sim()
```

Value

A simulation object, of class `sim_obj`

See Also

Visit https://avi-kenny.github.io/SimEngine/ for more information on how to use the **SimEngine** simulation framework.

Examples

```
sim <- new_sim()
print(sim)
```

---

run *Run the simulation*

Description

This is the workhorse function of **SimEngine** that actually runs the simulation. This should be called after all functions that set up the simulation (`set_config`, `set_script`, etc.) have been called.

Usage

```
run(sim)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by `new_sim`

Value

The original simulation object but with the results attached (along with any errors and warnings). Results are stored in `sim$results`, errors are stored in `sim$errors`, and warnings are stored in `sim$warnings`.

Examples

```
sim <- new_sim()
create_data <- function(n) { rpois(n, lambda=5) }
est_mean <- function(dat, type) {
  if (type=="M") { return(mean(dat)) }
  if (type=="V") { return(var(dat)) }
}
sim %<>% set_levels(n=c(10,100,1000), est=c("M","V"))
sim %<>% set_config(num_sim=1)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  lambda_hat <- est_mean(dat=dat, type=L$est)
  return (list("lambda_hat"=lambda_hat))
})
sim %<>% run()
sim$results %>% print()
```

run_on_cluster *Framework for running simulations on a cluster computing system*

Description

This function allows for simulations to be run in parallel on a cluster computing system (CCS). See the Parallelization vignette for a detailed overview of how CCS parallelization works in **SimEngine**. `run_on_cluster` acts as a wrapper for the code in your simulation, organizing the code into three sections, labeled "first" (code that is run once at the start of the simulation), "main" (running the simulation script repeatedly), and "last" (code to process or summarize simulation results). This function is to be used in conjunction with job scheduler software (e.g., Slurm or Oracle Grid Engine) to divide the simulation into tasks that are run in parallel on the CCS. See the Parallelization documentation for a detailed overview of how CCS parallelization works in **SimEngine**. `run`)), and "last" (usually code to process or summarize simulation results). This function interacts with cluster job scheduler software (e.g. Slurm or Oracle Grid Engine) to divide parallel tasks over cluster nodes.

Usage

```
run_on_cluster(first, main, last, cluster_config)
```

Arguments

`first` Code to run at the start of a simulation. This should be a block of code enclosed by curly braces that creates and sets up a simulation object.

`main` Code that will run for every simulation replicate. This should be a block of code enclosed by curly braces , and will typically be a single line of code calling the `run`) function. This code block will have access to the simulation object you created in the 'first' code block, but any changes made here to the simulation object will not be saved.

`last` Code that will run after all simulation replicates have been run. This should be a block of code enclosed by curly braces that processes your simulation object (which at this point will contain your results), which may involve calls to `summarize`, creation of plots, and so on.

`cluster_config` A list of configuration options. You must specify either `js` (the job scheduler you are using) or `tid_var` (the name of the environment variable that your task ID is stored in); see examples. Run `js_support()` to see a list of job schedulers that are currently supported. You can optionally also specify `dir`, which is a character string representing a path to a directory on the CCS; this directory will serve as your working directory and hold your simulation object and all temporary objects created by **SimEngine**. If unspecified, this defaults to the working directory of the R script that contains your simulation code).

Examples

```
## Not run:
```

```
# The following code is saved in a file called my_simulation.R:
library(SimEngine)
run_on_cluster(
  first = {
    sim <- new_sim()
    create_data <- function(n) { return(rpois(n=n, lambda=20)) }
    est_lambda <- function(dat, type) {
      if (type=="M") { return(mean(dat)) }
      if (type=="V") { return(var(dat)) }
    }
    sim %<>% set_levels(estimator = c("M","V"), n = c(10,100,1000))
    sim %<>% set_script(function() {
      dat <- create_data(L$n)
      lambda_hat <- est_lambda(dat=dat, type=L$estimator)
      return(list("lambda_hat"=lambda_hat))
    })
    sim %<>% set_config(num_sim=100, n_cores=20)
  },
  main = {
    sim %<>% run()
  },
  last = {
    sim %>% summarize()
  },
  cluster_config = list(js="slurm")
)

# The following code is saved in a file called run_sim.sh:
# #!/bin/bash
# Rscript my_simulation.R

# The following lines of code are run on the CCS head node:
# sbatch --export=sim_run='first' run_sim.sh
# sbatch --export=sim_run='main' --array=1-20 --depend=afterok:101 run_sim.sh
# sbatch --export=sim_run='last' --depend=afterok:102 run_sim.sh

## End(Not run)
```

set_config *Modify the simulation configuration*

Description

This function sets configuration options for the simulation. If the 'packages' argument is specified, all packages will be loaded and attached via `library` when `set_config` is called. Multiple calls to `set_config` will only overwrite configuration options that are specified in the subsequent calls, leaving others in place. You can see the current configuration via `print(sim)`, where `sim` is your simulation object.

Usage

```
set_config(
  sim,
  num_sim = 1000,
  parallel = FALSE,
  n_cores = NA,
  packages = NULL,
  stop_at_error = FALSE,
  progress_bar = TRUE,
  seed = as.integer(1e+09 * runif(1)),
  batch_levels = NA,
  return_batch_id = FALSE
)
```

Arguments

`sim` A simulation object of class `sim_obj`, usually created by `new_sim`

`num_sim` An integer; the number of simulations to conduct for each level combination

`parallel` Boolean; if set to TRUE, **SimEngine** will run one simulation per core. if set to FALSE, code will not be parallelized. See the Parallelization vignette for an overview of how parallelization works in **SimEngine**.

`n_cores` An integer; determines the number of cores on which the simulation will run if using parallelization. Defaults to one fewer than the number of available cores.

`packages` A character vector of packages to load and attach

`stop_at_error` Boolean; if set to TRUE, the simulation will stop if it encounters an error in any single replicate Useful for debugging.

`progress_bar` Boolean; if set to FALSE, the progress bar that is normally displayed while the simulation is running is suppressed.

`seed` An integer; seeds allow for reproducible simulation results. If a seed is specified, then consecutive runs of the same simulation with the same seed will lead to identical results (under normal circumstances). If a seed was not set in advance by the user, **SimEngine** will set a random seed, which can later be retrieved using the `vars` function. See details for further info.

`batch_levels` Either NULL or a character vector. If the `batch` function is being used within the simulation script, this should contain the names of the simulation levels that are used within the `batch` function code block. If no simulation levels are used within the `batch` function code block, specify NULL. See the documentation for the `batch` function.

`return_batch_id` Boolean. If set to TRUE, the batch_id will be included as part of the simulation results

Details

- If a user specifies, for example, `set_config(seed=4)`, this seed is used twice by **SimEngine**. First, **SimEngine** executes `set.seed(4)` at the end of the `set_config` call. Second, this seed is used to generate a new set of seeds, one for each simulation replicate. Each of these seeds is set in turn (or in parallel) when `run` is called.

- Even if seeds are used, not all code will be reproducible. For example, a simulation that involves getting the current date/time with `Sys.time` or dynamically retrieving external data may produce different results on different runs.

Value

The original simulation object with a modified configuration

Examples

```
sim <- new_sim()
sim %<>% set_config(
  num_sim = 10,
  seed = 2112
)
print(sim)
```

`set_levels` *Set simulation levels*

Description

Set one or more simulation levels, which are things that vary between simulation replicates.

Usage

```
set_levels(sim, ..., .keep = NA)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by `new_sim`
`...`	One or more key-value pairs representing simulation levels. Each value can either be a vector (for simple levels) or a list of lists (for more complex levels). See examples.
`.keep`	An integer vector of level_id values specifying which level combinations to keep; see the Advanced Functionality documentation.

Value

The original simulation object with the old set of levels replaced with the new set

Examples

```
# Basic simulation levels are numeric or character vectors
sim <- new_sim()
sim %<>% set_levels(
  n = c(10, 100, 1000),
  est = c("M", "V")
)

# Complex simulation levels can be set using named lists of lists
sim <- new_sim()
sim %<>% set_levels(
  n = c(10, 100, 1000),
  distribution = list(
    "Beta 1" = list(type="Beta", params=c(0.3, 0.7)),
    "Beta 2" = list(type="Beta", params=c(1.5, 0.4)),
    "Normal" = list(type="Normal", params=c(3.0, 0.2))
  )
)
```

`set_script` *Set the "simulation script"*

Description

Specify a function to be used as the "simulation script". The simulation script is a function that runs a single simulation replicate and returns the results.

Usage

```
set_script(sim, fn)
```

Arguments

`sim` A simulation object of class `sim_obj`, usually created by `new_sim`

`fn` A function that runs a single simulation replicate and returns the results. The results must be a list of key-value pairs. Values are categorized as simple (a number, a character string, etc.) or complex (vectors, dataframes, lists, etc.). Complex data must go inside a key called ".complex" and the associated value must be a list (see Advanced Functionality documentation and examples). The function body can contain references to the special object `L` that stores the current set of simulation levels (see examples). The keys must be valid R names (see ?make.names). Any functions used within the script must be declared before `set_script` is called.

Value

The original simulation object with the new "simulation script" function added.

Examples

```
sim <- new_sim()
create_data <- function(n) { rpois(n, lambda=5) }
est_mean <- function(dat, type) {
  if (type=="M") { return(mean(dat)) }
  if (type=="V") { return(var(dat)) }
}
sim %<>% set_levels(n=c(10,100,1000), est=c("M","V"))
sim %<>% set_config(num_sim=1)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  lambda_hat <- est_mean(dat=dat, type=L$est)
  return (list("lambda_hat"=lambda_hat))
})
sim %<>% run()
sim$results %>% print()

# To return complex result data, use the special key ".complex".
sim <- new_sim()
create_data <- function(n) {
  x <- runif(n)
  y <- 3 + 2*x + rnorm(n)
  return(data.frame("x"=x, "y"=y))
}
sim %<>% set_levels("n"=c(10, 100, 1000))
sim %<>% set_config(num_sim=1)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  model <- lm(y~x, data=dat)
  return (list(
    "beta1_hat" = model$coefficients[[2]],
    ".complex" = model
  ))
})
sim %<>% run()
sim$results %>% print()
get_complex(sim, 1) %>% print()
```

summarize *Summarize simulation results*

Description

This function calculates summary statistics for simulation results, including descriptive statistics (e.g. measures of center or spread) and inferential statistics (e.g. bias or confidence interval coverage). All summary statistics are calculated over simulation replicates within a single simulation level.

Usage

```
summarize(sim, ..., mc_se = FALSE)
```

Arguments

`sim` A simulation object of class `sim_obj`, usually created by `new_sim`

`...` One or more lists, separated by commas, specifying desired summaries of the `sim` simulation object. See examples. Each list must have a `stat` item, which specifies the type of summary statistic to be calculated. The `na.rm` item indicates whether to exclude `NA` values when performing the calculation (with default being `FALSE`). For `stat` options where the `name` item is optional, if it is not provided, a name will be formed from the type of summary and the column on which the summary is performed. Additional required items are detailed below for each `stat` type.

- `list(stat="mean", x="col_1", name="mean_col", na.rm=F)` computes the mean of column `sim$results$col_1` for each level combination and creates a summary column named `"mean_col"`. Other single-column summary statistics (see the next few items) work analogously. `name` is optional.
- `list(stat="median", ...)` computes the median.
- `list(stat="var", ...)` computes the variance.
- `list(stat="sd", ...)` computes the standard deviation.
- `list(stat="mad", ...)` computes the mean absolute deviation.
- `list(stat="iqr", ...)` computes the interquartile range.
- `list(stat="min", ...)` computes the minimum.
- `list(stat="max", ...)` computes the maximum.
- `list(stat="is_na", ...)` computes the number of NA values.
- `list(stat="correlation", x="col_1", y="col_2", name="cor_12")` computes the (Pearson's) correlation coefficient between `sim$results$col_1` and `sim$results$col_2` for each level combination and creates a summary column named `"cor_12"`.
- `list(stat="covariance", x="col_1", y="col_2", name="cov_12")` computes the covariance between `sim$results$col_1` and `sim$results$col_2` for each level combination and creates a summary column named `"cov_12"`.
- `list(stat="quantile", x="col_1", prob=0.8, name="q_col_1")` computes the 0.8 quantile of column `sim$results$col_1` and creates a summary column named `"q_col_1"`. `prob` can be any number in [0,1].
- `list(stat="bias", estimate="est", truth=5, name="bias_est")` computes the absolute bias of the estimator corresponding to column `"sim$results$est"`, relative to the true value given in `truth`, and creates a summary column named `"bias_est"`. `name` is optional. See *Details*.
- `list(stat="bias_pct", estimate="est", truth=5, name="bias_est")` computes the percent bias of the estimator corresponding to column `"sim$results$est"`, relative to the true value given in `truth`, and creates a summary column named `"bias_pct_est"`. `name` is optional. See *Details*.
- `list(stat="mse", estimate="est", truth=5, name="mse_est")` computes the mean squared error of the estimator corresponding to column `"sim$results$est"`, relative to the true value given in `truth`, and creates a summary column named `"mse_est"`. `name` is optional. See *Details*.

- `list(stat="mae", estimate="est", truth=5, name="mae_est")` computes the mean absolute error of the estimator corresponding to column `"sim$results$est"`, relative to the true value given in `truth`, and creates a summary column named `"mae_est"`. `name` is optional. See *Details*.
- `list(stat="coverage", estimate="est", se="se_est", truth=5, name="cov_est")` or `list(stat="coverage", lower="est_l", upper="est_u", truth=5, name="cov_est")` computes confidence interval coverage. With the first form, `estimate` gives the name of the variable in `sim$results` corresponding to the estimator of interest and `se` gives the name of the variable containing the standard error of the estimator of interest. With the second form, `lower` gives the name of the variable containing the confidence interval lower bound and `upper` gives the name of the confidence interval upper bound. In both cases, `truth` is the true value (see *Details*), and a summary column named `"cov_est"` is created.

`mc_se` A logical argument indicating whether to compute Monte Carlo standard error and associated confidence interval for inferential summary statistics. This applies only to the `bias`, `bias_pct`, `mse`, `mae`, and `coverage` summary statistics.

Details

- For all inferential summaries there are three ways to specify `truth`: (1) a single number, meaning the estimand is the same across all simulation replicates and levels, (2) a numeric vector of the same length as the number of rows in `sim$results`, or (3) the name of a variable in `sim$results` containing the estimand of interest.
- There are two ways to specify the confidence interval bounds for `coverage`. The first is to provide an `estimate` and its associated `se` (standard error). These should both be variables in `sim$results`. The function constructs a 95% Wald-type confidence interval of the form `(estimate-1.96*se, estimate+1.96*se)`. The alternative is to provide `lower` and `upper` bounds, which should also be variables in `sim$results`. In this case, the confidence interval is (`lower`, `upper`). The coverage is the proportion of simulation replicates for a given level combination in which `truth` lies within the interval.

Value

A data frame containing the result of each specified summary function as a column, for each of the simulation levels. The column `n_reps` returns the number of successful simulation replicates within each level.

Examples

```
sim <- new_sim()
create_data <- function(n) { rpois(n, lambda=5) }
est_mean <- function(dat, type) {
  if (type=="M") { return(mean(dat)) }
  if (type=="V") { return(var(dat)) }
}
sim %<>% set_levels(n=c(10,100,1000), est=c("M","V"))
sim %<>% set_config(num_sim=5)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  lambda_hat <- est_mean(dat=dat, type=L$est)
  return (list("lambda_hat"=lambda_hat))
})
sim %<>% run()
sim %>% summarize(
  list(stat = "mean", name="mean_lambda_hat", x="lambda_hat"),
  list(stat = "mse", name="lambda_mse", estimate="lambda_hat", truth=5)
)
```

update_sim *Update a simulation*

Description

This function updates a previously run simulation. After a simulation has been run, you can alter the levels of the resulting object of class sim_obj using set_levels, or change the configuration (including the number of simulation replicates) using set_config. Executing update_sim on this simulation object will only run the added levels/replicates, without repeating anything that has already been run.

Usage

```
update_sim(sim, keep_errors = T)
```

Arguments

sim	A simulation object of class sim_obj, usually created by new_sim, that has already been run by the run function
keep_errors	logical (TRUE by default); if TRUE, do not try to re-run simulation reps that results in errors previously; if FALSE, attempt to run those reps again

Details

- It is not possible to add new level variables, only new levels of the existing variables. Because of this, it is best practice to include all potential level variables before initially running a simulation, even if some of them only contain a single level. This way, additional levels can be added later.

Value

The original simulation object with additional simulation replicates in results or errors

Examples

```
sim <- new_sim()
create_data <- function(n) { rpois(n, lambda=5) }
est_mean <- function(dat, type) {
  if (type=="M") { return(mean(dat)) }
  if (type=="V") { return(var(dat)) }
}
sim %<>% set_levels(n=c(10,100), est="M")
sim %<>% set_config(num_sim=10)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  lambda_hat <- est_mean(dat=dat, type=L$est)
  return (list("lambda_hat"=lambda_hat))
})
sim %<>% run()
sim %>% summarize(list(stat="mean", x="lambda_hat"))
sim %<>% set_levels(n=c(10,100,1000), est=c("M","V"))
sim %<>% set_config(num_sim=5)
sim %<>% update_sim()
sim %>% summarize(list(stat="mean", x="lambda_hat"))
```

`update_sim_on_cluster` *Framework for updating simulations on a cluster computing system*

Description

This function allows for simulations to be updated in parallel on a cluster computing system (CCS). See the Parallelization vignette for a detailed overview of how CCS parallelization works in **SimEngine**. Like `run_on_cluster`, the `update_sim_on_cluster` function acts as a wrapper for the code in your simulation, organizing the code into three sections, labeled "first" (code that is run once at the start of the simulation), "main" (running the simulation script repeatedly), and "last" (code to process or summarize simulation results). This function is to be used in conjunction with job scheduler software (e.g., Slurm or Oracle Grid Engine) to divide the simulation into tasks that are run in parallel on the CCS.

Usage

```
update_sim_on_cluster(first, main, last, cluster_config, keep_errors = T)
```

Arguments

`first` Code to run at the start of a simulation. This should be a block of code enclosed by curly braces that reads in a previously-run simulation object via `readRDS` and makes changes to it via `set_levels` or `set_config`.

`main` Code that will run for every simulation replicate. This should be a block of code enclosed by curly braces , and will typically be a single line of code calling the `update_sim`) function. This code block will have access to the simulation object you created in the ’first’ code block, but any changes made here to the simulation object will not be saved.

`last` Code that will run after all simulation replicates have been run. This should be a block of code enclosed by curly braces that processes your simulation object (which at this point will contain your updated results), which may involve calls to `summarize`, creation of plots, and so on.

`cluster_config` A list of configuration options. You must specify either `js` (the job scheduler you are using) or `tid_var` (the name of the environment variable that your task ID is stored in); see examples. Run `js_support()` to see a list of job schedulers that are currently supported. You can optionally also specify `dir`, which is a character string representing a path to a directory on the CCS; this directory will serve as your working directory and hold your simulation object and all temporary objects created by **SimEngine**. If unspecified, this defaults to the working directory of the R script that contains your simulation code).

`keep_errors` logical (`TRUE` by default); if `TRUE`, do not try to re-run simulation reps that results in errors previously; if `FALSE`, attempt to run those reps again

Examples

```
## Not run:
# The following code is saved in a file called my_simulation.R:
library(SimEngine)
update_sim_on_cluster(
  first = {
    sim <- readRDS(”sim.rds”)
    sim %<>% set_levels(n=c(100,500,1000))
  },
  main = {
    sim %<>% update_sim()
  },
  last = {
    sim %>% summarize()
  },
  cluster_config = list(js=”slurm”)
)

# The following code is saved in a file called run_sim.sh:
# #!/bin/bash
# Rscript my_simulation.R

# The following lines of code are run on the CCS head node:
# sbatch --export=sim_run='first' run_sim.sh
# sbatch --export=sim_run='main' --array=1-20 --depend=afterok:101 run_sim.sh
# sbatch --export=sim_run='last' --depend=afterok:102 run_sim.sh

## End(Not run)
```

`use_method` *Use a method*

Description

This function calls the specified method, passing along any arguments that have been specified in `args`. It will typically be used in conjunction with the special object L to dynamically run methods that have been included as simulation levels. This function is a wrapper around do.call and is used in a similar manner. See examples.

Usage

```
use_method(method, args = list())
```

Arguments

`method`	A character string naming a function that has been declared or loaded via `source`.
`args`	A list of arguments to be passed onto `method`

Value

The result of the call to `method`

Examples

```
# The following is a toy example of a simulation, illustrating the use of
# the use_method function.
sim <- new_sim()
create_data <- function(n) { rpois(n, lambda=5) }
est_mean_1 <- function(dat) { mean(dat) }
est_mean_2 <- function(dat) { var(dat) }
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "estimator" = c("est_mean_1", "est_mean_2")
)
sim %<>% set_config(num_sim=1)
sim %<>% set_script(function() {
  dat <- create_data(L$n)
  lambda_hat <- use_method(L$estimator, list(dat))
  return (list("lambda_hat"=lambda_hat))
})
sim %<>% run()
sim$results
```

`vars` *Access internal simulation variables*

Description

This is a "getter function" that returns the value of an internal simulation variable. Do not change any of these variables manually.

Usage

```
vars(sim, var)
```

Arguments

sim	A simulation object of class sim_obj, usually created by new_sim
var	If this argument is omitted, vars will return a list containing all available internal variables. If this argument is provided, it should equal one of the following character strings: • seed: the simulation seed; see set_config for more info on seeds. • env: a reference to the environment in which individual simulation replicates are run (advanced) • num_sim_total: The total number of simulation replicates for the simulation. This is particularly useful when a simulation is being run in parallel on a cluster computing system as a job array and the user needs to know the range of task IDs. • run_state: A character string describing the "run state" of the simulation. This will equal one of the following: "pre run" (the simulation has not yet been run), "run, no errors" (the simulation ran and had no errors), "run, some errors" (the simulation ran and had some errors), "run, all errors" (the simulation ran and all replicates had errors). • session_info: The results of a call to utils::sessionInfo() that occures when new_sim is called.

Value

The value of the internal variable.

Examples

```
sim <- new_sim()
sim %<>% set_levels(n = c(10, 100, 1000))
sim %<>% set_config(num_sim=10)
print(vars(sim, "seed"))
print(vars(sim, "env"))
print(vars(sim, "num_sim_total"))
```

Index