Package 'SimEngine'

September 27, 2021

Type Package

Title An Open-Source Framework for Statistical Simulations in R

Version 1.0.0

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

License GPL-3

Encoding UTF-8

RoxygenNote 7.1.1

Depends magrittr

Imports dplyr, parallel, pbapply, data.table, rlang, methods

Suggests covr, knitr, rmarkdown, testthat (>= 2.1.0), tidyr, ggplot2, sandwich

VignetteBuilder knitr

NeedsCompilation no

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

Maintainer Avi Kenny <avikenny@uw.edu>

Repository CRAN

Date/Publication 2021-09-27 09:50:12 UTC

R topics documented:

`add_constants` *Add one or more simulation constants*

Description

Add one or more simulation constants

Usage

```
add_constants(sim, ...)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by new_sim()
`...`	Key-value pairs will be added as "simulation constants" (i.e. objects that don't change across simulations). Keys should be strings. The purpose of this (rather than "hard-coding" constants in your scripts) is to serve as an organizational container to easily change constants later, and so that constants are automatically available on each cluster node if you decide to run your simulation code in parallel

Value

The original simulation object with added constants

Examples

```
sim <- new_sim()
sim %<>% add_constants(alpha=4, beta=c(1,2,3))
```

`add_creator` *Add a "creator" function*

Description

Add a "creator" function to your simulation object. A creator is a function that generates a dataset for use in your simulation.

Usage

```
add_creator(sim, name, fn)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by `new_sim`
`name`	A name for the creator function
`fn`	A creator function

Details

- There are two ways to use `add_creator`. If two arguments are supplied (`sim` and `fn`), you can create a function separately and add it to your simulation object later. If three arguments are supplied, you can do both at the same time, using an anonymous function for the `fn` argument. See examples.
- Your creator will be stored in `sim$creators`. If you added a creator called `create_data`, you can test it out by running `sim$creators$create_data()`. See examples.

Value

The original simulation object with the new creator function added

Examples

```
# The first way to use add_creator is to declare a function and add it to
# your simulation object later:

sim <- new_sim()
create_data <- function (n) { rpois(n, lambda=5) }
sim %<>% add_creator(create_data)

# The second way is to do both at the same time:

sim <- new_sim()
sim %<>% add_creator("create_data", function(n) {
  rpois(n, lambda=5)
})

# With either option, you can test your function as follows:
```

```
sim$creators$create_data(10)
```

`add_method` *Add a "method" function*

Description

Add a "method" function to your simulation object. A method function is just a function, and can be used anywhere that you would normally write and use a regular global function. The advantages of explicitly adding a method function to your simulation (rather than declaring and using a function within your simulation script) are that (1) you can use the method function as a simulation level, and (2) parallelization is automated. Often, the method function will be a statistical method that you want to test (e.g. an estimator), and will take in a dataset returned by a creator function as its first argument; however, this is not always the case.

Usage

```
add_method(sim, name, fn)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by `new_sim`
`name`	A name for the method function
`fn`	A method function

Details

- As with `add_creator`, there are two ways to use `add_method`. If two arguments are supplied (`sim` and `fn`), you can create a function separately and add it to your simulation object later. If three arguments are supplied, you can do both at the same time, using an anonymous function for the `fn` argument. See examples.
- Your method will be stored in `sim$methods`. If you added a method called `estimator_1`, you can test it out by running `sim$creators$estimator_1()`. See examples.

Value

The original simulation object with the new method function added

Examples

```
sim <- new_sim()
sim %<>% add_creator("create_data", function(n) { rpois(n, lambda=5) })

# The first way to use add_method is to declare a function and add it to
# your simulation object later:

estimator_1 <- function (dat) { mean(dat) }
```

```
sim %<>% add_method(estimator_1)

# The second way is to do both at the same time:

sim %<>% add_method("estimator_2", function(dat) {
  var(dat)
})

# With either option, you can test your function as follows:

dat <- sim$creators$create_data(10)
sim$methods$estimator_1(dat)
sim$methods$estimator_2(dat)
```

`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 %<>% add_creator("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()
```

`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()
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 (`add_creator`, `set_config`, etc.) have been called.

Usage

```
run(sim, sim_uids = NA)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by `new_sim`
`sim_uids`	Advanced; a vector of `sim_uid` values, each of which uniquely identifies a simulation replicate. This will normally be omitted. If this is specified, only the simulation replicates with a matching `sim_uid` will be run.

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

```
# The following is a toy example of a simulation, illustrating the use of
# the run function.
sim <- new_sim()
sim %<>% add_creator("create_data", function(n) { rpois(n, lambda=5) })
sim %<>% add_method("estimator_1", function(dat) { mean(dat) })
sim %<>% add_method("estimator_2", function(dat) { var(dat) })
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "estimator" = c("estimator_1", "estimator_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 %>% print()
```

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

Description

This function serves a scaffold for running simulations in parallel on a cluster computing system. It acts as a wrapper for the code in your simulation script, organizing the code into three sections, labeled "first" (code that is run once at the start of the simulation, e.g. setting simulation levels), "main" (the simulation script, which is run repeatedly), and "last" (code to combine and 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. See https://avi-kenny.github.io/SimEngine/parallelization/ for an overview of how cluster parallelization works in **SimEngine**.

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 that creates a simulation object. Put everything you need in the simulation object, since global variables declared in this block will not be available when the 'main' and 'last' code blocks run.
`main`	Code that will run for every simulation replicate. This should be a block of code enclosed by curly braces that includes a call to `run`. 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 takes your simulation object (which at this point will contain your results) and do something with it, such as display your results on a graph.
`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). 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; this directory will serve as your working directory and hold your simulation object, temporary **SimEngine** objects, and simulation results (this defaults to the working directory of the R script that contains your simulation code).

Examples

```
## Not run:
# The following is a toy simulation that could be run on a cluster computing
# environment. It runs 10 replicates of 2 simulation levels as 20 separate
```

```
# cluster jobs, and then summarizes the results. This function is designed to
# be used in conjunction with cluster job scheduler software (e.g. Slurm or
# Oracle Grid Engine). We include both the R code as well as sample BASH code
# for running the simulation using Oracle Grid Engine.

# This code is saved in a file called my_simulation.R
library(SimEngine)
run_on_cluster(

  first = {
    sim <- new_sim()
    sim %<>% add_creator("create_data", function(n){ rnorm(n) })
    sim %<>% set_script(function() {
      data <- create_data(L$n)
      return(mean(data))
    })
    sim %<>% set_levels(n=c(100,1000))
    sim %<>% set_config(num_sim=10)
  },

  main = {
    sim %<>% run()
  },

  last = {
    sim %<>% summarize()
  },

  cluster_config = list(js="ge")

)

# This 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 cluster head node.
# qsub -v run='first' run_sim.sh
# qsub -v run='main' -t 1-20 -hold_jid 101 run_sim.sh
# qsub -v run='last' -hold_jid 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 = "none",
  n_cores = parallel::detectCores() - 1,
  packages = NULL,
  stop_at_error = FALSE,
  progress_bar = TRUE,
  seed = NA
)
```

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	A string; one of c("outer", "inner", "none"). Controls which sections of the code are parallelized. Setting to "outer" will run one simulation per core. Setting to "inner" will allow for parallelization within a single simulation replicate. Setting to "none" will not parallelize any code. See https://avi-kenny.github.io/SimEngine/parallelization/ for an overview of how parallelization works in **SimEngine**. This option will be ignored if the simulation is being run on a cluster computing system.
n_cores	An integer; determines the number of CPUs on which the simulation will run if using parallelization. Defaults to one fewer than the number of available CPUs on the current host.
packages	A character vector of packages to load and attach
stop_at_error	A Boolean. If set to TRUE, the simulation will stop if it encounters an error in any single replicate Useful for debugging.
progress_bar	A 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.

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
)
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, ..., .add = FALSE)
```

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.
`.add`	Only relevant if `set_levels` is called twice or more. On the second call, if add=FALSE (default) the old set of levels will be replaced by the new set, whereas if add=TRUE the new set of levels will be merged with the old set. See examples.

Value

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

Examples

```
# Basic usage is as follows:
sim <- new_sim()
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "theta" = c(2, 3)
)
sim$levels

# More complex levels can be set using lists:
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "theta" = c(2, 3),
  "method" = list(
    "spline1" = list(knots=c(2,4), slopes=c(0.1,0.4)),
    "spline2" = list(knots=c(1,5), slopes=c(0.2,0.3))
  )
)
sim$levels

# By default, set_levels will overwrite old levels if it is called twice:
sim %<>% set_levels(alpha=c(1,2), beta=c(5,6))
sim %<>% set_levels(alpha=c(3,4), gamma=c(7,8))
sim$levels

# To merge the old levels with the new levels instead, specify .add=TRUE:
sim %<>% set_levels(alpha=c(1,2), beta=c(5,6))
sim %<>% set_levels(alpha=c(3,4), gamma=c(7,8), .add=TRUE)
sim$levels
```

---

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 examples). The function body can contain references to the special objects L (simulation levels) and C (simulation constants) (see examples). The keys must be valid R names (see ?make.names).

Value

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

Examples

```
# The following is a toy example of a simulation, illustrating the use of
# the set_script function.
sim <- new_sim()
sim %<>% add_creator("create_data", function(n) { rpois(n, lambda=5) })
sim %<>% add_method("estimator_1", function(dat) { mean(dat) })
sim %<>% add_method("estimator_2", function(dat) { var(dat) })
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "estimator" = c("estimator_1", "estimator_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

# If you need to return complex result data (vectors, dataframes, lists,
# etc.), use the construct ".complex"=list().
sim <- new_sim()
sim %<>% set_levels(n=c(4,9))
sim %<>% set_config(num_sim=1)
sim %<>% set_script(function() {
  dat <- rnorm(L$n)
  mtx <- matrix(dat, nrow=sqrt(length(dat)))
  return (list(
    "mean" = mean(dat),
    "det" = det(mtx),
    ".complex" = list(dat=dat, mtx=mtx)
  ))
})
sim %<>% run()
```

Description

SimEngine is an open-source R package for structuring, maintaining, running, and debugging statistical simulations on both local and cluster-based computing environments. Emphasis is placed on thorough documentation and scalability.

Documentation

The primary documentation for **SimEngine** is housed at https://avi-kenny.github.io/SimEngine/ - we recommend that users start here when first learning how to use the package. Additionally, standard R documentation is provided, which can be accessed in the usual way (e.g. `?new_sim`).

`summarize` *Summarize simulation results*

Description

This function calculates summary statistics for simulation results. Options for summary statistics include 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, ...)
```

Arguments

`sim`	A simulation object of class `sim_obj`, usually created by `new_sim`
`...`	Name-value pairs of summary statistic functions. The possible functions (names) are listed below. The value for each summary function is a list of summaries to perform.

- `mean`: Each `mean` summary is a named list of three arguments. `name` gives a name for the summary, `x` gives the name of the variable in `sim$results` on which to calculate the mean, and `na.rm` indicates whether to exclude `NA` values when performing the calculation.
- `median`: Each `median` summary is a named list of three arguments. `name` gives a name for the summary, `x` gives the name of the variable in `sim$results` on which to calculate the median, and `na.rm` indicates whether to exclude `NA` values when performing the calculation.
- `var`: Each `var` (variance) summary is a named list of three arguments. `name` gives a name for the summary, `x` gives the name of the variable in `sim$results` on which to calculate the variance, and `na.rm` indicates whether to exclude `NA` values when performing the calculation.
- `sd`: Each `sd` (standard deviation) summary is a named list of three arguments. `name` gives a name for the summary, `x` gives the name of the variable in `sim$results` on which to calculate the standard deviation, and `na.rm` indicates whether to exclude `NA` values when performing the calculation.

- mad: Each mad (mean absolute deviation) summary is a named list of three arguments. name gives a name for the summary, x gives the name of the variable in sim$results on which to calculate the MAD, and na.rm indicates whether to exclude NA values when performing the calculation.
- iqr: Each iqr (interquartile range) summary is a named list of three arguments. name gives a name for the summary, x gives the name of the variable in sim$results on which to calculate the IQR, and na.rm indicates whether to exclude NA values when performing the calculation.
- min: Each min (minimum) summary is a named list of three arguments. name gives a name for the summary, x gives the name of the variable in sim$results on which to calculate the minimum, and na.rm indicates whether to exclude NA values when performing the calculation.
- max: Each max (maximum) summary is a named list of three arguments. name gives a name for the summary, x gives the name of the variable in sim$results on which to calculate the maximum, and na.rm indicates whether to exclude NA values when performing the calculation.
- quantile: Each quantile summary is a named list of four arguments. name gives a name for the summary, x gives the name of the variable in sim$results on which to calculate the quantile, prob is a number in [0,1] denoting the desired quantile, and na.rm indicates whether to exclude NA values when performing the calculation.
- bias: Each bias summary is a named list of four arguments. name gives a name for the summary, estimate gives the name of the variable in sim$results containing the estimator of interest, truth is the estimand of interest (see *Details*), and na.rm indicates whether to exclude NA values when performing the calculation.
- bias_pct: Each bias_pct summary is a named list of four arguments. name gives a name for the summary, estimate gives the name of the variable in sim$results containing the estimator of interest, truth is the estimand of interest (see *Details*), and na.rm indicates whether to exclude NA values when performing the calculation.
- mse: Each mse (mean squared error) summary is a named list of four arguments. name gives a name for the summary, estimate gives the name of the variable in sim$results containing the estimator of interest, truth is the estimand of interest (see *Details*), and na.rm indicates whether to exclude NA values when performing the calculation.
- mae: Each mae (mean absolute error) summary is a named list of four arguments. name gives a name for the summary, estimate gives the name of the variable in sim$results containing the estimator of interest, truth is the estimand of interest (see *Details*), and na.rm indicates whether to exclude NA values when performing the calculation.
- coverage: Each coverage (confidence interval coverage) summary is a named list of five arguments. Either (estimate, se) or (lower, upper) must be provided. name gives a name for the summary, estimate gives the name of the variable in sim$results containing the estimator of interest, se gives the name of the variable in sim$results containing the standard error of the estimator of interest, lower gives the name of the variable in

sim$results containing the confidence interval lower bound, upper gives the name of the variable in sim$results containing the confidence interval upper bound, truth is the estimand of interest, and na.rm indicates whether to exclude NA values when performing the calculation. See *Details*.

Details

- For all summaries besides coverage, the name argument is optional. If name is not provided, a name will be formed from the type of summary and the column on which the summary is performed.
- 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 simply the proportion of simulation replicates for a given level 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.

Examples

```
# The following is a toy example of a simulation, illustrating the use of
# the summarize function.
sim <- new_sim()
sim %<>% add_creator("create_data", function(n) { rpois(n, lambda=5) })
sim %<>% add_method("estimator_1", function(dat) { mean(dat) })
sim %<>% add_method("estimator_2", function(dat) { var(dat) })
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "estimator" = c("estimator_1", "estimator_2")
)
sim %<>% set_config(num_sim=5)
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 %>% summarize(
  mean = list(name="mean_lambda_hat", x="lambda_hat"),
  mse = list(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 = TRUE, keep_extra = FALSE)
```

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
keep_extra	logical (FALSE by default); if TRUE, keep previously run simulation reps even if they exceed the current num_sim in config or are from a level that has been dropped; if FALSE, drop excess reps (starting from the last rep for that particular simulation level)

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.
- In general, if num_sim has been reduced prior to running update_sim, it is best to use the default option keep_extra = FALSE. Otherwise, some simulation levels will have more replicates than others, which makes comparison difficult.

Value

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

Examples

```
sim <- new_sim()
sim %<>% add_creator("create_data", function(n) { rpois(n, lambda=5) })
sim %<>% add_method("estimator_1", function(dat) { mean(dat) })
sim %<>% add_method("estimator_2", function(dat) { var(dat) })
sim %<>% set_levels(
```

```
  "n" = c(10, 100),
  "estimator" = c("estimator_1")
)
sim %<>% set_config(num_sim=10)
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 %<>% set_levels(
  "n" = c(10, 100, 1000),
  "estimator" = c("estimator_1", "estimator_2")
)
sim %<>% set_config(num_sim=5)
sim %<>% update_sim()
```

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

Description

This function serves a scaffold for updating a previously-run in parallel on a cluster computing system. Like `run_on_cluster`, it acts as a wrapper for the code in your simulation script, organizing the code into three sections, labeled "first" (code that is run once at the start of the simulation, e.g. setting simulation levels), "main" (the simulation script, which is run repeatedly), and "last" (code to combine and 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. See https://avi-kenny.github.io/SimEngine/parallelization/ for an overview of how cluster parallelization works in **SimEngine**.

Usage

```
update_sim_on_cluster(
  first,
  main,
  last,
  cluster_config,
  keep_errors = TRUE,
  keep_extra = FALSE
)
```

Arguments

`first`	Code to run before executing additional simulation replicates. For example, this could include altering the simulation levels or changing `nsim`. This block of code, enclosed by curly braces , must first read in an existing simulation object and then make alterations to it. Global variables declared in this block will not be available when the 'main' and 'last' code blocks run.

main	Code that will run for every simulation replicate. This should be a block of code enclosed by curly braces that includes a call to update_sim. This code block will have access to the simulation object you read 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 additional simulation replicates have been run. This should be a block of code enclosed by curly braces that takes your simulation object (which at this point will contain both your old and new results) and do something with it, such as display your results on a graph.
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). 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; this directory will serve as your working directory and hold your simulation object, temporary **SimEngine** objects, and simulation results (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
keep_extra	logical (FALSE by default); if TRUE, keep previously run simulation reps even if they exceed the current num_sim in config or are from a level that has been dropped; if FALSE, drop excess reps (starting from the last rep for that particular simulation level)

Examples

```
## Not run:
# The following code creates, runs, and subsequently updates a toy simulation
# on a cluster computing environment. We include both the R code as well as
# sample BASH code for running the simulation using Oracle Grid Engine.

# This code is saved in a file called my_simulation.R
library(SimEngine)
run_on_cluster(

  first = {
    sim <- new_sim()
    sim %<>% add_creator("create_data", function(n){ rnorm(n) })
    sim %<>% set_script(function() {
      data <- create_data(L$n)
      return(mean(data))
    })
    sim %<>% set_levels(n=c(100,1000))
    sim %<>% set_config(num_sim=10)
  },

  main = {
    sim %<>% run()
  },

  last = {
```

```
    sim %<>% summarize()
  },

  cluster_config = list(js="ge")

)

# This 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 cluster head node.
# qsub -v run='first' run_sim.sh
# qsub -v run='main' -t 1-20 -hold_jid 101 run_sim.sh
# qsub -v run='last' -hold_jid 102 run_sim.sh

# This code is saved in a file called update_my_simulation.R. Note that it
# reads in the simulation object created above, which is saved in a file
# called "sim.rds".
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="ge")

)

# This code is saved in a file called update_sim.sh
# #!/bin/bash
# Rscript update_my_simulation.R

# The following lines of code are run on the cluster head node. Note that
# only 10 new replicates are run, since 20 of 30 simulation replicates were
# run in the original call to run_on_cluster.
# qsub -v run='first' update_sim.sh
# qsub -v run='main' -t 1-10 -hold_jid 104 update_sim.sh
# qsub -v run='last' -hold_jid 105 update_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 added to your simulation object via `add_method`
`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()
sim %<>% add_creator("create_data", function(n) { rpois(n, lambda=5) })
sim %<>% add_method("estimator_1", function(dat) { mean(dat) })
sim %<>% add_method("estimator_2", function(dat) { var(dat) })
sim %<>% set_levels(
  "n" = c(10, 100, 1000),
  "estimator" = c("estimator_1", "estimator_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).

Details

- You can also access simulation variables through sim$vars, where sim is your simulation object (see examples).

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)
```

```
vars(sim, "num_sim_total") %>% print()
sim$vars$num_sim_total %>% print()
vars(sim) %>% print()
```

Index