

# Package ‘SimEngine’

April 27, 2022

**Type** Package

**Title** A Modular Framework for Statistical Simulations in R

**Version** 1.1.0

**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 <avi.kenny@gmail.com>

**Repository** CRAN

**Date/Publication** 2022-04-27 11:20:02 UTC

## R topics documented:

get_complex . . . . .	2
js_support . . . . .	3
new_sim . . . . .	3
run . . . . .	4
run_on_cluster . . . . .	5
set_config . . . . .	6
set_levels . . . . .	8

set_script . . . . .	9
SimEngine . . . . .	11
summarize . . . . .	11
update_sim . . . . .	14
update_sim_on_cluster . . . . .	15
use_method . . . . .	18
vars . . . . .	19

<b>Index</b>	<b>21</b>
--------------	-----------

---

get_complex	<i>Access internal simulation variables</i>
-------------	---

---

## Description

Extract complex simulation data from a simulation object

## Usage

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

## Arguments

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

## Value

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

## Examples

```
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()
```

---

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	<i>Run the simulation</i>
-----	---------------------------

---

### 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, sim_uids = NA)
```

### Arguments

<code>sim</code>	A simulation object of class <code>sim_obj</code> , usually created by <code>new_sim</code>
<code>sim_uids</code>	Advanced; a vector of <code>sim_uid</code> 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 <code>sim_uid</code> 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()
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()
```

## 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

- |                             |   |
|-----------------------------|---|
| <code>first</code>          | Code to run at the start of a simulation. This should be a block of code enclosed by curly braces 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.   |
| <code>main</code>           | Code that will run for every simulation replicate. This should be a block of code enclosed by curly braces that includes a call to <code>run</code> . 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.  |
| <code>last</code>           | 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.  |
| <code>cluster_config</code> | A list of configuration options. You must specify either <code>js</code> (the job scheduler you are using) or <code>tid_var</code> (the name of the environment variable that your task ID is stored in). Run <code>js_support()</code> to see a list of job schedulers that are currently supported. You can optionally also specify <code>dir</code> , 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 <b>SimEngine</b> 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()
    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 sim_run='first' run_sim.sh
# qsub -v sim_run='main' -t 1-20 -hold_jid 101 run_sim.sh
# qsub -v sim_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

<code>sim</code>	A simulation object of class <code>sim_obj</code> , usually created by <code>new_sim</code>
<code>num_sim</code>	An integer; the number of simulations to conduct for each level combination
<code>parallel</code>	A string; one of <code>c("outer", "inner", "none")</code> . 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 <a href="https://avi-kenny.github.io/SimEngine/parallelization/">https://avi-kenny.github.io/SimEngine/parallelization/</a> for an overview of how parallelization works in <b>SimEngine</b> . This option will be ignored if the simulation is being run on a cluster computing system.
<code>n_cores</code>	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.
<code>packages</code>	A character vector of packages to load and attach
<code>stop_at_error</code>	A Boolean. If set to TRUE, the simulation will stop if it encounters an error in any single replicate Useful for debugging.
<code>progress_bar</code>	A Boolean. If set to FALSE, the progress bar that is normally displayed while the simulation is running is suppressed.
<code>seed</code>	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, <b>SimEngine</b> will set a random seed, which can later be retrieved using the <code>vars</code> 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	<i>Set simulation levels</i>
------------	------------------------------

---

### Description

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

### Usage

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

### Arguments

sim	A simulation object of class <code>sim_obj</code> , usually created by <code>new_sim</code>
...	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 <code>set_levels</code> is called twice or more. On the second call, if <code>add=FALSE</code> (default) the old set of levels will be replaced by the new set, whereas if <code>add=TRUE</code> the new set of levels will be merged with the old set. See examples.
.keep	An integer vector specifying which level combinations to keep; 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

# If you don't want to run simulations for all level combinations, use the
# .keep option. First, set the levels normally. Second, view the
# sim$levels_grid dataframe to examine the level combinations and the
# associated level_id values. Third, call set_levels again with the .keep
# option to specify which levels to keep (via a vector of level_id values).
sim %<>% set_levels(alpha=c(1,2,3), beta=c(5,6))
sim$levels_grid
sim %<>% set_levels(.keep=c(1,2,6))
sim$levels_grid

```

---

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 <code>sim_obj</code> , usually created by <code>new_sim</code>
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 <code>".complex"</code> and the associated value must be a list (see examples). The function body can contain references to the special object <code>L</code> that stores the current set of simulation levels (see examples). The keys must be valid R names (see <code>?make.names</code> ).

**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()
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

# 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()
```

---

 SimEngine

*SimEngine: A modular framework for statistical simulations in R*


---

## 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

- |                  |   |
|------------------|---|
| <code>sim</code> | A simulation object of class <code>sim_obj</code> , usually created by <code>new_sim</code>   |
| <code>...</code> | 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. <ul style="list-style-type: none"> <li>• <code>mean</code>: Each mean summary is a named list of three arguments. <code>name</code> gives a name for the summary, <code>x</code> gives the name of the variable in <code>sim\$results</code> on which to calculate the mean, and <code>na.rm</code> indicates whether to exclude NA values when performing the calculation.</li> <li>• <code>median</code>: Each median summary is a named list of three arguments. <code>name</code> gives a name for the summary, <code>x</code> gives the name of the variable in <code>sim\$results</code> on which to calculate the median, and <code>na.rm</code> indicates whether to exclude NA values when performing the calculation.</li> </ul> |

- `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()
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(
  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

<code>sim</code>	A simulation object of class <code>sim_obj</code> , usually created by <a href="#">new_sim</a> , that has already been run by the <a href="#">run</a> function
<code>keep_errors</code>	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
<code>keep_extra</code>	logical (FALSE by default); if TRUE, keep previously run simulation reps even if they exceed the current <code>num_sim</code> 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()
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 %<>% set_levels(n=c(10,100,1000), est=c("M", "V"))
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 <code>update_sim</code> . 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 <code>js</code> (the job scheduler you are using) or <code>tid_var</code> (the name of the environment variable that your task ID is stored in). Run <code>js_support()</code> to see a list of job schedulers that are currently supported. You can optionally also specify <code>dir</code> , 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 <b>SimEngine</b> 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 <code>num_sim</code> 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()
    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 sim_run='first' run_sim.sh
# qsub -v sim_run='main' -t 1-20 -hold_jid 101 run_sim.sh
# qsub -v sim_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 sim_run='first' update_sim.sh
# qsub -v sim_run='main' -t 1-10 -hold_jid 104 update_sim.sh
# qsub -v sim_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

<code>method</code>	A character string naming a function that has been declared or loaded via <code>source()</code> .
<code>args</code>	A list of arguments to be passed onto <code>method</code>

## 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 <code>sim_obj</code> , usually created by <code>new_sim</code>
var	If this argument is omitted, <code>vars()</code> will return a list containing all available internal variables. If this argument is provided, it should equal one of the following character strings: <ul style="list-style-type: none"><li>• <code>seed</code>: the simulation seed; see <code>set_config</code> for more info on seeds.</li><li>• <code>env</code>: a reference to the environment in which individual simulation replicates are run (advanced)</li><li>• <code>num_sim_total</code>: 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.</li><li>• <code>run_state</code>: 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).</li></ul>

### 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

`get_complex`, 2

`js_support`, 3

`new_sim`, 2, 3, 4, 7, 8, 10, 11, 14, 19

`run`, 4, 5, 7, 14

`run_on_cluster`, 5, 15

`set_config`, 6, 14, 19

`set_levels`, 8, 8, 14

`set_script`, 9

`SimEngine`, 11

`summarize`, 11

`update_sim`, 14, 16

`update_sim_on_cluster`, 15

`use_method`, 18

`vars`, 7, 19