SimEngine: A Modular Framework for Statistical Simulations in R

by Avi Kenny and Charles J. Wolock

Abstract This article describes **SimEngine**, an open-source R package for structuring, maintaining, running, and debugging statistical simulations on both local and cluster-based computing environments. Several R packages exist for facilitating simulations, but **SimEngine** is the only package specifically designed for running simulations in parallel via job schedulers on high-performance cluster computing systems. The package provides structure and functionality for common simulation tasks, such as setting simulation levels, managing seeds for random number generation, and calculating summary metrics (such as bias and confidence interval coverage). **SimEngine** also brings several unique features, such as automatic calculation of Monte Carlo error and information-sharing across simulation replicates. We provide an overview of the package and demonstrate some of its advanced functionality.

1 Introduction

For the past several decades, the design and execution of simulation studies has been a pillar of methodological research in statistics (Hauck and Anderson, 1984). Simulations are commonly used to evaluate finite sample performance of proposed statistical procedures, but can also be used to identify problems with existing methods, ensure that statistical code functions as designed, and test out new ideas in an exploratory manner. Additionally, simulation can be a statistical method in itself; two common examples are study power calculation (Arnold et al., 2011) and sampling from a complex distribution (Wakefield, 2013).

Although the power of personal computers has increased exponentially over the last four decades, many simulation studies require far more computing power than what is available on even a high-end laptop. Accordingly, many academic (bio)statistics departments and research institutions have invested in so-called cluster computing systems (CCS), which are essentially networks of servers available for high-throughput parallel computation. A single CCS typically serves many researchers, can be securely accessed remotely, and operates job scheduling software (e.g., Slurm) designed to coordinate the submission and management of computing tasks between multiple groups of users.

Thankfully, the majority of statistical simulations can be easily parallelized, since they typically involve running the same (or nearly the same) code many times and then performing an analysis of the results of these replicates. This allows for a simulation study to be done hundreds or thousands of times faster than if the user ran the same code on their laptop. Despite this, many potential users never leverage an available CCS; a major reason for this is that it can be difficult to get R and the CCS to "work together," even for an experienced programmer.

To address this gap, we created **SimEngine**, an open-source R package (R Core Team, 2021) for structuring, maintaining, running, and debugging statistical simulations on both local and cluster-based computing environments. Several R packages exist for structuring simulations (see Section 2.6); however, **SimEngine** is the only package specifically designed for running simulations both locally and in parallel on a high-performance CCS. The package provides structure and functionality for common simulation tasks, such as setting simulation levels, managing random number generator (RNG) seeds, and calculating summary metrics (such as bias and confidence interval coverage). In addition, **SimEngine** offers a number of unique features, such as automatic calculation of Monte Carlo error (Koehler et al., 2009) and information-sharing across simulation replicates.

This article is organized as follows. In Section 2.2, we outline the overarching design principles of the package. In Section 2.3, we give a broad overview of the **SimEngine** simulation workflow. In Section 2.4, we describe how simulations can be parallelized both locally and on a CCS. Section 2.5 contains details on advanced functionality of the package. The Appendix includes two example simulation studies carried out using **SimEngine**.

2 Design principles

There are four main principles that guided the design of **SimEngine**, which we refer to as (1) generality, (2) modularity, (3) parallelizability, and (4) appropriate scope. We discuss each in turn.

The first principle is *generality*. Many simulation frameworks assume that users will have a

particular type of workflow that involves a data generation step, an analysis step, and an evaluation step. This three-step workflow is common — in fact, we use it to illustrate the use of the package in the introductory documentation — but we also do not want to impose this workflow on the user and disallow other possible workflows. In **SimEngine**, instead of providing facilities for the user to specify a data-generating step, an analysis step, and an evaluation step, the user writes R code representing a single simulation replicate within a special function called the *simulation script*. This imposes virtually no constraints on what the user is able to do within their simulations, and allows for workflows that cannot be shoehorned into this three-step process, such as simulations involving resampling from existing datasets, complex data transformations, saving intermediate objects or plots, and so on. The only requirement of the simulation script is that it returns data in a specific format (to facilitate processing of results), but this format is completely general and allows for arbitrarily complex objects (matrices, dataframes, model objects, etc.) to be returned in addition to simple numeric values.

The second principle is *modularity*. In short, it should be simple and straightforward to change one component of a simulation without breaking other components, such as adding a new estimator or changing a sample size. Additionally, it should be easy to run additional simulation replicates, possibly with certain elements changed, without having to rerun the entire simulation. Finally, the step of evaluating the results of a simulation should be completely separated from the step of actually running the simulation replicates. This implies that a user does not need to decide in advance how they are going to summarize or visualize their results, and that they can calculate new summary statistics, produce new visualizations, and otherwise process their simulations results in an exploratory manner, possibly long after the simulation itself has been run. In short, we designed **SimEngine** to mirror the real-world workflow of statistical simulations, in which researchers often make small changes in response to new ideas or results.

The third principle is *parallelizability*. As mentioned in Section 2.1, a primary reason for creating **SimEngine** was to build a package that encapsulated the repetitive tasks related to both local and CCS parallelization. We have seen firsthand that many statistical researchers run time-consuming simulations serially on their laptops because the barrier to entry for parallelizing code, particularly on a CCS, is daunting. It is our hope that making parallelization as easy as possible will allow researchers to easily leverage parallelization hardware and boost productivity.

The fourth principle is *appropriate scope*. We designed **SimEngine** to *only* provide functionality related to simulations. While some other packages provide functionality to plot results, for example, we intentionally choose to not do so, since all analysts have their own preferences when it comes to displaying data. Similarly, unlike many packages, we do not provide functionality to generate certain data structures, since this would bloat the package and still only satisfy the needs of a small handful of users.

Finally, although not a software design principle per se, an additional goal of **SimEngine** was to create documentation that assumes as little statistical knowledge as possible. Different statisticians have different areas of background knowledge, and we wanted to avoid having potential users sidetracked by trying to understand the statistics of a particular example rather than understand the functions and workflow of **SimEngine**. Accordingly, we strove to create the simplest possible illustrations and examples in the package documentation. In general, we aimed to make the documentation as comprehensive and accessible as possible, while remaining concise and minimizing the barriers to entry.

3 Overview of simulation workflow and primary functions

The latest stable version of **SimEngine** can be installed from CRAN using `install.packages`. The current development version can be installed using `devtools::install_github`.

```
R> install.packages("SimEngine")
R> devtools::install_github(repo="Avi-Kenny/SimEngine")
```

The goal of many statistical simulations is to compare the behavior of two or more statistical methods; we use this framework to demonstrate the **SimEngine** workflow. Most statistical simulations of this type include three basic phases: (1) generate data, (2) run one or more methods using the generated data, and (3) compare the performance of the methods.

To briefly illustrate how these phases are implemented using **SimEngine**, we use a simple example of estimating the rate parameter λ of a Poisson(λ) distribution. To anchor the simulation in a real-world situation, one can imagine that a sample of size n from this Poisson distribution models the number of patients admitted daily to a hospital over the course of n consecutive days. Suppose that the data consist of n independent and identically distributed observations $X_1, X_2, \ldots, X_n$ drawn from a Poisson(λ) distribution. Since the λ parameter of the Poisson distribution is equal to both the mean

and the variance, one may ask whether the sample mean $\hat{\lambda}_{M,n} := \frac{1}{n}\sum_{i=1}^{n} X_i$ or the sample variance $\hat{\lambda}_{V,n} := \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \hat{\lambda}_{M,n})^2$ is a better estimator of λ.

Load the package and create a simulation object

After loading the package, the first step is to create a simulation object (an R object of class `sim_obj`) using the `new_sim` function. The simulation object contains all data, functions, and results related to the simulation.

```
R> library(SimEngine)
R> set.seed(1)
R> sim <- new_sim()
```

Code a function to generate data

Many simulations involve a function that creates a dataset designed to mimic a real-world data-generating mechanism. Here, we write and test a simple function to generate a sample of `n` observations from a Poisson distribution with $\lambda = 20$.

```
R> create_data <- function(n) {
+    return(rpois(n=n, lambda=20))
+  }
R> create_data(n=10)
 [1] 18 25 25 21 13 22 23 22 18 26
```

Code the methods (or other functions)

With **SimEngine**, any functions declared (or loaded via `source`) are automatically stored in the simulation object when the simulation runs. In this example, we test the sample mean and sample variance estimators of the λ parameter. For simplicity, we write this as a single function and use the `type` argument to specify which estimator to use.

```
R> est_lambda <- function(dat, type) {
+    if (type=="M") { return(mean(dat)) }
+    if (type=="V") { return(var(dat)) }
+  }
R> dat <- create_data(n=1000)
R> est_lambda(dat=dat, type="M")
[1] 19.646
R> est_lambda(dat=dat, type="V")
[1] 20.8195
```

Set the simulation levels

Often, we wish to run the same simulation multiple times. We refer to each run as a *simulation replicate*. We may wish to vary certain features of the simulation between replicates. In this example, perhaps we choose to vary the sample size and the estimator used to estimate λ. We refer to the features that vary as *simulation levels*; in the example below, the simulation levels are the sample size (`n`) and the estimator (`estimator`). We refer to the values that each simulation level can take on as *level values*; in the example below, the `n` level values are `10`, `100`, and `1000`, and the `estimator` level values are ``` ``M'' ``` (for "sample mean") and ``` ``V'' ``` (for "sample variance"). We also refer to a combination of level values as a *scenario*; in this example, the combination of `n=10` and ``` estimator=``M'' ``` is one of the six possible scenarios defined by the two values of `n` and the three values of `estimator`. By default, **SimEngine** runs one simulation replicate for each scenario, although the user will typically want to increase this; 1,000 or 10,000 replicates per scenario is common. An appropriate number of replicates per scenario may be informed by the desired level of Monte Carlo error; see Section 2.5.6.

```
R> sim %<>% set_levels(
+    estimator = c("M", "V"),
+    n = c(10, 100, 1000)
+  )
```

Note that we make extensive use of the pipe operators (`%>%` and `%<>%`) from the **magrittr** package (Bache and Wickham, 2022). Briefly, the operator `%>%` takes the object to the left of the operator and "pipes" it to (the first argument of) the function to the right of the operator; the operator `%<>%` does the same thing, but then assigns the result back to the variable to the left of the operator. For example, `x %>% mean()` is equivalent to `mean(x)` and `x %<>% mean()` is equivalent to `x <-mean(x)`. See the **magrittr** documentation for further detail.

Create a simulation script

The simulation script is a user-written function that assembles the pieces above (generating data, analyzing the data, and returning results) to code the flow of a single simulation replicate. Within a script, the level values for the current scenario can be referenced using the special variable `L`. For instance, in the running example, when the first simulation replicate is running, `L$estimator` will equal ``M'' and `L$n` will equal `10`. In the next replicate, `L$estimator` will equal ``M'' and `L$n` will equal `100`, and so on. The simulation script will automatically have access to any functions or objects that have been declared in the global environment.

```
R> sim %<>% set_script(function() {
+    dat <- create_data(n=L$n)
+    lambda_hat <- est_lambda(dat=dat, type=L$estimator)
+    return (list("lambda_hat"=lambda_hat))
+  })
```

The simulation script should always return a list containing one or more key-value pairs, where the keys are syntactically valid names. The values may be simple data types (numbers, character strings, or boolean values) or more complex data types (lists, dataframes, model objects, etc.); see Section 2.5.3 for how to handle complex data types. Note that in this example, the estimators could have been coded instead as two different functions and then called from within the script using the `use_method` function.

Set the simulation configuration

The `set_config` function controls options related to the entire simulation, such as the number of simulation replicates to run for each scenario and the parallelization type, if desired (see Section 2.4). Packages needed for the simulation should be specified using the `packages` argument of `set_config` (rather than using `library` or `require`). We set `num_sim` to 100, and so **SimEngine** will run a total of 600 simulation replicates (100 for each of the six scenarios).

```
R> sim %<>% set_config(
+    num_sim = 100,
+    packages = c("ggplot2", "stringr")
+  )
```

Run the simulation

All 600 replicates are run at once and results are stored in the simulation object.

```
R> sim %<>% run()
  |########################################| 100%
Done. No errors or warnings detected.
```

View and summarize results

Once the simulation replicates have finished running, the `summarize` function can be used to calculate common summary statistics, such as bias, variance, mean squared error (MSE), and confidence interval coverage.

```
R> sim %>% summarize(
+    list(stat="bias", name="bias_lambda", estimate="lambda_hat", truth=20),
+    list(stat="mse", name="mse_lambda", estimate="lambda_hat", truth=20)
+  )
  level_id estimator    n n_reps bias_lambda   mse_lambda
1        1         M   10    100 -0.17600000   1.52480000
```

```
2        2         V   10    100 -1.80166667 103.23599630
3        3         M  100    100 -0.03770000   0.19165100
4        4         V  100    100  0.22910707   7.72262714
5        5         M 1000    100  0.01285000   0.01553731
6        6         V 1000    100  0.02514744   0.92133037
```

In this example, we see that the MSE of the sample variance is much higher than that of the sample mean and that MSE decreases with increasing sample size for both estimators, as expected. From the `n_reps` column, we see that 100 replicates were successfully run for each scenario. Results for individual simulation replicates can also be directly accessed via the `sim$results` dataframe.

```
R> head(sim$results)
  sim_uid level_id rep_id estimator  n      runtime lambda_hat
1       1        1      1         M 10 0.0003290176       20.1
2       7        1      2         M 10 0.0002038479       20.5
3       8        1      3         M 10 0.0001709461       17.3
4       9        1      4         M 10 0.0001630783       20.3
5      10        1      5         M 10 0.0001599789       18.3
6      11        1      6         M 10 0.0001561642       20.4
```

Above, the `sim_uid` uniquely identifies a single simulation replicate and the `level_id` uniquely identifies a scenario (i.e., a combination of level values). The `rep_id` is unique within a given scenario and identifies the index of that replicate within the scenario. The `runtime` column shows the runtime of each replicate (in seconds).

Update a simulation

After running a simulation, a user may want to update it by adding additional level values or replicates; this can be done with the `update_sim` function. Prior to running `update_sim`, the functions `set_levels` and/or `set_config` are used to declare the updates that should be performed. For example, the following code sets the total number of replicates to 200 (i.e., adding 100 replicates to those that have already been run) for each scenario, and adds one additional level value for `n`.

```
R> sim %<>% set_config(num_sim = 200)
R> sim %<>% set_levels(
+    estimator = c("M", "V"),
+    n = c(10, 100, 1000, 10000)
+  )
```

After the levels and/or configuration are updated, `update_sim` is called.

```
R> sim %<>% update_sim()
  |######################################| 100%
Done. No errors or warnings detected.
```

Another call to `summarize` shows that the additional replicates were successfully:

```
R> sim %>% summarize(
+    list(stat="bias", name="bias_lambda", estimate="lambda_hat", truth=20),
+    list(stat="mse", name="mse_lambda", estimate="lambda_hat", truth=20)
+  )
  level_id estimator     n n_reps  bias_lambda   mse_lambda
1        1         M    10    200 -0.205500000  1.875450000
2        2         V    10    200 -1.189166667 96.913110494
3        3         M   100    200 -0.055000000  0.197541000
4        4         V   100    200  0.023244949  7.955606709
5        5         M  1000    200  0.017495000  0.017497115
6        6         V  1000    200  0.053941807  0.874700025
7        7         M 10000    200 -0.005233000  0.002096102
8        8         V 10000    200 -0.007580998  0.072997135
```

It is also possible to delete level values. However, it is not possible to add or delete *levels*, as this would require updating the simulation script after the simulation has already run, which is not allowed.

4 Parallelization

User-friendly parallelization is a hallmark of **SimEngine**. There are two modes of parallelizing code using **SimEngine**, which we refer to as *local parallelization* and *cluster parallelization*. Local parallelization refers to splitting the computational work of a simulation between multiple cores of a single computer (e.g., a multicore laptop). Cluster parallelization refers to running a simulation on a CCS using job arrays. **SimEngine** is designed to automate as much of the parallelization process as possible. We give an overview of each parallelization mode below.

Local parallelization

Local parallelization is the easiest way to parallelize code, as the entire process is handled by the package and executed on the user's computer. This mode is activated using `set_config`, as follows.

```
R> sim <- new_sim()
R> sim %<>% set_config(parallel = TRUE)
```

SimEngine handles the mechanics related to parallelization internally using the base R package **parallel** (R Core Team, 2021). If a single simulation replicate runs in a very short amount of time (e.g., less than one second), using local parallelization can actually result in an *increase* in total runtime. This is because there is a certain amount of computational overhead involved in the parallelization mechanisms inside **SimEngine**. A speed comparison can be performed by running the code twice, once with `set_config(parallel = TRUE)` and once with `set_config(parallel = FALSE)`, each followed by `sim %>% vars("total_runtime")`, to see the difference in total runtime. The exact overhead involved with local parallelization will differ between machines. A simple example of a speed comparison is given below:

```
R> for (p in c(FALSE, TRUE)) {
+    sim <- new_sim()
+    sim %<>% set_config(num_sim=20, parallel=p)
+    sim %<>% set_script(function() {
+      Sys.sleep(1)
+      return (list("x"=1))
+    })
+    sim %<>% run()
+    print(paste("Parallelizing:", p))
+    print(sim %>% vars("total_runtime"))
}
  |########################################| 100%
Done. No errors or warnings detected.

[1] "Parallelizing: FALSE"
[1] 20.42414
Done. No errors or warnings detected.

[1] "Parallelizing: TRUE"
[1] 4.41772
```

Removing the line `Sys.sleep(1)` and rerunning the code above can give a sense of the amount of overhead time incurred for a given simulation and hardware configuration. If the user's computer has `n` cores available, **SimEngine** will use `n-1` cores by default. The `n_cores` argument of `set_config` can be used to manually specify the number of cores to use, as follows.

```
R> sim %<>% set_config(n_cores = 2)
```

Cluster parallelization

Parallelizing code using a CCS is more complicated, but **SimEngine** is built to streamline this process as much as possible. A CCS is a supercomputer that consists of a number of nodes, each of which may have multiple cores. Typically, a user logs into the CCS and runs programs by submitting "jobs" to the CCS using a special program called a job scheduler. The job scheduler manages the process of running the jobs in parallel across multiple nodes and/or multiple cores. Although there are multiple ways to run code in parallel on a CCS, **SimEngine** makes use of job arrays. The main cluster parallelization

function in **SimEngine** is `run_on_cluster`. Throughout this example, we use Slurm as an example job scheduler, but an analogous workflow will apply to other job scheduling software.

To illustrate the cluster parallelization workflow, consider the simulation from Section 2.3.

```
R> sim <- new_sim()
R> create_data <- function(n) { return(rpois(n=n, lambda=20)) }
R> est_lambda <- function(dat, type) {
+    if (type=="M") { return(mean(dat)) }
+    if (type=="V") { return(var(dat)) }
+  }
R> sim %<>% set_levels(estimator = c("M","V"), n = c(10,100,1000))
R> sim %<>% set_script(function() {
+    dat <- create_data(L$n)
+    lambda_hat <- est_lambda(dat=dat, type=L$estimator)
+    return(list("lambda_hat"=lambda_hat))
+  })
R> sim %<>% set_config(num_sim=100)
R> sim %<>% run()
R> sim %>% summarize()
```

To run this code on a CCS, we simply wrap it in the `run_on_cluster` function. To use this function, we must break the code into three blocks, called `first`, `main`, and `last`. The code in the `first` block will run only once, and will set up the simulation object. When this is finished, **SimEngine** will save the simulation object in the filesystem of the CCS. The code in the `main` block will then run once for each simulation replicate, and will have access to the simulation object created in the `first` block. In most cases, the code in the `main` block will simply include a single call to `run` (or to `update_sim`, as detailed below). Finally, the code in the `last` block will run after all simulation replicates have finished running, and after **SimEngine** has automatically compiled the results into the simulation object. Use of the `run_on_cluster` function is illustrated below:

```
R> 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")
+  )
```

Note that none of the actual simulation code changed (with the exception of specifying `n_cores=20` in the `set_config` call); we simply divided the code into chunks and and placed these chunks into the appropriate block (`first`, `main`, or `last`) within `run_on_cluster`. Additionally, we specified which job scheduler to use in the `cluster_config` argument list. The command `js_support` can be run in R to see a list of supported job scheduler software; the value in the `js_code` column is the value that should be specified in the `cluster_config` argument. Unsupported job schedulers can still be used for cluster parallelization, as detailed below. Note that the `cluster_config` argument can also be used to specify which folder on the cluster should be used to store the simulation object and associated simulation files (the default is the working directory).

Next, we must give the job scheduler instructions on how to run the above code. In the following, we assume that the R code above is stored in a file called `my_simulation.R`. We also need to create a simple shell script called `run_sim.sh` with the following two lines, which will run `my_simulation.R` (we demonstrate this using BASH scripting language, but any shell scripting language may be used).

```
> #!/bin/bash
> Rscript my_simulation.R
```

If created on a local machine, the two simulation files (`my_simulation.R` and `run_sim.sh`) must be transferred to the filesystem of the CCS. Finally, we use the job scheduler to submit three jobs. The first will run the `first` code, the second will run the `main` code, and the third will run the `last` code. With Slurm, we run the following three shell commands:

```
> sbatch --export=sim_run='first' run_sim.sh
Submitted batch job 101
> sbatch --export=sim_run='main' --array=1-20 --depend=afterok:101 run_sim.sh
Submitted batch job 102
> sbatch --export=sim_run='last' --depend=afterok:102 run_sim.sh
Submitted batch job 103
```

In the first line, we submit the `run_sim.sh` script using the `sim_run=`first'` environment variable, which tells **SimEngine** to only run the code in the `first` block. After running this, Slurm returns the message `Submitted batch job 101`. The number `101` is called the "job ID" and uniquely identifies the job on the CCS.

In the second line, we submit the `run_sim.sh` script using the `sim_run=`main'` environment variable and tell Slurm to run a job array with "task IDs" 1-20. Each task corresponds to one core, and so in this case 20 cores will be used. This number should equal the `n_cores` number specified via `set_config`. **SimEngine** handles the work of dividing the simulation replicates between the cores; the only restriction is that the number of cores cannot exceed the total number of simulation replicates.

Also note that we included the option `--depend=afterok:101`, which instructs the job scheduler to wait until the first job finishes before starting the job array. (In practice, the number 101 must be replaced with whatever job ID Slurm assigned to the first job.) Once this command is submitted, the code in the `main` block will be run for each replicate. A temporary folder called `sim_results` will be created and filled with temporary objects containing data on the results and/or errors for each replicate.

In the third line, we submit the `run_sim.sh` script using the `sim_run='last'` environment variable. Again, we use `--depend=afterok:102` to ensure this code does not run until all tasks in the job array have finished. When this job runs, **SimEngine** will compile the results from the `main` block, run the code in the `last` block, save the simulation object to the filesystem, and delete the temporary `sim_results` folder and its contents. If desired, the user can leave the `last` block empty, but this third `sbatch` command should be run anyways to compile the results and save the simulation object for further analysis.

Further automating job submission

Advanced users may wish to automatically capture the job IDs so that they don't need to be entered manually; sample code showing how this can be done is shown below:

```
> jid1=$(sbatch --export=sim_run='first' run_sim.sh | sed 's/Submitted batch job //')
> jid2=$(sbatch --export=sim_run='main' --array=1-20 --depend=afterok:$jid1 run_sim.sh | sed 's/Submitted bat
> sbatch --export=sim_run='last' --depend=afterok:$jid2 run_sim.sh
Submitted batch job 103
```

While this is slightly more complicated, this code allows all three lines to be submitted simultaneously without the need to copy and paste the job IDs manually every time.

Additional cluster parallelization functionality

Running locally

The `run_on_cluster` function is programmed such that it can also be run locally. In this case, the code within the `first`, `main`, and `last` blocks will be executed in the calling environment of the `run_on_cluster` function (typically the global environment); this can be useful for testing simulations locally before sending them to a CCS.

Using unsupported job schedulers

There may be job schedulers that **SimEngine** does not natively support. If this is the case, **SimEngine** can still be used for cluster parallelization; this requires identifying the environment variable that the job scheduler uses to uniquely identify tasks within a job array. For example, Slurm uses the variable `"SLURM_ARRAY_TASK_ID"` and Grid Engine uses the variable `"SGE_TASK_ID"`. Once this variable is identified, it can be specified in the `cluster_config` block, as follows:

```
R> run_on_cluster(
+    first = {...},
+    main = {...},
+    last = {...},
+    cluster_config = list(tid_var="SLURM_ARRAY_TASK_ID")
+  )
```

Updating a simulation on a CCS

To update a simulation on a CCS, the `update_sim_on_cluster` function can be used. The workflow is similar to that of `run_on_cluster`, with several key differences. Instead of creating a new simulation object in the `first` block using `new_sim`, the existing simulation object (which would have been saved to the filesystem when `run_on_cluster` was called originally) is loaded using `readRDS`. Then, the functions `set_levels` and/or `set_config` are called to specify the desired updates (see Section 2.3.9). In the `main` block, `update_sim` is called (instead of `run`). In the `last` block, code can remain the same or change as needed. These differences are illustrated in the code below.

```
R> 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")
+  )
```

Submission of this code via a job scheduler proceeds in the same manner as described earlier for `run_on_cluster`.

5 Advanced functionality

In this section, we review the following functionality, targeting advanced users of the package:

- using the `batch` function, which allows for information to be shared across simulation replicates;
- using *complex simulation levels* in settings where simple levels (e.g., a vector of numbers) are insufficient;
- handling *complex return data* in settings where a single simulation replicate returns nested lists, dataframes, model objects, etc.;
- managing RNG seeds;
- best practices for debugging and handling errors and warnings;
- capturing Monte Carlo error using the `summarize` function.

Using the batch function to share information across simulation replicates

The `batch` 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. This may be of interest if, for example, it is computationally expensive to generate a simulated dataset.

To illustrate the use of batch using this example, we first consider the following simulation:

```
R> sim <- new_sim()
R> create_data <- function(n) { rnorm(n=n, mean=3) }
R> est_mean <- function(dat, type) {
+    if (type=="est_mean") { return(mean(dat)) }
+    if (type=="est_median") { return(median(dat)) }
+  }
R> sim %<>% set_levels(est=c("est_mean","est_median"))
R> sim %<>% set_config(num_sim=3)
R> sim %<>% set_script(function() {
+    dat <- create_data(n=100)
+    mu_hat <- est_mean(dat=dat, type=L$est)
+    return(list(
+      "mu_hat" = round(mu_hat,2),
+      "dat_1" = round(dat[1],2)
+    ))
+  })
R> sim %<>% run()
  |######################################| 100%
Done. No errors or warnings detected.
```

From the ``dat_1'' column of the results object (equal to the first element of the dat vector created in the simulation script), we see that a unique dataset was created for each simulation replicate:

```
R> sim$results[order(sim$results$rep_id),c(1:7)!=5]
  sim_uid level_id rep_id        est mu_hat dat_1
1       1        1      1   est_mean   3.05  4.09
4       2        2      1 est_median   3.06  3.23
2       3        1      2   est_mean   3.03  2.99
5       5        2      2 est_median   3.02  2.78
3       4        1      3   est_mean   2.85  1.47
6       6        2      3 est_median   3.03  2.35
```

Suppose that instead, we wish to analyze each simulated dataset using multiple methods (in this case corresponding to ``est_mean'' and ``est_median''), and repeat this procedure a total of three times. We can do this using the batch function, as follows:

```
R> sim <- new_sim()
R> create_data <- function(n) { rnorm(n=n, mean=3) }
R> est_mean <- function(dat, type) {
+    if (type=="est_mean") { return(mean(dat)) }
+    if (type=="est_median") { return(median(dat)) }
+  }
R> sim %<>% set_levels(est=c("est_mean","est_median"))
R> sim %<>% set_config(num_sim=3, batch_levels=NULL)
R> sim %<>% set_script(function() {
+    batch({
+      dat <- create_data(n=100)
+    })
+    mu_hat <- est_mean(dat=dat, type=L$est)
+    return(list(
+      "mu_hat" = round(mu_hat,2),
+      "dat_1" = round(dat[1],2)
+    ))
+  })
R> sim %<>% run()
  |######################################| 100%
Done. No errors or warnings detected.
```

In the code above, we changed two things. First, we added batch_levels=NULL to the set_config call; this will be explained below. Second, we wrapped the code line dat <-create_data(n=100) inside the batch function. Whatever code goes inside the batch function will produce the same output for all simulations in a batch.

```
R> sim$results[order(sim$results$rep_id),c(1:7)!=5]
  sim_uid level_id rep_id        est mu_hat dat_1
1       1        1      1   est_mean   3.02  2.74
4       2        2      1 est_median   3.19  2.74
2       3        1      2   est_mean   2.91  3.71
5       5        2      2 est_median   2.95  3.71
3       4        1      3   est_mean   3.10  3.52
6       6        2      3 est_median   3.01  3.52
```

In this case, from the ``dat_1'' column of the results object, we see that one dataset was created and shared by the batch corresponding to `sim_uids` 1 and 2 (likewise for `sim_uids` $\{3,5\}$ and $\{4,6\}$).

However, the situation is often more complicated. Suppose we have a simulation with multiple levels, some that correspond to creating data and some that correspond to analyzing the data. Here, the `batch_levels` argument of `set_config` plays a role. Specifically, this argument should be a character vector equal to the names of the simulation levels that are referenced (via the special variable `L`) from within a `batch` block. In the example below, the levels `n` and `mu` are used within the `batch` call, while the level `est` is not.

```
R> sim <- new_sim()
R> create_data <- function(n, mu) { rnorm(n=n, mean=mu) }
R> est_mean <- function(dat, type) {
+    if (type=="est_mean") { return(mean(dat)) }
+    if (type=="est_median") { return(median(dat)) }
+  }
R> sim %<>% set_levels(n=c(10,100), mu=c(3,5), est=c("est_mean","est_median"))
R> sim %<>% set_config(num_sim=2, batch_levels=c("n", "mu"), return_batch_id=T)
R> sim %<>% set_script(function() {
+    batch({
+      dat <- create_data(n=L$n, mu=L$mu)
+    })
+    mu_hat <- est_mean(dat=dat, type=L$est)
+    return(list(
+      "mu_hat" = round(mu_hat,2),
+      "dat_1" = round(dat[1],2)
+    ))
+  })
R> sim %<>% run()
  |######################################| 100%
Done. No errors or warnings detected.
R> sim$results[order(sim$results$batch_id),c(1:10)!=8]
   sim_uid level_id rep_id batch_id   n mu        est mu_hat dat_1
1        1        1      1        1  10  3   est_mean   2.87  4.29
9        5        5      1        1  10  3 est_median   2.94  4.29
2        9        1      2        2  10  3   est_mean   2.79  2.77
10      13        5      2        2  10  3 est_median   2.73  2.77
3        2        2      1        3 100  3   est_mean   2.93  1.77
11       6        6      1        3 100  3 est_median   3.01  1.77
4       10        2      2        4 100  3   est_mean   2.80  4.44
12      14        6      2        4 100  3 est_median   2.71  4.44
5        3        3      1        5  10  5   est_mean   5.49  4.78
13       7        7      1        5  10  5 est_median   5.25  4.78
6       11        3      2        6  10  5   est_mean   4.57  4.48
14      15        7      2        6  10  5 est_median   4.62  4.48
7        4        4      1        7 100  5   est_mean   4.98  5.66
15       8        8      1        7 100  5 est_median   4.95  5.66
8       12        4      2        8 100  5   est_mean   5.08  5.55
16      16        8      2        8 100  5 est_median   5.14  5.55
```

The batches were created such that each batch contained two replicates, one for each level value of `est`. For expository purposes, we also specified the `return_batch_id=T` option in `set_config` so that the results object would return the `batch_id`. This is not necessary in practice. The `batch_id` variable defines the batches; all simulations that share the same `batch_id` are in a single batch. The `return_batch_id=T` option can be useful to ensure correct usage of the `batch` function.

We note the following about the `batch` function:

- The code within the `batch` code block must *only* create objects; this code should not change or delete existing objects, as these changes will be ignored.
- In the majority of cases, the `batch` function will be called just once, at the beginning of the simulation script. However, it can be used anywhere in the script and can be called multiple times. The `batch` function should never be used outside of the simulation script.
- Although we have illustrated the use of the `batch` function to create a dataset to share between multiple simulation replicates, it can be used for much more, such as taking a sample from an existing dataset or computing shared nuisance function estimators.
- If the simulation is being run in parallel (either locally or on a CCS), `n_cores` cannot exceed the number of batches, since all simulations within a batch must run on the same core.
- If the simulation script uses the `batch` function, the simulation cannot be updated using the `update_sim` or `update_sim_on_cluster` functions, with the exception of updates that only entail removing simulation replicates.

Complex simulation levels

Often, simulation levels are simple, such as a vector of sample sizes:

```
R> sim <- new_sim()
R> sim %<>% set_levels(n = c(200,400,800))
```

However, there are many instances in which more complex objects are needed. For these cases, instead of a vector of numbers or character strings, a named list of lists can be used. The toy example below illustrates this.

```
R> sim <- new_sim()
R> sim %<>% set_levels(
+    n = c(10,100),
+    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))
+    )
+  )
R> create_data <- function(n, type, params) {
+    if (type=="Beta") {
+      return(rbeta(n, shape1=params[1], shape2=params[2]))
+    } else if (type=="Normal") {
+      return(rnorm(n, mean=params[1], sd=params[2]))
+    }
+  }
R> sim %<>% set_script(function() {
+    x <- create_data(L$n, L$distribution$type, L$distribution$params)
+    return(list("y"=mean(x)))
+  })
R> sim %<>% run()
  |########################################| 100%
Done. No errors or warnings detected.
```

Note that the list names (``Beta 1'', ``Beta 2'', and ``Normal'') become the entries in the `sim$results` dataframe, as well as the dataframe returned by `summarize`.

```
R> sim %>% summarize(list(stat="mean", x="y"))
  level_id   n distribution n_reps    mean_y
1        1  10       Beta 1      1 0.1635174
2        2 100       Beta 1      1 0.2740965
3        3  10       Beta 2      1 0.6234866
4        4 100       Beta 2      1 0.7664522
5        5  10       Normal      1 3.0903062
6        6 100       Normal      1 3.0179944
```

Complex return data

In most situations, the results of simulations are numeric. However, we may want to return more complex data, such as matrices, lists, or model objects. To do this, we add a key-value pair to the list returned by the simulation script with the special key ".complex" and a list (containing the complex data) as the value. This is illustrated in the toy example below. Here, the simulation script estimates the parameters of a linear regression and returns these as numeric, but also returns the estimated covariance matrix and the entire model object.

```
R> sim <- new_sim()
R> sim %<>% set_levels(n=c(10, 100, 1000))
R> create_data <- function(n) {
+    x <- runif(n)
+    y <- 3 + 2*x + rnorm(n)
+    return(data.frame("x"=x, "y"=y))
+  }
R> sim %<>% set_config(num_sim=2)
R> 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)
+      )
+    ))
+  })
R> sim %<>% run()
  |########################################| 100%
Done. No errors or warnings detected.
```

After running this simulation, the numeric results can be accessed directly via `sim$results` or using the `summarize` function, as usual:

```
R> head(sim$results)
  sim_uid level_id rep_id    n     runtime beta0_hat beta1_hat
1       1        1      1   10 0.012123823  2.113012  3.780972
2       4        1      2   10 0.001345873  2.058610  3.726216
3       2        2      1  100 0.006520033  2.784147  2.058041
4       5        2      2  100 0.001568794  3.066045  2.097569
5       3        3      1 1000 0.001888037  3.144964  1.783498
6       6        3      2 1000 0.002427101  3.125128  1.714405
```

To examine the complex return data, we can use the special function `get_complex`, as illustrated below:

```
R> c5 <- get_complex(sim, sim_uid=5)
R> print(summary(c5$model))
Call:
lm(formula = y ~ x, data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-2.7050 -0.7429  0.1183  0.7470  1.9673

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.0660     0.2127  14.413  < 2e-16 ***
x             2.0976     0.3714   5.647 1.59e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.003 on 98 degrees of freedom
```

```
Multiple R-squared:  0.2455,        Adjusted R-squared:  0.2378
F-statistic: 31.89 on 1 and 98 DF,  p-value: 1.593e-07
R> print(c5$cov_mtx)
            (Intercept)           x
(Intercept)  0.04525148 -0.06968649
x           -0.06968649  0.13795995
```

Random number generator seeds

In statistical research, it is desirable to be able to reproduce the exact results of a simulation study. Since R code often involves stochastic (random) functions like `rnorm` or `sample` that return different values when called multiple times, reproducibility is not guaranteed. In a simple R script, calling the `set.seed` function at the beginning of the script ensures that the stochastic functions that follow will produce the same results whenever the script is run. However, a more nuanced strategy is needed when running simulations. When running 100 replicates of the same simulation, we do not want each replicate to return identical results; rather, we would like for each replicate to be different from one another, but for *the entire set of replicates* to be the same when the entire simulation is run twice in a row. **SimEngine** manages this process, even when simulations are being run in parallel locally or on a cluster computing system. In **SimEngine**, a single "global seed" is used to generate a different seed for each simulation replicate. The `set_config` function is used to set or change this global seed:

```
R> sim %<>% set_config(seed=123)
```

If a seed is not set using `set_config`, **SimEngine** will set a random seed automatically so that the results can be replicated if desired. To view this seed, we use the `vars` function:

```
R> sim <- new_sim()
R> print(vars(sim, "seed"))
[1] 287577520
```

Debugging and error/warning handling

In the simulation coding workflow, errors are inevitable. Some errors may affect all simulation replicates, while other errors may only affect a subset of replicates. By default, when a simulation is run, **SimEngine** will not stop if an error occurs; instead, errors are logged and stored in a dataframe along with information about the simulation replicates that resulted in those errors. Examining this dataframe by typing `print(sim$errors)` can sometimes help to quickly pinpoint the issue. This is demonstrated below:

```
R> sim <- new_sim()
R> sim %<>% set_config(num_sim=2)
R> sim %<>% set_levels(
+    Sigma = list(
+      s1 = list(mtx=matrix(c(3,1,1,2), nrow=2)),
+      s3 = list(mtx=matrix(c(4,3,3,9), nrow=2)),
+      s2 = list(mtx=matrix(c(1,2,2,1), nrow=2)),
+      s4 = list(mtx=matrix(c(8,2,2,6), nrow=2))
+    )
+  )
R> sim %<>% set_script(function() {
+    x <- MASS::mvrnorm(n=1, mu=c(0,0), Sigma=L$Sigma$mtx)
+    return(list(x1=x[1], x2=x[2]))
+  })
R> sim %<>% run()
  |########################################| 100%
Done. Errors detected in 25% of simulation replicates. Warnings detected in
0% of simulation replicates.
R> print(sim$errors)
  sim_uid level_id rep_id Sigma      runtime                          message
1       5        3      1    s2 0.0004692078 'Sigma' is not positive definite
2       6        3      2    s2 0.0006608963 'Sigma' is not positive definite
                                                                        call
1 MASS::mvrnorm(n = 1, mu = c(0, 0), Sigma = L$Sigma$mtx)
2 MASS::mvrnorm(n = 1, mu = c(0, 0), Sigma = L$Sigma$mtx)
```

From the output above, we see that the code fails for the simulation replicates that use the level with `Sigma="s2"` because it uses an invalid (not positive definite) covariance matrix. Similarly, if a simulation involves replicates that throw warnings, all warnings are logged and stored in the dataframe `sim$warnings`. If an error occurs for a subset of simulation replicates and that error is fixed, it is possible to rerun the replicates that had errors, as follows:

```
R> sim %<>% update_sim(keep_errors=FALSE)
```

The workflow demonstrated above can be useful to pinpoint errors, but it has two main drawbacks. First, it is undesirable to run a time-consuming simulation involving hundreds or thousands of replicates, only to find at the end that every replicate failed because of a typo. It may therefore useful to stop an entire simulation after a single error has occurred. Second, it can sometimes be difficult to determine exactly what caused an error without making use of more advanced debugging tools. For both of these situations, **SimEngine** includes the following configuration option:

```
R> sim %<>% set_config(stop_at_error=TRUE)
```

Setting `stop_at_error=TRUE` will stop the simulation when it encounters any error. Furthermore, the error will be thrown by R in the usual way, so if the simulation is being run in in RStudio, the built-in debugging tools (such as "Show Traceback" and "Rerun with debug") can be used to find and fix the bug. Placing a call to `browser` at the top of the simulation script can also be useful for debugging.

Monte Carlo error

Statistical simulations are often based on the principle of Monte Carlo approximation; specifically, pseudo-random sampling is used to evaluate the performance of a statistical procedure under a particular data-generating process. The performance of the procedure can be viewed as a statistical parameter and, due to the fact that only a finite number of simulation replicates can be performed, there is uncertainty in any estimate of performance. This uncertainty is often referred to as *Monte Carlo error* (see, e.g., Lee and Young, 1999). We can quantify Monte Carlo error using, for example, the standard error of the performance estimator.

Measuring and reporting Monte Carlo error is a vital component of a simulation study. **SimEngine** includes an option in the `summarize` function to automatically estimate the Monte Carlo standard error for any inferential summary statistic, e.g., estimator bias or confidence interval coverage. The standard error estimates are based on the formulas provided in Morris et al. (2019). If the option `mc_se` is set to `TRUE`, estimates of Monte Carlo standard error will be included in the summary data frame, along with associated 95% confidence intervals based on a normal approximation.

```
R> sim <- new_sim()
R> create_data <- function(n) { rpois(n, lambda=5) }
R> est_mean <- function(dat) {
+    return(mean(dat))
+  }
R> sim %<>% set_levels(n=c(10,100,1000))
R> sim %<>% set_config(num_sim=5)
R> sim %<>% set_script(function() {
+    dat <- create_data(L$n)
+    lambda_hat <- est_mean(dat=dat)
+    return (list("lambda_hat"=lambda_hat))
+  })
R> sim %<>% run()
  |######################################| 100%
Done. No errors or warnings detected.
R> sim %>% summarize(
+    list(stat="mse", name="lambda_mse", estimate="lambda_hat", truth=5),
+    mc_se = TRUE
+  )
  level_id    n n_reps lambda_mse lambda_mse_mc_se lambda_mse_mc_ci_l
1        1   10      5  0.5020000      0.274178774      -0.0353903966
2        2  100      5  0.0142800      0.012105759      -0.0094472876
3        3 1000      5  0.0031878      0.001919004      -0.0005734471
  lambda_mse_mc_ci_u
1        1.039390397
```

```
2       0.038007288
3       0.006949047
```

6 Discussion

In this article, we described **SimEngine**, an open-source R package for structuring, maintaining, running, and debugging statistical simulations. We reviewed the overall guiding design principles of the package, illustrated its workflow and main functions, highlighted local and CCS-based parallelization capabilities, and demonstrated advanced functionality. It is our hope that this article publicizes the existence of the package and leads to an increase in the size and engagement of the active user community.

SimEngine is one of several R packages aimed at users conducting statistical simulations but is, to our knowledge, the only package explicitly intended for use in a CCS environment. The **simulator** package (Bien, 2016) provides a modular and flexible framework to structure simulations but contains no functionality designed specifically for debugging simulations or leveraging a CCS. The **simpr** package (Brown and Bye, 2023) uses tidy principles but does not emphasize flexibility, with a relatively limited scope. Likewise, **simFrame** (Alfons et al., 2010), an object-oriented simulation package, makes available a suite of relatively specific simulation tools, which are intended primarily for survey statistics. Both **SimDesign** (Chalmers and Adkins, 2020) and **simsalapar** (Hofert and Mächler, 2013) are designed for ease of use but are somewhat less modular than other packages, with a single function for generating simulated data, running analyses, and summarizing results. The recent **simChef** package (Duncan et al., 2024) uses a tidy grammar framework and has additional functionality for preparing reports on the results of a simulation study. Additionally, there are a host of packages available that aid in simulating particular data structures, such as **riskSimul** (Hormann and Basoglu, 2023), **GlmSimulatoR** (McMahan, 2023), and **PhenotypeSimulator** (Meyer and Birney, 2018), which can be used in conjunction with a package for structuring simulations if applicable.

We chose to write this package specifically for the R programming language since this language is widely used by statistical methods developers; as evidence of this claim, a simple review of the top ten studies from a simple Google Scholar search of the term "simulation study" (restricted to the last five years) shows that, out of the eight studies that stated the programming language used, six were coded using R (Belias et al., 2019; Edelsbrunner et al., 2023; Todorov et al., 2020; Rusticus and Lovato, 2019; Manolov, 2019; Bower et al., 2021; Hamza et al., 2021; Thompson et al., 2021). That being said, it could be of practical value in the future to port the package to other programming environments.

Moving forward, future development of the package will be driven mainly by requests from active users, screening and prioritizing potential additions or modifications based on of the design principles articulated in Section 2.2. In particular, we hope to focus on improving parallelization capabilities, including expansion of SimEngine to other parallelization platforms (e.g., Apache Spark), support for futures/promises (as in the **future** package), and further automation of the cluster parallelization process (possibly by having **SimEngine** automatically create and submit `sbatch` commands, as is done in the **rslurm** package). Users can navigate to `https://github.com/Avi-Kenny/SimEngine/issues` to submit feature requests and view current open issues. Additionally, because SimEngine is built using the R S3 class system, it is extensible and allows for users to write additional methods customized to their particular needs, workflow, and style, such as functionality for plotting simulation results.

Computational details

The results in this paper were obtained using R 4.3.2 with the **SimEngine** 1.4.0 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/`.

Bibliography

A. Alfons, M. Templ, and P. Filzmoser. An object-oriented framework for statistical simulation: The r package simframe. *Journal of Statistical Software*, 37:1–36, 2010. [p16]

B. F. Arnold, D. R. Hogan, J. M. Colford, and A. E. Hubbard. Simulation methods to estimate design power: an overview for applied research. *BMC medical research methodology*, 11(1):1–10, 2011. [p1]

S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2022. URL `https://CRAN.R-project.org/package=magrittr`. R package version 2.0.3. [p4]

M. Belias, M. M. Rovers, J. B. Reitsma, T. P. Debray, and J. IntHout. Statistical approaches to identify subgroups in meta-analysis of individual participant data: a simulation study. *BMC medical research methodology*, 19:1–13, 2019. [p16]

J. Bien. The simulator: an engine to streamline simulations. *arXiv preprint arXiv:1607.00021*, 2016. [p16]

H. Bower, M. J. Crowther, M. J. Rutherford, T. M.-L. Andersson, M. Clements, X.-R. Liu, P. W. Dickman, and P. C. Lambert. Capturing simple and complex time-dependent effects using flexible parametric survival models: A simulation study. *Communications in Statistics-Simulation and Computation*, 50 (11):3777–3793, 2021. [p16]

E. Brown and J. Bye. *simpr: Flexible 'Tidyverse'-Friendly Simulations*, 2023. URL https://CRAN.R-project.org/package=simpr. R package version 0.2.6. [p16]

R. P. Chalmers and M. C. Adkins. Writing effective and reliable monte carlo simulations with the simdesign package. *The Quantitative Methods for Psychology*, 16(4):248–280, 2020. [p16]

J. Duncan, T. Tang, C. F. Elliott, P. Boileau, and B. Yu. simchef: High-quality data science simulations in r. *Journal of Open Source Software*, 9(95):6156, 2024. [p16]

P. A. Edelsbrunner, M. Flaig, and M. Schneider. A simulation study on latent transition analysis for examining profiles and trajectories in education: Recommendations for fit statistics. *Journal of Research on Educational Effectiveness*, 16(2):350–375, 2023. [p16]

T. Hamza, A. Cipriani, T. A. Furukawa, M. Egger, N. Orsini, and G. Salanti. A bayesian dose–response meta-analysis model: A simulations study and application. *Statistical methods in medical research*, 30 (5):1358–1372, 2021. [p16]

W. W. Hauck and S. Anderson. A survey regarding the reporting of simulation studies. *The American Statistician*, 38(3):214–216, 1984. [p1]

M. Hofert and M. Mächler. Parallel and other simulations in r made easy: An end-to-end study. *arXiv preprint arXiv:1309.4402*, 2013. [p16]

W. Hormann and I. Basoglu. *riskSimul: Risk Quantification for Stock Portfolios under the T-Copula Model*, 2023. URL https://CRAN.R-project.org/package=riskSimul. R package version 0.1.2. [p16]

E. Koehler, E. Brown, and S. J.-P. Haneuse. On the assessment of monte carlo error in simulation-based statistical analyses. *The American Statistician*, 63(2):155–162, 2009. [p1]

S. M. Lee and G. A. Young. The effect of monte carlo approximation on coverage error of double-bootstrap confidence intervals. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 61(2):353–366, 1999. [p15]

R. Manolov. A simulation study on two analytical techniques for alternating treatments designs. *Behavior Modification*, 43(4):544–563, 2019. [p16]

G. McMahan. *GlmSimulatoR: Creates Ideal Data for Generalized Linear Models*, 2023. URL https://CRAN.R-project.org/package=GlmSimulatoR. R package version 1.0.0. [p16]

H. V. Meyer and E. Birney. Phenotypesimulator: A comprehensive framework for simulating multi-trait, multi-locus genotype to phenotype relationships. *Bioinformatics*, 34(17):2951–2956, 2018. [p16]

T. P. Morris, I. R. White, and M. J. Crowther. Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11):2074–2102, 2019. [p15]

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021. URL https://www.R-project.org/. [p1, 6]

S. A. Rusticus and C. Y. Lovato. Impact of sample size and variability on the power and type i error rates of equivalence tests: A simulation study. *Practical Assessment, Research, and Evaluation*, 19(1):11, 2019. [p16]

J. Thompson, K. Hemming, A. Forbes, K. Fielding, and R. Hayes. Comparison of small-sample standard-error corrections for generalised estimating equations in stepped wedge cluster randomised trials with a binary outcome: a simulation study. *Statistical methods in medical research*, 30 (2):425–439, 2021. [p16]

H. Todorov, E. Searle-White, and S. Gerber. Applying univariate vs. multivariate statistics to investigate therapeutic efficacy in (pre) clinical trials: A monte carlo simulation study on the example of a controlled preclinical neurotrauma trial. *PLoS One*, 15(3):e0230798, 2020. [p16]

J. Wakefield. *Bayesian and Frequentist Regression Methods*, volume 23. Springer, 2013. [p1]

Avi Kenny
Department of Biostatistics and Bioinformatics, Duke University
Global Health Institute, Duke University
2424 Erwin Rd
Durham, NC 27710, USA
ORCiD: 0000-0002-9465-7307
avi.kenny@duke.edu

Charles J. Wolock
Department of Biostatistics, Epidemiology and Informatics
University of Pennsylvania
423 Guardian Drive
Philadelphia, PA 19104, USA
ORCiD: 0000-0003-3527-1102
cwolock@upenn.edu