2020 ■ Vol. 16 ■ no. 4

# Writing effective and reliable Monte Carlo simulations with the SimDesign package

R. Philip Chalmers [a] and Mark C. Adkins [a]

[a]Department of Psychology, York University

**Abstract** ■ The purpose of this tutorial is to discuss and demonstrate how to write safe, effective, and intuitive computer code for Monte Carlo simulation experiments containing one or more simulation factors. Throughout this tutorial the `SimDesign` package (Chalmers, 2020), available within the R programming environment, will be adopted due to its ability to accommodate a number of desirable execution features. The article begins by discussing a selection of attractive coding strategies that should be present in Monte Carlo simulation experiments, showcases how the `SimDesign` package can satisfy many of these desirable strategies, and provides a worked mediation analysis simulation example to demonstrate the implementation of these features. To demonstrate how the package can be used for real-world experiments, the simulation explored by Flora and Curran (2004) pertaining to a confirmatory factor analysis robustness study with ordinal response data is also presented and discussed.

**Keywords** ■ Monte Carlo simulation, simulation experiments. **Tools** ■ R package.

rphilip.chalmers@gmail.com

10.20982/tqmp.16.4.p248

**Acting Editor** ■ Denis Cousineau (Université d'Ottawa)

**Reviewers**
■ Kevin Hallgren (UW Medical Center)
■ Gellen-Kamel Alexandre (Université du Québec à Montréal)

## Introduction

Monte Carlo simulation (MCS) experiments represent a set of computer-driven, stochastic sampling-based methods that researchers can utilize to approximate difficult to track mathematical and statistical modeling problems. At their core, MCSs involve the generation and analysis of synthetic variables by means of one or more computers that have been designed to execute a set of user-defined instructions (i.e., code). In many cases, particularly when studying mathematical and statistical models, synthetic variables are constructed by way of drawing random samples from one or more probability distributions of interest. Ultimately, the goal behind MCSs is to independently repeat a computational experiment many times, collect the results of the mathematical models or statistical analyses, summarise these results, and interpret the summaries of the results to draw approximately asymptotic conclusions about the behaviour of the analyses under investigation.

The execution of simulation experiments can often be run on personal (i.e., local) computers, though for more intensive simulation experiments the use of remote computing resources (commonly referred to as "super" or "cluster" computing) is an effective strategy to evaluate thousands of simulation replications across different "nodes". However, optimally organizing MCS code that is amenable to this type of parallel computing architecture, among other types of necessary features (e.g., saving files to the hard-drive), is entirely the responsibility of the investigator. As such, if the investigator does not carefully plan the structure of their code early in the development process then inefficient, error-prone, inflexible, and multiple trial-and-error coding attempts (with a significant amount of effort dedicated to debugging) will often be required. Unfortunately, in order for investigators to avoid coding issues when constructing their computer simulation they must have obtained a great deal of technical mastery of their select programming language prior to writing their MCS experiment.

As Burton, Altman, Royston, and Holder (2006) have highlighted, writing high-quality simulation experiments that reflect real-world phenomenon is not a simple process in that it involves careful prior considerations of all design aspects — including the coding strategies and design. To help guide the process of writing effective MCSs code, as well as to ensure the written code is "optimal" in some sense, a number of practical design features ought to be included at the outset to ensure that a simulation runs smoothly throughout the development process, and so that the likelihood of writing problematic code can be minimized as early as possible. Although it is difficult to list all the desideratum that investigators should wish to have in their simulation experiment, the following contains a number of code-based features that are often desirable:

- *Intuitive to read, write, and debug*. If not properly organized early in the planning stages of the development process, MCS code can quickly become unwieldy. This makes distributing the written code to others problematic in that the implementation may be difficult to cognitively parse. Most importantly though, unorganized code causes difficulties in detecting execution mistakes, and often makes debugging an arduous rather than straightforward task.
- *Flexible and extensible*. Selecting sub-optimal approaches to coding, such as applying nested for-loops or while-loops, can make the execution flow inflexible if and when code needs to be edited at a later time (e.g., adding or removing simulation factors and conditions). Such loop or "go-to" style approaches also create issues regarding the definition of suitable storage containers, and require the use of temporary variables that represent indexed elements or locations of the object being looped over.
- *Computationally efficient*. It is desirable to have computer code that: is executed as quickly as possible to make optimal use of all available processors; requires minimal random access memory (RAM) demands; and avoids using excess hard-drive space when storing simulation results. Hence, MCS code should be designed with optimal functional execution in mind, automatically support parallel processing in the form of local and remote computing, and be organized to avoid allocating excess computer storage resources in the form of RAM or hard-drive space.
- *Reproducible at the macro and micro level*. In principle, the entire simulation, as well as any given combination of the simulation factors investigated, should be reproducible on demand; this is termed *macro* replication. This is useful, for instance, to ensure that the complete simulation study is reproducible by independent investigators. Additionally, any given Monte Carlo simulation replicate should also be reproducible, should the need arise, which is termed *micro* replication. Micro replication is particularly important for debugging purposes so that code or execution errors can be tracked down and replicated with minimal effort.
- *Safe and reliable*. Finally, simulation code should: avoid hard-to-spot errors related to storing simulation results and writing files to hard-disks; avoid infinite loops in situations where redrawing simulated data is required; track and record warnings and error messages so that these are easy to reproduce at a later time; utilize meta-analysis functions (e.g., bias, root mean-square errors, etc) that are pre-written and well tested to avoid common calculation errors; and so on. Ensuring safety and reliability of MCS code is perhaps the most cognitively and technically demanding aspect of MCS coding because it requires intimate familiarity and foresight of the select programming language and MCS design.

Unfortunately, and as Sigal and Chalmers (2016) have noted, the process of *evaluating* the simulation experiment is what is of interest to investigators, not specific coding-related issues pertaining to organization, debugging, object structures, formatting, computer storage, and so on. Hence, the task of programming feature-rich and safe simulation code is largely a secondary concern to the investigator, despite the clear importance of writing defensive and readable code. Ideally then, such features should be automated as best as possible to (at least initially) release the investigator from several responsibilities so that their cognitive efforts can be more efficiently spent on writing the design and logic of the MCS experiment itself. For a recent example that demonstrates the necessity for first mastering the R programming language, as well as the manual implementation of a small selection of the desired features listed above, see the tutorial written by Lee, Sriutaisuk, and Kim (2019) which adopts the `tidyverse` (Wickham et al., 2019) workflow for writing MCSs. Note that in their presentation, features such as parallel processing, debugging, micro and macro reproducibility, storage and RAM considerations, use of predefined functions for meta-statistics, and many other important and desirable features are not included; evidently, even within state-of-the-art programming frameworks the burden of writing an optimal simulation workflow is still unfortunately left to the investigator to implement.

The purpose of this tutorial is to demonstrate how to write more optimal code for MCS experiments. To do so, the structure utilized by the R package `SimDesign` (Chalmers, 2020) is presented to demonstrate how optimal principles and coding practices can be included in all defined simulation studies without requiring the investigator to explicitly program these important features themselves.[1] Following the discussion of MCSs and the `SimDesign` package, a real-world simula-

[1]While the R programming language is used for presentation purposes, the coding aspects and practical recommendations largely translate to other general purpose programming languages that can be used for MCSs as well.

tion experiment is presented and discussed. This simulation experiment pertains to a robustness study for confirmatory factor analyses with ordinal response data explored by Flora and Curran (2004). This specific simulation study is included to provide first-hand recommendations when designing more complex simulation code. Throughout the presentation some familiarity with the R programming language is assumed, particularly with respect to implementing user-defined R functions and debugging. Readers who are less familiar with utilizing R for constructing Monte Carlo simulations should refer to Hallgren (2013), Jones, Maillardet, and Robinson (2014), and Sigal and Chalmers (2016). Readers less familiar with writing and debugging R functions should refer to the relevant chapters in Jones et al. (2014) and Wickham (2019).

## Optimal Coding Practices When Writing Monte Carlo Simulations

While there is an abundance of literature exploring the topic of Monte Carlo simulation experiments (e.g., Burton et al., 2006; Mooney, 1997; Paxton, Curran, Bollen, Kirby, & Chen, 2001), very few recommendations regarding optimal coding practices and strategies have been suggested. Most recently, Morris, White, and Crowther (2019) provided a small number of coding recommendations to adopt when designing MCSs; namely, start with very few replications in the early stages of coding, store the computer seeds for macro reproducibility, begin with small coding tasks, catch and record errors (which in R is achieved via the `try()` family of functions) and treat these errors as "missing values", and if more than one software program is required for analyses then a single general-purpose software package should be selected for generating the data and collecting the results.

In addition to Morris et al. (2019)'s general recommendations, there are a number of other features that should be considered that are related to the desideratum presented above. Specifically, in addition to saving all potential seeds for macro replication it is desirable to actively record seeds that are *specifically* problematic during execution of the code. As an example, if a simulation replication were to throw an error message (or less severely a *warning* message) then recording how often these errors occur, as well as which specific seeds caused these errors, should be readily available to the investigator to help gauge the severity of the issues and to help track down, reproduce, and debug the causes of these errors. In this situation, macro reproducibility alone would be highly inefficient because several non-problematic replication instances would needlessly be re-executed until the problematic replication appeared in the sequence of random seeds — which, for more intensive simulations, can be particularly time consuming.

Regarding the coding structure of MCSs in R specifically, Sigal and Chalmers (2016) have highlighted why various MCS coding strategies should be avoided, particularly with respect to nested loops (e.g., `while`, `for`, and `repeat` loops). In addition to executing more slowly in interpretive languages (such as R), nested loops often cause conceptual and practical organizational issues when extracting and storing components of numerical experiments. For instance, identical results could be stored into 1) a three-dimensional `array`, indexed via `obj[i,j,k]`, 2) a `list` containing matrices, indexed with `obj[[i]][j,k]`, 3) a one-dimensional `vector`, indexed via `obj[i*j+k]`, and so on. Clearly, specific object indices are more difficult to track and debug because locating the exact index state of a nested-loop can be tedious, object assignment is more difficult to parallelize, conditional statements (if-then-else) are harder to setup for simulation condition combinations that should not be evaluated, objects must be completely re-designed if more simulation conditions need to be evaluated (e.g., a nested `list` may be required instead, indexed via `obj[[i]][[j]][k,l]`), and so on.

To avoid using a for-loop strategy and non-standardized object containers, Sigal and Chalmers (2016) recommend replacing nested for-loop combinations involving the simulation factors by organizing all experimental factors and combinations into `data.frame`-type objects, where the factors are organized in each column and unique combinations of the factors in each respective row. After constructing this object the entire simulation can then be executed by applying a single for-loop (or `apply()` function) over the respective row elements.[2] The advantage of this organizational structure is that selecting particular simulation combinations is notably more straightforward, redundant or irrelevant factor combinations may be excluded row-wise if they are not required (creating "partially-crossed" simulation designs), execution times are typically lower than using nested for-loop strategies (see below), and the organization of the output object(s) will often be more consistent and easier to manipulate. This particular structure reflects a cornerstone component of the `SimDesign` package (Chalmers, 2020) known as the *design* component, which is discussed in greater detail in the following sections.

### *Additional recommendations applicable within the R environment*

To help detect potential problems in MCSs, Morris et al. (2019) suggested using various diagnostic techniques to use

[2]Lee et al. (2019) have also recently adopted this single object simulation conditions structural approach when writing MCSs.

early in the writing process. For instance, when drawing data from select probability sampling functions (e.g., R's `rnorm()`) the use of summary statistics or plots can be adopted. In the event that the results do not visually or statistically conform to the investigators expectations, the investigator can then begin to check for coding issues, such as accidentally passing a variance parameter to a function that was expecting a standard deviation (Morris et al., 2019). Taking this one step further, it is therefore recommended that investigators *explicitly* match the respective argument names for their function inputs, particularly for functions that they (or members of their team) are less familiar with. In the case of the `rnorm()` function, users should try to avoid invoking order-based inputs for the function's arguments, such as `rnorm(N, m, d)`, and instead match the arguments explicitly according to the argument names, such as `rnorm(n=N, mean=m, sd=d)`. Explicitly matching function arguments not only improves readability of the code but helps in detecting whether the function's inputs were indeed what the investigator was anticipating at the outset.[3]

In interpretive programming languages such as R, extracting information from statistical analysis objects is also of paramount importance. R provides many ways to extract data elements, though naturally the extraction method depends upon the `class` of the storage object. As a small set of examples, elements in a `vector` may be extracted via the `[]` operator, elements in a `list` or `data.frame` via the `$` or `[[]]` operators, elements from an S4 class via the function `slot()` or the `@` operator, and so on. As such, knowing the structure of an object and the contents found within the object is extremely important. One very useful function in R for this purpose is the `str()` function (alternatively see `dplyr::glimpse()`), which can be used interactively (e.g., when debugging) to print the structure and contents of an object, and provides information regarding how the elements in the object(s) can be extracted. As was true regarding the matching of arguments to functions, whenever possible investigators should also extract elements from R objects according to the name of the element (e.g., `x["element_name"]` and `x$element_name`) rather than based on the respective location of the element (e.g., `x[2]` and `x[[2]]`) since this will help improve the overall readability of the code and help protect against extracting incorrect information in the event that future code modifications (either by the writers or by package developers) alter the ordering of an object's elements.[4]

While R's extraction approaches are indeed powerful and flexible, they may not be the most consistent approaches to use, particularly when using third-party packages. For example, `lavaan` (Rosseel, 2012) and `mirt` (Chalmers, 2012) are two packages that provide explicit extraction functions that should be used to obtain internal object information. The purpose of these extraction functions is to provide consistency and stability in the code-base across package versions so that if internal object information is ever moved the behaviour of the extraction functions will remain consistent.[5] Therefore, whenever possible it is recommended that investigators use dedicated extraction functions provided by package developers, which also require the explicit name of the element to be extracted (see above), since these are often safer than extracting elements manually and, consequently, are often better documented.

### *The `SimDesign` package*

In addition to the general desirable features of MCSs listed in the introduction section, there are of course numerous specific defensive programming features that should be considered before the writing of MCS code begins. For example:

- Accidentally overwriting previously defined files on the hard-drive should be prevented, and instead new file names should be generated automatically in cases of naming conflicts,
- Temporary storage of the simulation state should be saved to the hard-drive in case of power outages or other computational issues,
- MCS conditions should be terminated early if more than a certain number of consecutive errors appear, which ultimately helps to avoid infinite loops and execution inefficiency,
- External package and software information (useful for independent reproducibility), computer hardware information, dates, execution times, etc, should be automatically stored,
- ... and much more.

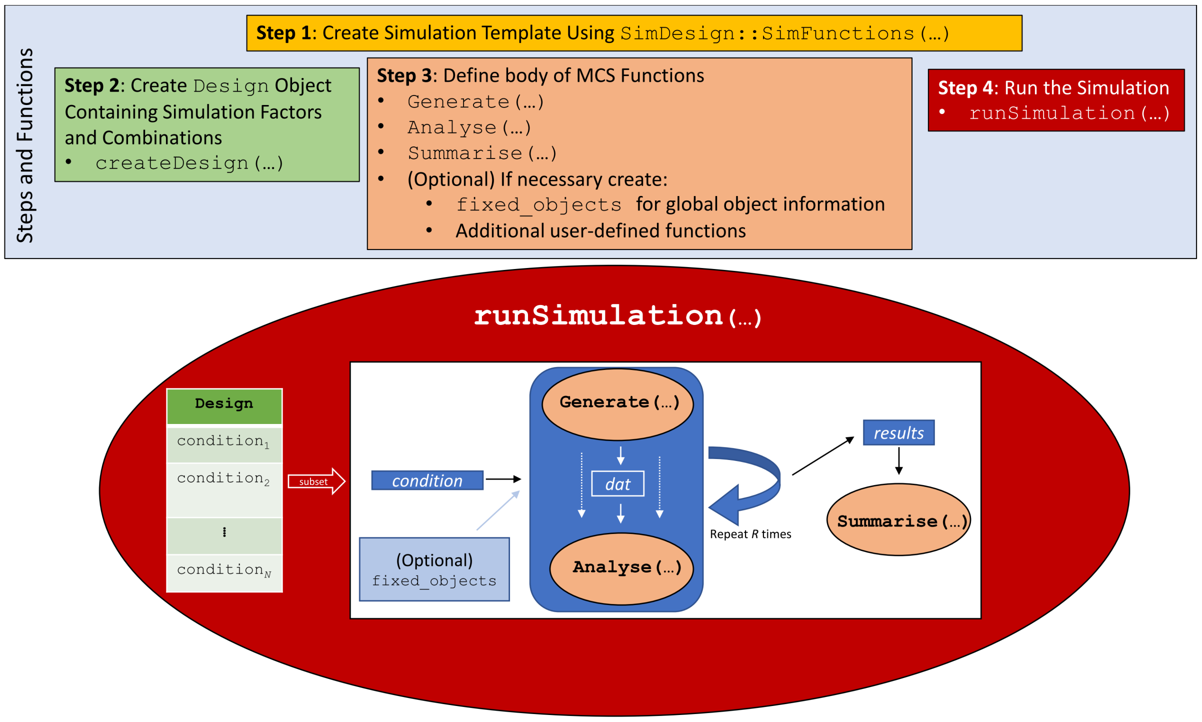
The list of specific desirable features for MCSs is potentially quite large, and each feature adds additional complexity to the MCS code-base. Given the above coding recommendations and desired properties, it is clear that in order to write a safe, efficient, well-organized, reproducible, and

[3]Integrated development environments (IDEs), such as RStudio, are useful in this respect in that they provide effective support for auto-completion matching of function arguments.

[4]Again, high-quality IDEs such as RStudio are helpful in this setting due to their auto-completion features, which also applies to extracting data elements by name from defined objects.

[5]Compare this behavior to the extraction of an element from an S3 wrapped `list` object via the `$` operator that no longer exists, which would inadvertently return `NULL` instead of the desired element.

**Figure 1** ■ Conceptual organization of SimDesign's workflow. Top graphical boxes represents the four steps required, while the bottom elliptical image depicts the conceptual organization and interaction of the functions and R objects utilized by the package.


Steps and Functions
Step 1: Create Simulation Template Using SimDesign::SimFunctions(...)
Step 2: Create Design Object Containing Simulation Factors and Combinations
• createDesign(...)
Step 3: Define body of MCS Functions
• Generate(...)
• Analyse(...)
• Summarise(...)
• (Optional) If necessary create:
• fixed_objects for global object information
• Additional user-defined functions
Step 4: Run the Simulation
• runSimulation(...)
runSimulation(...)
Design
condition$_1$
condition$_2$
⁝
condition$_N$
subset
condition
(Optional) fixed_objects
Generate(...)
dat
Analyse(...)
Repeat R times
results
Summarise(...)

otherwise optimal MCSs requires adding a large (and often unrealistic) amount of extra code to the simulation experiment's code-base. In this vein, we recommend that investigators begin writing their MCS code using previously developed template tools from software that are specifically dedicated to implementing the features and overall desideratum listed above. Specifically, we recommend adopting the SimDesign package (Chalmers, 2020) because it has been designed to contain all of the coding principles and features previously discussed, either automatically or with minimal effort to the investigator, thereby reducing the cognitive load and technical barriers that investigators will likely encounter.

When first constructing a MCS experiment with SimDesign (and after installing the package via install.packages("SimDesign")) the following four steps are required. As well, see Figure 1 for a visual portrayal of SimDesign's coding workflow.

1. Use SimDesign::SimFunctions() to generate a structural template containing a generate-analyse-summarise functional workflow[6]. The first argument to SimFunctions() (labeled file) can be used to automatically save this template to a suitably named R script on the hard-drive, which by default saves the file to the current working directory. If front-end users are using the RStudio IDE then this associated file will also be opened automatically, allowing editing to begin immediately.
2. Modify the default Design object definition to include the simulation factors and combinations to be studied. This is performed by passing named, comma separated vector objects to createDesign(), which builds a completely-crossed MCS experimental design object containing all combinations of the supplied factor variables row-wise. For designs that are not completely crossed, for reasons of redundancy or explicit removal of problematic simulation conditions, a subset of this object can be obtained by supplying a logical vector to argument subset.
3. Modify the Generate(), Analyse(), and Summarise() functions to generate the data, perform the statistical/mathematical analyses, and sum-

[6]The :: operator avoids the need for first loading the package via library(SimDesign) when defining the template code.

Listing 1 ■ Code-block output which contains the template result when calling `SimDesign::SimFunction()` with no additional arguments passed

```
#-------------------------------------------------------------------
library (SimDesign)
Design <- createDesign(factor1 = NA,
                       factor2 = NA)
#-------------------------------------------------------------------
Generate <- function(condition, fixed_objects = NULL) {
    dat <- data.frame()
    dat
}
Analyse <- function(condition, dat, fixed_objects = NULL) {
    ret <- c(stat1 = NaN, stat2 = NaN)
    ret
}
Summarise <- function(condition, results, fixed_objects = NULL) {
    ret <- c(bias = NaN, RMSE = NaN)
    ret
}
#-------------------------------------------------------------------
res <- runSimulation(design=Design, replications=1000, generate=Generate,
                     analyse=Analyse, summarise=Summarise)
res
```

marise the results over a large number of independent replications, respectfully. If not completed earlier, investigators should also define/`source()` any other user-defined functions that are to be utilized in the simulation.

4. Finally, modify the arguments to `runSimulation()` to control the execution features of the MCS experiment. This includes, but is not limited to: the number of `replications`, enabling parallel processing, saving the simulation results and temporary object states, debugging, attaching third-party packages for ease of function calling, and so on.

Finally, the code-block output of Listing 1 (as part of Step 1) contains the template result when calling `SimDesign::SimFunctions()` with no additional arguments passed.

Before beginning with `SimDesign`, users may find it helpful to inspect the documentation associated with `runSimulation()` by typing `help(runSimulation)` in the R console after first attaching the package. This documentation, though complex, provides helpful examples and technical descriptions of the function arguments, describes the coding work-flow required for utilizing the `SimDesign` package, documents several of the additional features that the package supports, and provides references to other functions that may be of interest in other sections of the package. As well, readers may find it helpful to inspect the (somewhat dated) pedagogical overview of the `SimDesign` package provided by Sigal and Chalmers (2016), in addition to the freely available examples located on the package's wiki page: https://github.com/philchalmers/SimDesign/wiki.

## A walk-through example adapted from Hallgren (2013)

As a motivating example that expands upon previously published simulation code, the following represents a re-expression of the simulation presented by Hallgren (2013) pertaining to the performance of mediation analyses using the Sobel (1986) test. In this simulation, Hallgren provided R code to investigate whether mediation models produce significant results (i.e., $p < .05$) when correctly and incorrectly specifying a linear three-variable system intended to reflect an underlying causal mediation structure. For more detailed information regarding this simulation refer to Hallgren (2013), as well as their Appendix A. Finally, for ease of execution, the following described code blocks are presented as a single script in Appendix , which can be copy-pasted into an active R session.

### *Simulation template, user-defined functions, and simulation design*

As discussed above, when writing simulation code investigators should avoid tackling the whole task all at once. Instead, investigators should think of ways to organize and

**Listing 2** ■ Sobel's delta method

```
# Generate then edit R template file ''Hallgren2013.R''
# SimDesign::SimFunctions('Hallgren2013')
# function returns data.frame of p-values, estimates, SEs, etc
sobel_test <- function(X, M, Y){
    M_X <- lm(M ~ X)
    Y_XM <- lm(Y ~ X + M)
    a <- coefficients(M_X)['X'] # coef for 'X'
    b <- coefficients(Y_XM)['M'] # coef for 'M'
    stdera <- summary(M_X)$coefficients["X", "Std. Error"] # SE for 'X'
    stderb <- summary(Y_XM)$coefficients["M", "Std. Error"] # SE for 'M'
    sobelz <- a*b / sqrt(b^2 * stdera^2 + a^2 * stderb^2)
    sobelp <- pnorm(abs(sobelz), lower.tail=FALSE)*2
    ret <- data.frame(a=a, SE_a=stdera, b=b, SE_b=stdera,
                      z=sobelz, p=sobelp)
    ret
}
```

test the coding steps in an isolated function-by-function basis and, particularly early on, frequently inspect whether each line of the code is providing the intended result (Morris et al., 2019). In the context of the SimDesign package, this functional organizational task is strictly imposed by following and modifying the default simulation template provided, which in this example was generated using SimDesign::SimFunctions("Hallgren2013") to save the template to a file named "Hallgren2013.R". This template provides a stand-alone R script to be (sometimes recursively) edited from top-to-bottom until the investigator has implemented their MCS design successfully. In this example, the coding information was largely prewritten by Hallgren (2013), and therefore only minimal changes were required after copy-pasting code from the original publication.

After executing SimFunctions() it is important to locate the statistical analyses and data generation functions that are required for the MCS. If a particular analysis/generation function is not available in the base R installation, nor in third-party R packages, then the investigator must define these functions themselves.[7] In this example, the construction of a user-defined sobel_test() function was created to perform Sobel's delta method test (Sobel, 1986), which in this functional implementation returns a data.frame containing the inferential information and parameter estimates of a linear three-variable mediation analysis. It is provided in Listing 2. [8]

After locating or defining all the necessary statistical analysis functions, the next step in the MCS code is the construction of the simulation *factors* to study and their respective combinations to form the unique simulation *conditions*, as given in Listing 3. In SimDesign's template this information is organized via the object Design, which is conveniently created by createDesign() after supplying a set of meaningfully named vector objects. The simulation factors passed to createDesign() will automatically generate a fully-crossed simulation experiment containing all possible combinations of the supplied factor levels, where the resulting rows of the Design object indicate the unique factor combinations (and therefore the unique simulation conditions to be evaluated). In this example there are four factors that are completely crossed: the sample size (N) with two levels, and three distinct sets of slope parameters (a, b, and cp), each with three levels, resulting in $2 \times 3 \times 3 \times 3 = 54$ unique simulation conditions; hence, Design will have 54 unique rows and four columns. Again, for this Design object it is important to name the associated factors meaningfully, and in most cases concisely, because the column names of Design will be used in the subsequent code-base (e.g., if N was not a particularly meaningful factor name the investigator could instead use sample_size to improve subsequent readability).

By default, the returned object from createDesign() is a tibble, which is effectively a modern variant of

[7]"Locating or defining functions" could be performed before or after the Design object has been constructed because these steps are largely independent.

[8]The bda package contains the function mediation.test(), which could have been adopted instead of the user-defined function, and could be included in the simulation by passing runSimulation(..., packages = "bda") to attach the package for single or multi-core execution of the simulation. However, the user-defined function approach below is included to keep consistent with Hallgren (2013).

**Listing 3** ■ Construction of the simulation *factors* and their respective combinations

```
library(SimDesign)
# fully-crossed simulation experiment
Design <- createDesign(N = c(100, 300),
                        a = c(-.3, 0, .3),
                        b = c(-.3, 0, .3),
                        cp = c(-.2, 0, .2))
Design

## # A tibble: 54 x 4
##        N     a     b    cp
##    <dbl> <dbl> <dbl> <dbl>
## 1   100  -0.3  -0.3  -0.2
## 2   300  -0.3  -0.3  -0.2
## 3   100   0    -0.3  -0.2
## 4   300   0    -0.3  -0.2
## 5   100   0.3  -0.3  -0.2
## 6   300   0.3  -0.3  -0.2
## 7   100  -0.3   0    -0.2
## 8   300  -0.3   0    -0.2
## 9   100   0     0    -0.2
## 10  300   0     0    -0.2
## # ... with 44 more rows
```

the `data.frame` object with better print formatting, and is a cornerstone of the `tidyverse` framework. `SimDesign`'s philosophy when creating this `tibble` object is also consistent with the `tidyverse` in that, for example, `character` vectors are not automatically coerced to `factor` variables (which is the default behaviour in the `data.frame()` function in R in versions lower than 4.0.0), and the R output is automatically adjusted to fit the width of the investigator's console. However, printing the `Design` object may be slightly different when a `list` element is passed to `createDesign()`, in which case the output is printed as a modified `character` vector if this is deemed reasonable (e.g., see the Flora & Curran, 2004 example below). [9]

### *The* `runSimulation()` *function*

Although `runSimulation()` is the last function presented in the generated `SimFunctions()` template, it is beneficial to introduce this function at this stage because it provides a conceptual end-of-the-road goal that investigators are ultimately working towards. This function is also briefly discussed now because a selection of arguments passed also appear in the required generate-analyse-summarise functional definitions, and therefore being aware of their origin will be helpful in conceptually understanding their purpose.

By default, `SimFunctions()` defines `runSimulation()` as

```
res <- runSimulation(design=Design,
                     replications=1000,
                     generate=Generate,
                     analyse=Analyse,
                     summarise=Summarise
    )
```

indicating that the function requires, at minimum, three functional definitions for the generate-analyse-summarise steps, the previously described simulation `Design`, and the number of independent simulation `replications` to evaluate for each row condition in `Design` (default is 1000). For simplicity, we will retain these default arguments in the example in this section; however, in the next section the details of `runSimulation()` are unpacked further to discuss the features and benefits of executing MCS code through this controller function.

When editing the remaining template functions, it is important to keep in mind that `runSimulation()` will accept only a single function for each of the `generate`, `analyse`, and `summarise` arguments. Much the same as R's "`apply`" function, this strict structure implies that

[9]This does not change the values within the `list` object, it just modifies how it is printed to the console to improve readability of the `list` elements.

the to-be-redefined template functions must validly support each of the supplied row conditions in `Design`. This is important to be aware of early on because if a particular row in `Design` requires a notably different set of coding instructions then the investigator must provide a suitable "logical branch" to accommodate these forks in the execution workflow. Such forks are accomplished by way of `if(logical)` statements, where `logical` is a single `TRUE/FALSE` value to be defined according to R's logical evaluation operators (e.g., `==`, `!=`, `<`, `<=`, `is()`). Fortunately, in the current example such `if(...)` branches are not required due to the general simplicity of the simulation; however, in the simple extension (Appendix B) and the real-world example below such forks are required due to the added complexity of the experimental conditions and analyses.

*Commonalities across generate-analyse-summarise functions*

Following the definition of the `Design` object, and keeping focus on the execution control via `runSimulation()`, the body of the templated `Generate()`, `Analyse()`, and `Summarise()` functions must be modified to suit the purpose of the simulation. Each of these core functions share a common design and grammar for internal consistency to help reduce the cognitive load required to understand the code. An example of this consistency can be found in the argument `condition`, which is the first argument to all three MCS definition functions. This internally defined variable always represents a single extracted row from the `Design` object (e.g., on the first major MCS iteration, `condition <- Design[1, ]`), and therefore will always contain only one simulation condition at a time. By default, this object will be a `tibble` with one row, where the names of the columns are identical to the names of the supplied `Design` object. As alluded to earlier, this is why naming the columns in `Design` is particularly important because the names of the supplied variables are used directly in the resulting simulation code. Additionally, a new column variable called `ID` is included in this object to indicate which row in `Design` is currently being evaluated, and a new column called `REPLICATION` is included to indicate which independent replication is currently being evaluated; hence, `ID` will range from `1` to `nrow(Design)`, while `REPLICATION` will be an integer between `1` and `replications`.

The benefit of using a subset of the `Design` object directly in the form of the `condition` variable is that extracting and creating temporary objects for each factor level is not required. Additionally, because meaningful factor names have already been assigned to `Design` there is no need to create new naming conventions based on the specific levels being extracted. Compare this conceptual indexing setup to the more common for-loop strategy (Sigal & Chalmers, 2016), which strictly requires creating temporary index objects such as `for(n in N){...}` and tracking the original (`N`) *and* indexed (`n`) version of the object. If, on the other hand, the `for()` loop is designed as an integer indexing value to extract the *i*th element from, for instance, a `list`, then a total of three objects will be required: `n`, `N`, and the `list` object being indexed. Evidently, the added complexity in the for-loop approach will always double or triple the number of objects to track, making it more likely to accidentally replace or misplace an object in the workflow (e.g., `n` or `N` may be redefined as a new object within the `for()`), ultimately increasing the cognitive load when reading, writing, and debugging the script. This highlights one of the underlying philosophies of the `SimDesign` package, and why the supplied template structure is structurally strict: *readability is greatly improved by avoiding multiple named instances of the same conceptual object*.

Additionally, one of the main benefits of isolating the data generation (`Generate()`), statistical analyses (`Analyse()`), and summerization (`Summarise()`) procedures by way of independent functional implementations is that they help to keep the programming environment as uncluttered as possible. This is accomplished by recognizing that only the final object created within `Generate()`, for example, will be accessible to `Analyse()`, while all other objects defined within `Generate()` will be discarded after the function exits. Moreover, the underlying structural benefit of returning a single data object from `Generate()` is that it keeps the data generation and subsequent statistical/mathematical analyses completely independent, thereby forcing investigators to explicitly organize (and meaningfully name) the variables in their generated data. This functional separation of the simulation components naturally results in the removal of any temporarily defined objects that were required to create the final data object returned by `Generate()` (incidentally also resulting in lower memory usage due to R's garbage collector). More importantly, this separation conceptually reflects the structure of how statistical/mathematical analyses are implemented with real data: analyses are performed after collecting empirical data *without definitive knowledge of the population generating structure*. As such, MCS code should respect this particular data-analysis separation nature so as to avoid any temptation to utilize 'known' population information

[10]There are special practical exceptions to this general data-analysis separation philosophy, such as supplying better starting values to analysis functions that require iterative estimation methods that are prone to terminating on local optimums given their respective objective functions. In these

**Listing 4** ■ Construction of the `Generate()` function

```
Generate <- function(condition, fixed_objects = NULL ) {
    Attach(condition) # make N, a, b, and cp directly accessable
    X <- rnorm(N)
    M <- a*X + rnorm(N)
    Y <- cp*X + b*M + rnorm(N)
    dat <- data.frame(X=X, M=M, Y=Y)
    dat
}
```

from the data generation steps that could be unrealistically borrowed in the analyse step[10].

Continuing on, the template of the generate-analyse-summarise functions also contains an optional argument called `fixed_objects`, which can be of any data type or structure, and is caught from the respective argument passed to `runSimulation()`. This argument is useful when various fixed objects are to be used across two or more simulation conditions, and acts as though the supplied object is 'global' information to be shared across conditions. Most often, it is sufficient for this object to be a named `list` containing the entire set of fixed objects required within the simulation conditions[11]. As an example of its use, say that an investigator has three sets of distinct `matrix` objects containing structured population parameter definitions called `mat1`, `mat2`, and `mat3`. Supplying `runSimulation(..., fixed_objects = list(mat1=mat1, mat2=mat2, mat3=mat3))` will create a named `list` with the respective objects to be indexed from inside the simulation code.[12] This said, the use of `fixed_objects` is generally only necessary for convenience purposes, and by-and-large can be avoided until it becomes practically beneficial to utilize.

### *The `Generate()` function*

Focusing on the first templated MCS function that should be modified, `Generate()`, the goal of this function is to return a suitable data object that subsequently will be supplied to the analysis function(s) implemented in `Analyse()`. The object returned by `Generate()` is not limited to one object class, in that it could be a `numeric` vector, `list`, `array`, and so on, though more often returning a `data.frame` (or in some cases, a `tibble`) will be the most appropriate since this is R's primary data container for statistical analysis functions. Continuing with Hallgren's 2013 simulation with respect to the body of `Generate()` given in Listing 4, after defining the distribution of `X`, as well as the required linear equation definitions for `M` and `Y`, the data are organized and returned as a `data.frame` object. Notice that regardless of the selected row in `Design` (defined as the variable `condition`) the code executes correctly.

Readers familiar with R's scoping rules will notice at this point that the objects `N`, `a`, `b`, and `cp` technically should not be accessible within the scope of this function because they are contained within the `condition` object; hence, in traditional R code these elements must be accessed using, for example, `condition$N`, `condition$a`, `condition$b`, and `condition$cp` in all instances when they appear, or wrapped within a `with()` call. However, the `SimDesign` package contains a useful convenience function called `Attach()`, which extracts or "attaches" any object contained in a given `list` object (which `data.frames` and `tibbles` are a special case of) and places these objects in the environment from which the function was called.[13] Attaching the names of a `list` object this way has a number of inherent benefits: it reduces the amount of typing required, makes the code more readable, reduces the cognitive load of object tracking, eliminates the need to rename conceptually identical

cases, the population generating parameters could be passed along with the dataset as a `list` object from `Generate()` to provide the starting values for the statistical estimates as well as the generated data. Nevertheless, what is important here is that although `SimDesign` imposes a strict coding philosophy, there remains sufficient flexibility in the package to allow violating the base recommendations, should the need arise.

[11] Some examples include defining larger objects such as covariance matrices, large sets of population parameters, regression-based design matrices that would be identical across multiple simulation condition combinations, and so on.

[12] If convenient, the names of the fixed objects can also be included in the `Design` definition for ease of tracking in the simulation design and for extraction since only one defined name will be included at a time on a per-row basis. Specifically, `Design <- createDesign(..., mat=c('mat1', 'mat2', 'mat3'))`, and selecting the respective matrix during the simulation can be achieved via `mat <- fixed_object[[condition$mat]]`.

[13] `Attach()` is similar in spirit to R's `attach()` function in that objects can be directly accessed via their explicit names. However, `Attach()` differs from `attach()` in that it only assigns objects to a specific environment (not the global environment), requires much less execution time, and will throw an error if an object exists in the scope that would be replaced upon attaching. This is one of the many silent built-in safety features to prevent investigators from making unnecessary mistakes.

**Listing 5** ■ Definition of SimDesign's `Analyse()` function for replicating Hallgren's (2013) simulation study

```
Analyse <- function(condition, dat, fixed_objects = NULL ) {
    Attach(dat) # make objects X, M, and Y directly accessible
    sobel <- sobel_test(X=X, M=M, Y=Y)$p
    sobel_incorrect <- sobel_test(X=X, M=Y, Y=M)$p
    ret <- c(sobel=sobel, sobel_incorrect=sobel_incorrect)
    ret # named vector of p-values
}
```

objects, and capitalizes on the originally meaningful factor names supplied to the `Design` object (i.e., `N` now *always* represents a single value for sample size in the code).

### *The `Analyse()` function*

Moving now to `Analyse()`, the purpose of this function is to return all the analysis information (given a generated data object) required to answer the researcher's simulation experiment questions. Depending on the nature of the simulation experiment this function may return a set of parameter estimates, confidence intervals, $p$-values, goodness-of-fit test statistics, and so on. As such, the goal when writing the contents of `Analyse()` is to 1) perform all the desired statistical analyses on the generated dataset, which in the definition of `Analyse()` appears in the second argument called `dat`, 2) extract the required information from these resulting analysis objects, and 3) return this information in the form of a meaningfully labeled R object. As a safety feature, if a `numeric` vector is to be returned then `SimDesign` will check whether a unique label has been assigned to each element in the vector for future identification purposes. This forces investigators to explicitly label their resulting analysis information so as to avoid subsequent confusions regarding the collected simulation results (more on this below).

In the mediation analysis example, and looking at Listing 5, the generated data vectors `X`, `M`, and `Y` are first made accessible by `SimDesign`'s `Attach()` function and subsequently passed to the user-defined function `sobel_test()` described earlier. Recall that while `sobel_test()` returns a `data.frame` containing several named elements, the goal of this particular MCS study is to investigate the difference between the statistical significance of Sobel's test when the mediation system is correctly or incorrectly specified; hence, we must extract the element `p` from the resulting analysis objects to obtain the required $p$-values, whose sampling behaviour will be summarised later. In the following, two extracted $p$-values are returned in a named vector object with the elements named `"sobel"` and `"sobel_incorrect"`, indicating whether the mediation system was or was not correctly specified, respectively.

As was the case with `Generate()` there is no strict structural rule regarding the object type that must be returned from `Analyse()`, though most often a named `numeric` vector will be the most effective form of output for the package to automate. As such, a named vector object is the recommended object that users should try and return since it will ultimately make the final MCS function, `Summarise()`, the most general, and also releases the investigator from additional data reshaping steps. In more complicated simulations, `data.frames` (with multiple rows) or `lists` may be returned, for instance, though indexing these resulting objects will ultimately require manual reshaping and indexing by the investigator.

### *The `Summarise()` function*

Finally, given a single row from the `Design` object (i.e., `condition`), the provided `Generate()` and `Analyse()` functions are independently replicated $R$ times, where $R$ is the number of independent `replications` to be evaluated. After these replications have been executed and collected within each simulation condition, it becomes important to summarise the general sampling behaviour of these replicates for inferential purposes. Depending on the purpose of the MCS, many meta-statistical summaries are available, including, but not limited to: bias, root mean-square error, mean-absolute error, and more, for judging the "closeness" of the parameter estimates ($\hat{\theta}$) to the population generating parameters ($\theta$); empirical detection and coverage rates for evaluating the behavior of $p$-values and confidence intervals at nominal $\alpha$ levels; estimator comparison measures such as relative absolute bias, relative difference, relative efficiency, mean-square relative standard error; and so on.

Depending on which meta-statistics are deemed appropriate, investigators are typically faced with manually programming their desired summary meta-statistics (e.g., see Mooney, 1997), which can often lead to unnecessary coding errors and other inefficiencies. To circumvent these potential coding issues, `SimDesign` conveniently includes predefined functions that implement a variety of meta-statistics to ensure that they are computed accurately. As seen in Table 1 at the end of the article, `SimDesign`

**Listing 6** ■ Definition of SimDesign's `Sumarise()` function for replicating Hallgren's (2013) simulation study

```
Summarise <- function(condition, results, fixed_objects = NULL) {
    ret <- EDR(results, alpha = .05) # results object is a data.frame
    ret # empirical detection rate returned
}
```

**Listing 7** ■ Final output

```
res
## # A tibble: 54 x 10
##         N     a     b    cp sobel sobel_incorrect REPLICATIONS SIM_TIME COMPLETED
##     <dbl> <dbl> <dbl> <dbl> <dbl>           <dbl>        <int>    <dbl>     <chr>
##  1   100  -0.3  -0.3  -0.2 0.462           0.048         1000     7.59   Sat Nov ~
##  2   300  -0.3  -0.3  -0.2 0.997           0.374         1000     8.35   Sat Nov ~
##  3   100   0    -0.3  -0.2 0.01            0.225         1000     8.15   Sat Nov ~
##  4   300   0    -0.3  -0.2 0.049           0.863         1000     8.21   Sat Nov ~
##  5   100   0.3  -0.3  -0.2 0.465           0.448         1000     7.32   Sat Nov ~
##  6   300   0.3  -0.3  -0.2 0.994           0.987         1000     8.03   Sat Nov ~
##  7   100  -0.3   0    -0.2 0.016           0.003         1000     6.63   Sat Nov ~
##  8   300  -0.3   0    -0.2 0.034           0.021         1000     6.80   Sat Nov ~
##  9   100   0     0    -0.2 0               0.006         1000     6.70   Sat Nov ~
## 10   300   0     0    -0.2 0.001           0.026         1000     6.84   Sat Nov ~
## # ... with 44 more rows, and 1 more variable: SEED <int>
```

provides a wide variety of these meta-statistics as pre-defined functions. Moreover, related summary meta-statistics are bundled into single functions, and specific types can be accessed via the `type` argument; for example, `bias(..., type = "relative")` or `bias(..., type = "standardized")` will compute the relative bias or standardized bias meta-statistics, respectively.

In and of themselves the meta-statistics available within `SimDesign` are useful for avoiding unnecessary implementation errors; however, their usefulness stretches further in the context of `Summarise()`. After `Generate()` and `Analyse()` have been executed $R$ times a `results` object is internally defined to contain these $R$ sets of analysis results, and is ultimately passed as an argument to `Summarise()` given in Listing 6. Whenever possible, however, `SimDesign` will attempt to *combine all of these replications by row to form a* `data.frame` *object with $R$ rows and $P$ named columns* (where $P$ is the length of the named returned object from `Analyse()`). This will occur *only* when `Analyse()` returns a named vector object; otherwise, `results` will be a list object of length `replications`. This `data.frame` simplification is performed because all of `SimDesign`'s meta-statistical functions have been constructed to support *column-dominant* inputs, meaning that the meta-statistics are applied independently to each respective column while preserving the associated column names in the resulting objects.

While the column-dominant nature of `SimDesign`'s functions may sound intimidating at first, the conceptual application of this approach is straightforward. In the running simulation example, and given the description above, the reader can understand `results` as an $R \times 2$ `data.frame` containing the collected $p$-values, where the column names are "`sobel`" and "`sobel_incorrect`", in accordance with the named vector output from `Analyse()`; hence, each column in `results` pertains to the $R$ observed $p$-values from the respective statistical analyses. In the above code, `results` is passed to `SimDesign`'s empirical detection rate function, `EDR()`, and the proportion of collected $p$-values that fall below $\alpha = .05$ are computed for each respective column. The object `ret`, which stores the results from `EDR()`, will in turn be a `numeric` vector consisting of two proportion estimates given $\alpha$, and most importantly will retain the associated variable names for each respective element. The flexibility of this simplification and column-dominant nature is now hopefully more clear and appealing: regardless of the number of statistical analysis procedures utilized in `Analyse()`, the results from the meta-statistics supplied within `Summarise()` are capable of retaining the associated vector names originally returned by `Analyse()` without any further modifications, thereby preserving the original object labels. As such, this greatly reduces the chance of incorrectly relabeling/mislabeling any meta-statistical summary operation.

## Final output

After suitable definitions of the generate-analyse-summarise functions have been constructed, the simulation can be finally be executed using `runSimulation()` and stored into an R object, such as the default template object named `res`. This object, which for this example is printed in Listing 7, inherits the properties of a `tibble`, and therefore gains many attractive printing qualities such as: clipping the rows and columns of the output to the dimensions of the R console, printing the `class` of each column variable at the top of the output, and implicitly supports `dplyr`'s powerful verbs for data manipulations and summarizing. This output structure is useful because after a simulation is completed the subsequent results are often further disentangled through the use of (conditional or marginal) summary statistics, visualization methods, statistical modeling, and so on, as though the collection of summarized MCS results were obtained as a sample of observations in an empirical experiment (Mooney, 1997).

Regarding the variables in the `res` object, the structure of this `tibble` can be best understood as three distinct pieces that have been combined by columns: the initial `Design` object in the left block (or the leftmost column(s) in the `tibble`), the results from `Summarise()` in the middle block (columns `sobel` and `sobel_incorrect` in this case), and extra MCS implementation information in the right block. The extra information in this object, whose columns are always labeled with capital letters by convention, reflect the number of replications used for each simulation condition (`REPLICATIONS`), the execution time each simulation condition took to complete (`SIM_TIME`), the date at which the simulation completed (`COMPLETED`), and the initial seed used for each simulation combination to allow for macro reproducibility of each row condition (`SEED`). These are not the only additional pieces of information stored in this object, however, as more column variables may be appended depending on the arguments passed to `runSimulation()` (e.g., see the `boot_method` argument) or whether noteworthy characteristics were observed during the simulation (such as the frequency of error/warning messages).

Other useful implementation information stored within the returned simulation object `res` can be extracted by using `summary(res)`. This S3 generic function extracts information such as the R session information used to execute the simulation, which additional packages (if any) were requested and passed through `runSimulation(..., packages)`, the file and directory information used if any objects were saved to the hard-disk, the number of cores requested (if parallel processing was used), the number of simulation conditions executed, the date and time the simulation completed, and the total time required to complete the simulation. Coupled with the variables located within the `tibble` object itself, such additional information is of great importance when attempting to reproduce or report the details of a simulation study. Additional information may be extracted from this object through functions such as `SimExtract()`, some of which are discussed in the next section.

Finally, while the code-base from the simulation above and the code provided by Hallgren (2013) are functionally identical, it seems noteworthy to highlight that there were immediate organizational and performance benefits when using `SimDesign`'s setup. Specifically, as can be readily seen there are no visible for-loops required in the code-base, where instead the simulation is controlled by automatically iterating over the row conditions in the `Design` object. This has obvious organizational benefits, some of which were briefly discussed in this section and above. Perhaps most surprisingly, however, is that executing the `SimDesign` variant of Hallgren's simulation on a single-core R session was more than twice as fast as the original for-loop approach, despite the fact the `SimDesign` is performing many more safety checks and data manipulation operations behind the scenes.

## A Selection of Noteworthy Features and Helper Functions

This section overviews a number of important features currently supported by `SimDesign`; however, it is by no means exhaustive. Interested readers may locate further information pertaining to how `SimDesign`: catches and manages errors and warnings; handles objects and functions for parallel computing architecture; saves simulation results to hard disks; and more, by visiting the associated CRAN location on-line and perusing the HTML vignette files at https://cran.r-project.org/web/packages/SimDesign/index.html.

### *In-line simulation modifications*

One particularly attractive property of `SimDesign` is that coping with future changes to the experimental conditions, data generation process, statistical analyses, and meta-statistical summaries will often require minimal — and often entirely isolated — modifications to the working version of the simulation script. For instance, adding additional levels to existing factors, such as in Hallgren's (2013) example above (e.g., `Design <- createDesign(N = c(100, 300, 500), ...))`), does not require any modification to the internal code-base, while adding or removing factor variables will require minor modifications to accommodate these structural changes. For example, including an additional factor dictating whether the `X` variable should be drawn from a standard normal distribution or a Bernoulli distribution with $\rho = .5$ could be achieved by adding a new factor to `Design`,

**Listing 8** ■ Construction of the `Generate()` function

```
Generate <- function(condition, fixed_objects = NULL ) {
    Attach(condition)
    X <- if(dist == 'norm') rnorm(N) else rbinom(N, size=1, prob=.5)
    ... # remainder of Generate() code is the same as above
}
```

**Listing 9** ■ Construction of the `Analyse()` function

```
Analyse <- function(condition, dat, fixed_objects = NULL ) {
    ... # portion of Analyse() code from sobel_test() and above
    boot <- boot_test(X=X, M=M, Y=Y)$p
    boot_incorrect <- boot_test(X=X, M=Y, Y=M)$p
    ret <- c(sobel=sobel, sobel_incorrect=sobel_incorrect,
             boot=boot, boot_incorrect=boot_incorrect)
    ret
}
```

```
Design <- createDesign(
    # include all previous factors
    ...,
    # add new factor to cross
    dist = c('norm', 'bern'))
```

where "`norm`" and "`bern`" indicate whether a normal or Bernoulli distribution should be used in the data generation, and after including this new factor a one-line modification can be made for the `X` variable definition in `Generate()`, as seen in Listing 8.

This example highlights the usefulness of the logical flow-controller `if()` to create a fork in the code depending on the given instance of `dist`, and also highlights the ease with which new simulation factors and conditions can be added to existing code.

Adding additional statistical analyses in `Analyse()` also generally requires a trivial amount of effort on the investigator's end. In this case, investigators need only include their new statistical function(s), extract the relevant information they are interested in, and add these new results as named elements in the final object in `Analyse()`. `SimDesign` will then automatically adjust the size of the internal storage objects by adding new column(s) to the `results` object in `Summarise()`. Listing 9 is an example of such an extension, where a bias-corrected bootstrap analysis using the `lavaan` package (Rosseel, 2012) has been added to `Analyse()` to return suitable non-parametric bootstrapped $p$-values. Notice that the only changes to the code are 1) the inclusion of the new function `boot_test()`, and 2) the organization of the returned named vector in `Analyse()`, which now contains four elements instead of two. For completeness, each of the above described modifications to the original code-base, as well as a suitable definition for `boot_test()`, are presented in Appendix B.

The purpose of the above modification information was to highlight that adding, removing, and modifying simulation code should have minimal impact on the structure and readability of the original code-base. Under `SimDesign`'s structure, investigators are free to begin with simulation code that is of minimal working complexity, such as defining only one simulation factor in the `Design` object at first, and writing bare-minimum `Generate()`, `Analyse()`, and `Summarise()` functions until the simplest version of the code executes as desired. Once this baseline has been established and well tested, additional complexity can gradually be added since the majority of the (typically distracting) data organizational programming work is automatically managed by the package.

### *Noteworthy features in `runSimulation()`*

In addition to the general performance and readability improvements compared to for-loop strategies, implicit support for parallel architecture execution, and ease of adding modifications to the code-base, `runSimulation()` can be used to control other commonly desired features in MCSs. For instance, if users are worried about power-outages or software crashes while executing their MCS, and therefore wish to include temporary checkpoints to store the progress of their simulation, then the arguments `save` and `filename` should be included. The argument `save` (defaulted to `TRUE`) triggers the automatic storage of a temporary `.rds` file (an R binary file that can be read into R and assigned to a unique object using `readRDS()`) that is saved in the current working directory, while `filename` specifies the hard-drive file to save the final simulation results to upon completion (also as a `.rds` file; default is `NULL`). When `save = TRUE`, and in the event that a simulation is interrupted, all that is required to restart the simulation is to re-run `runSimulation()` in the original working directory. `SimDesign` will then automatically detect the temporary file at the last stable state, and continue the MCS at this checkpoint. Both of these inputs are recommended for more intensive simulations, where the likelihood of computer crashes or power outages increases as a function of time and resources available. As a safety precaution, files saved to the working directory, including temporary ones, will never accidentally be over-written by the package; instead, unique file names will be generated if `SimDesign` detects a name conflict, thereby preventing accidental over-writing of objects on the hard-disk.

The following code block demonstrates the use of file saving inputs, in addition to executing the code using all available computing cores (i.e., processors) by passing `parallel = TRUE`. In many simulations, slower computations within each independent generate-analyse iteration will benefit from parallel execution, where computation times will typically decrease at a rate proportional to the number of cores available[14]. For comparison purposes, when inspecting the original for-loop code presented by Hallgren it is more difficult to discern how to add parallel computation support due to the use of a single mutating R storage object, whose purpose was to store the simulation replication results across iterations (see the use of the `rbind()` function on p. 18 in Hallgren, 2013). With `SimDesign`, no such issues can occur because the structure is organized to always allow for parallel processing.

```
res <- runSimulation(
  design=Design,
  replications=1000,
  generate=Generate,
  analyse=Analyse,
  summarise=Summarise,
  parallel=TRUE,
  filename='Hallgren')
```

The three new arguments to `runSimulation()` described above are, of course, not the only useful inputs that could have been supplied. `runSimulation()` can also be modified to save the complete sets of $R$ simulation results per simulation condition to the hard-drive for later inspection and manipulations (via `save_results = TRUE`), while all possible `.Random.seed` states can all be saved for complete micro replication of the simulation (via `save_seeds = TRUE`).

For larger simulation studies, particularly studies that are to be submitted for publication purposes, it is recommended that `save_results` be set to `TRUE`. This is because `SimDesign` contains suitable post-summarising functions for re-summarising any generated results, should the need arise. Hence, any statistical information that may be required for summary purposes will be stored efficiently on the hard-drive, ultimately allowing for more flexibility in returning object information in the `Analyse()` step, even if these results were never used in the initial version of `Summarise()`. Storing the simulation results is also particularly important during the peer-review process in that if the investigator can anticipate reviewers' requests for additional information, or anticipate requests for alternative summarisation meta-statistics to be reported, then the simulation need not be re-run as all the analysis information has been previously stored on the hard-disk.

As a specific example regarding the usefulness of `save_results = TRUE`, suppose a MCS were designed for the purpose of studying the sampling behaviour of $p$-values, and therefore `Analyse()` will, at the bare minimum, return suitable $p$-values for `Summarise()` to utilize. However, there is little consequence in also returning additional information not used by the initial `Summarise()` definition, such as including the degrees of freedom, parameter estimates, standard errors, and so on, because this information could be (re-)summarised at a later time via `SimDesign` with little difficulty. As well, if a different $\alpha$ cut-off should have been used for the empirical detection rate estimates (e.g., $\alpha = .10$) then re-summarising the saved results using `SimDesign`'s `reSummarise()` function is also entirely possible with minimal effort.

### *Other safety features*

In addition to supporting explicit features by passing additional arguments, `runSimulation()` will also silently perform various safety operations that are common in the development of MCS experiments. For instance, in the event that the R interpreter catches a warning or error message (via calls from functions such as `warning()` or `stop()`, which may or may not have been explicitly writ-

[14]Users familiar with the scoping rules for parallel processing in R will notice that predefined custom functions do not need to be exported explicitly. This is because, for convenience purposes, all functions visible in R's global workspace are exported automatically when runSimulation() is called, thereby reducing the need for front-end users to track this process. Users more familiar with R's parallel computing definitions may also pass their own "cluster" object definition to `runSimulation()`'s argument `cl`. That said, parallel computations are generally only useful in more intensive MCSs since the time it takes to distribute and collect the information across computing cores can itself be a bottleneck.

ten by the investigator) then these messages, as well as the frequency with which they occurred, will be stored in the final simulation object. These important details regarding errors and warnings help investigators gauge whether, and to what extent, problems are occurring in their MCS, as well as provides an opportunity to supply their own simulation state checks to assure the simulated data and analysis information conform to their expectations. For instance, when generating code for a $2 \times 2$ contingency table for the purpose of studying $\phi$ correlation coefficients, investigators may only wish to compute this statistic when all the counts are greater than some cut-off in each cell (e.g., counts of 5); otherwise, the generate data should be discarded and re-drawn until this required condition has been achieved. To accomplish this in `SimDesign`, investigators need only create their desired test in the form of an `if(...)` statement, and supply a conditional `stop("...")` call to interrupt the MCS flow.

One of the more important built-in safety procedures in `SimDesign` appears when too many consecutive errors arise. By default, if consecutive errors appear in more than 50 replications then the respective simulation condition will be completely terminated. In this event, the last error message will be printed to the console, and the next row condition in the `Design` object will begin to be evaluated. MCSs that return many consecutive errors indicate severe coding or implementation issues since analysis/data generation functions are consistently failing, and therefore require additional (potentially immediate) attention. While this behavior may seem trivially obvious at first, such built-in safety features help safeguard against early (and frustrating) coding mistakes, guarantees that the desired number of replications are constant for all the simulation design conditions, and promotes the reporting of any error or warning messages that could negatively affect the veracity of the experimental results (Hoaglin & Andrews, 1975).

### *Debugging*

Writing optimal MCS code on the first attempt is rather uncommon, particularly for larger and more involved simulation experiments. Similar mistakes often appear when preparing virtually any text document in that revisions to the content are frequent, of varying intensity, and are required when unexpected or important events come to fruition. These revisions can range from something as simple as typographical errors to more serious (and difficult to track down) issues caused by third-party functions throwing obscure error messages under rare circumstances. As such, the process of debugging, even in the early stages of the coding development, is of utmost importance.

As should come as no surprise at this point, `SimDesign` includes special debugging features to initialize R's interactive debugging mode for each function in the generate-analyse-summarise workflow. These appear in the form of the `debug` argument to `runSimulation()`, which accepts three global debugging options (`"none"`, `"error"`, and `"all"`) and three function-specific options (`"generate"`, `"analyse"`, `"summarise"`). Beginning with the latter, passing `runSimulation(..., debug = "generate")` will trigger the debugging of `Generate()` on the first line of the function; `debug = "analyse"` and `debug = "summarise"` have the same behaviour for their respective functions. The option `"error"` will initialize the debugger when any error message is detected in one of three generate-analyse-summarise functions, which is useful during early testing stages for quickly tracking down initial coding errors interactively, while `"all"` will debug all the user defined functions regardless of whether an error was thrown or not. The option `"all"` is useful when testing the code-base in exploratory and experimental writing stages, or when walking through a specific `.Random.seed` state to inspect all the objects in each function to understand the nature of an error (more on this important feature below). Finally, if more explicit debugging is preferred then a `browser()` call may be placed inside the desired function at the exact location where the debugger should be initiated, which is particularly useful when utilized in conjunction with conditional `if()` statements.

Once the debugger begins, investigators may interactively use their R console to trace the properties of the defined objects, make use of functions such as `print()` and `str()` to view the objects within the debugger, and execute specific debugger commands to navigate the execution flow; specifically, `n` (next line), `f` (finish loop), `c` (continue until next debugger flag called), `Q` (quit the debugger), and so on (see `help(debug)` for more specific details). This has the usual "debugging functions" flavour found throughout R, however `SimDesign` also contains built-in safety features to ensure that debugging is properly executed when requested. For example, in everyday for-loop or `apply` function applications, when parallel processing is requested the R debugger will be invoked yet *will not be accessible by the user* since the interactive debugger is initialized within each distributed core; hence, no single debugger will be entered into on the investigators console due to the processor distribution ambiguity. `SimDesign` forbids such ambiguous behaviour by first checking whether a `browser()` or debugging flag has been activated, and if present will temporarily disable any unwanted parallel processing to ensure the interactive debugger is properly activated.

Perhaps the most useful feature of debugging is the ability to quickly replicate error states that arose in previous implementation attempts. As mentioned above, `SimDesign` automatically catches any error or warning messages that occur during the code execution, and the frequency of these messages will appear in the rightmost-columns of the final simulation object under the names `ERRORS` and `WARNINGS`. In situations where error messages are appended to the final results, the final simulation object will also contain the associated `.Random.seed` states for all error message that appeared. This provides the investigator an efficient means to replicate all recorded error messages to determine, in a hands-on way, the object properties and simulation state that cause errors to arise. Specifically, the `.Random.seed` values may be extracted via `seeds <- SimExtract(res, what="error_seeds")`, where the column names of the `seeds` object contains the error messages that were recorded. Once the desired seed is located (e.g., for presentation reasons the 10th column) this can be passed via `runSimulation(..., load_seed = seeds[ ,10])` to reproduce the exact state when the error appeared.[15] When paired with the `debug` input, such as with `debug = "all"`, this allows the investigator to interactively inspect the simulation's state in all relevant functions, providing a means to efficiently track down where, how, and why the MCS code raised this specific error.

## Flora and Curran (2004) Simulation

The mediation simulation presented by Hallgren (2013) that was re-expressed above represents a relatively straightforward simulation in terms of the overall design and code complexity. To demonstrate the use of `SimDesign` in real-world simulation experiments, this section presents a replication of the simulation published by Flora and Curran (2004), which pertained to studying violations of latent distributional assumptions in a selection of item factor analysis models estimated using structural equation modeling software (see Muthén, 1984). Note that although Flora and Curran used the `Mplus` software package (Muthén & Muthén, 2008) to perform their analyses, the following code uses the `lavaan` package (Rosseel, 2012) for simplicity, and to present a complete R code replication of this simulation experiment.[16]

Without going into many of the specific details, the purpose of Flora and Curran's (2004) simulation was to determine the properties of an item factor analysis model for ordinal response data that utilizes polychoric correlation matrices (Olsson, 1979). In particular, these authors were interested in evaluating two types of estimation criteria for empirical applications where the underlying latent variables were, or were not, distributed multivariate normal. The estimation criteria under investigation were the weighted least squares (WLS) and a robust variant of the WLS where only the diagonal of the associated weight matrix is used (termed diagonally weighted least squares; DWLS) with a mean-variance correction to the $\chi^2$ goodness-of-fit statistic (termed WLSMV). In their simulation, Flora and Curran investigated the effects of various latent variable distribution shapes, the number of observed response categories, sample size, and model complexity (i.e., number of latent variables and observed indicator variables). The analysis of their simulated data focused on the behaviour of the (scaled) $\chi^2$ goodness-of-fit test statistic, recovery of the population parameters, and the consistency of the standard error estimates. Finally, simulation results were summarized using the relative bias meta-statistic, the distributional properties of the (scaled) $\chi^2$ statistic relative to the model's degrees of freedom, and the performance of the standard error estimates relative to the standard deviation of the parameter estimates across the independent replications.

Flora and Curran's (2004) simulation was adopted in this section because it contains a number of practically difficult coding and implementation details, and therefore provides a rich and sufficiently complex experimental design to study. Specifically, their simulation:

- Involves different population generating model structures (e.g., a one- versus two-factor model with five or ten indicator variables each), implying that the number of properties to study in the analysis steps (such as number of model parameters) varies across the simulation conditions,
- Requires specialized estimation software suitable for structural equation modeling (e.g., `Mplus`, or in this section the `lavaan` package), largely requiring the user to conform to a predefined syntax convention set by the software,
- Implements specialized algorithms for generating continuous variables (e.g., Vale & Maurelli, 1983) as well as manual transformations of these generated variables to create categorical truncations,
- Is a partially-crossed simulation design due to the removal of unstable factor combinations (details below), which if not known a priori would result in crashes during the execution of the simulation experiment,
- Utilizes different estimation criteria (WLS and DWLS) by way of iterative numerical algorithms that are not guaranteed to converge, and

[15] The `seeds` object is actually stored as `tibble`, so indexing with the `$` operator is also possible and generally preferable when using auto-completion features in IDEs such as RStudio.

[16] See `SimDesign`'s on-line wiki regarding how to call external software, such as `Mplus`, in `Analyse()` for single and multi-core applications.

**Listing 10** ■ Definition of SimDesign's Design object by way of the `createDesign()` function for replicating Flora and Curran's (2004) simulation study

```
library (SimDesign)
Design <- createDesign(N = c(100, 200, 500, 1000),
                       categories = c(2, 5),
                       skewness_kurtosis = list( c(0, 0), c(.75, 1.75), c(.75, 3.75),
                                                 c(1.25, 1.75), c(1.25, 3.75)),
                       factors = c(1, 2),
                       indicators = c(5, 10),
                       estimator = c('WLSMV', 'WLS'),
                       # remove known problematic conditions
                       subset = !(estimator == 'WLS' & N %in% c(100, 200) &
                                  factors == 2 & indicators == 10))
Design

## # A tibble: 300 x 6
##        N categories skewness_kurtosis factors indicators estimator
##    <dbl>      <dbl> <lst>               <dbl>      <dbl> <chr>
##  1   100          2 [0, 0]                  1          5 WLSMV
##  2   200          2 [0, 0]                  1          5 WLSMV
##  3   500          2 [0, 0]                  1          5 WLSMV
##  4  1000          2 [0, 0]                  1          5 WLSMV
##  5   100          5 [0, 0]                  1          5 WLSMV
##  6   200          5 [0, 0]                  1          5 WLSMV
##  7   500          5 [0, 0]                  1          5 WLSMV
##  8  1000          5 [0, 0]                  1          5 WLSMV
##  9   100          2 [0.75, 1.75]            1          5 WLSMV
## 10   200          2 [0.75, 1.75]            1          5 WLSMV
## # ... with 290 more rows

# include syntax generation function
source ('FloraCurran2004-functions.R')
```

- Collects heterogeneous analysis information to be summarised in the form of goodness-of-fit statistics, standard errors, and (marginal) parameter estimates.

Taken together, this particular MCS experiment is wrought with potential pitfalls that require a great deal of care early on by the investigator. Below provides one such implementation using `SimDesign`, and includes discussions regarding how the process of the final code came to fruition. Despite the fact that this simulation is more complicated than the previous mediation analysis, the steps required to begin coding this MCS in `SimDesign` is no different, and by-and-large poses little to no difficulty for the package to efficiently manage.

Before beginning, it is important to highlight that, in addition to using different software for data generation and statistical analyses, there are a few subtle differences between the following simulation code and the study by Flora and Curran (2004). Specifically, although the DWLS estimator was used for the robust version of the WLS estimator, the value of the degrees of freedom in the following simulation were not estimated from the data. Instead, the $\chi^2$ goodness-of-fit statistic with a mean-variance adjustment was applied (i.e., WLSMV), and therefore the expected value of the scaled $\chi^2$ distribution will theoretically equal the model degrees of freedom (as is the case for the WLS as well). Additionally, in situations where a model failed to converge a new dataset was automatically redrawn until the requested number of replications were collected (in this case, $R = 500$). This differs from Flora and Curran's study in that only a fixed number of datasets were originally generated and analyzed, where datasets with non-converged results were discarded from their summaries (D. Flora, personal communication, January 6, 2020); hence, in their original article, reported table cells with higher rates of non-convergence have greater sampling variability than cells with lower rates of non-

**Listing 11** ■ Construction of the `Generate()` function

```
Generate <- function(condition, fixed_objects = NULL) {
    Attach(condition)
    syntax <- genLavaanSyntax(factors=factors, indicators=indicators)
    cdat <- simulateData(syntax, model.type='cfa', sample.nobs=N,
                         skewness=skewness_kurtosis[1L],
                         kurtosis=skewness_kurtosis[2L])
    tau <- if(categories == 5)
           c(-1.645, -0.643, 0.643, 1.645) else 0
    # data generation fix described in Flora's (2002) unpublished dissertation
    if (categories == 5 && all(skewness_kurtosis == c(1.25, 1.75)))
        tau[1] <- -1.125
    dat <- apply(cdat, 2, function(x, tau){
        dat <- numeric(length(x))
        for (i in 1:length(tau))
            dat[x > tau[i]] <- i
        dat
    }, tau=tau)
    # throw error if number of categories not correct
    if (!all( apply(dat, 2, function(x) length(unique(x))) == categories))
        stop('Number of categories generated is incorrect')
    dat
}
```

convergence, while in the following implementation using `SimDesign` all simulation conditions are guaranteed to have $R = 500$ valid replication instances.

### *Implementation in `SimDesign`*

Beginning with the simulation factors, `Design` was constructed by crossing: the sample size ($N = 100, 200, 500, 1000$), number of observed categories per indicator (two versus five), five combinations of skewness and kurtosis (specified as a `list` containing five sets of two numbers to preserve the paired skewness-kurtosis information), number of latent factor variables (one or two), and number of indicator variables per unobserved variable (five or ten). Fully crossing these factors resulted in a design with 160 unique conditions. However, this does not completely reflect the organization of the object definition. Specifically, in our first implementation attempt of this `Design` object the factor `estimator` was not included, and instead the WLS and DWLS criteria were applied to the same generated dataset within the `Analyse()` step. Using the same data typically results in reduced systematic bias effects since the analyses are applied to identical data characteristics, and will often decrease the overall simulation times due to re-using the generated data. However, due to the high rates of non-convergence for the WLS criteria the two estimators were separated in the `Design` object as an additional simulation factor so that only one of the estimators would be used at a time, and so that a clearer picture of the non-convergence rates could be obtained for each estimator in isolation.

Additionally, in our first coding attempt the `subset` argument was not supplied to `createDesign()`, which resulted in a number of fatal crashes for a selection of row combinations in `Design` (see Flora & Curran, 2004, for similiar observations). However, in this attempt `runSimulation()` did not immediately terminate upon encountering these fatal errors, and instead: returned informative warning messages that were immediately printed to the console; `NA` placeholders were provided in the final simulation object to indicate that the desired results could not be computed, and; a `character` vector column labeled `FATAL_TERMINATION` in the final simulation object was included pertaining to the last observed error message for the problematic row conditions. Hence, even when evaluating highly problematic simulation conditions, and whether the investigator anticipates these problems beforehand or not, `SimDesign` will gracefully manage these issues and inform the investigator of any fatal errors so that these problems can be inspected at a later time. That said, due to Flora and Curran's prior report and our own coding implementation it was clear that there were a high number of non-converged results for the $N = 100$ and $N = 200$ sample sizes when using the WLS method for models with 10 indicators and two factors. In

**Listing 12** ■ Construction of the `Analyse()` function

```
Analyse <- function(condition, dat, fixed_objects = NULL) {
    Attach(condition)
    syntax <- genLavaanSyntax(factors=factors, indicators=indicators, analyse=TRUE)
    mod <- cfa(syntax, dat, ordered = colnames(dat), estimator=estimator)
    # check that model and coefficients are reasonable
    if (!lavInspect(mod, 'converged')) stop('Model did not converge')
    pick_lambdas <- matrix(TRUE, indicators*factors, factors)
    if(factors == 2)
        pick_lambdas[(indicators+1):(indicators*3)] <- FALSE
    cfs <- lavInspect(mod, what="std")$lambda[pick_lambdas]
    if(any(cfs > 1 | cfs < -1))
        stop('Model contains Heywood cases')
    if(factors > 2 && abs(lavInspect(mod, what="std")$psi[2,1]) >= 1)
        stop('Latent variable psi matrix not positive definite')

    # extract desired results
    fit <- fitMeasures(mod)
    ses <- lavInspect(mod, what="se")$lambda[pick_lambdas]
    stat_names <- extract_stats <- c('chisq', 'df', 'pvalue')
    if(estimator == 'WLSMV')
        extract_stats <- paste0(extract_stats, '.scaled')
    fitstats <- fit[extract_stats]
    names (fitstats) <- stat_names
    phi21 <- if(factors == 2)
    lavInspect(mod, what="std")$psi[1,2] else NULL
    ret <- c(fitstats, mean_ses= mean(ses), lambda=cfs, phi21=phi21)
    ret
}
```

the following, these problematic conditions were removed from the simulation a priori by utilizing the `subset` argument in `createDesign()`.

Next, because the `lavaan` package requires a specific form of a `character` vector as an input to specify the structure of the model, generation of such syntax is required for each simulation condition combination. For consistency with the program, `lavaan` also supports supplying a syntax input for generating data, which utilizes the Vale and Maurelli (1983) algorithm to generate non-normal continuous variables. Both the analysis and data generation syntax format were written as a stand-alone user-defined function called `genLavaanSyntax()`, which is a three argument function defined in Appendix C. This function returns the associated `lavaan` syntax given the number of factor (`factors`) and indicator (`indicators`) variables, and whether the function is to be used for data generation (`analysis = FALSE`) or analysis (`analysis = TRUE`). To avoid needless clutter in the current presentation, the function is instead sourced into R using `source()` after having been saved to file called "`FloraCurran2004-functions.R`".

Following the structural definition of the MCS and user-defined functions, we now turn to `Generate()`. For this simulation the desired object to return is a `data.frame` because this is the type of object required by `lavaan`. To simulate the continuous data the function `simulateData()` is used, which is exported from the `lavaan` package. However, rather than using the `::` operator to locate this function (as was used, for example, in Appendix ) the `lavaan` package is attached to the R session by passing `runSimulation(..., packages = "lavaan")`, which automatically distributes the package's exported functions to each computing node whenever the simulation is executed using parallel computing architecture. Next, the continuous data in the object `cdat` is categorized according to the threshold values defined in the `tau` vector to create either two- or five-category response data by way of an `apply()` function. Finally, to ensure that the number of categories created was as expected for each observed variable, a conditional `if()` test was in-

**Listing 13** ■ Construction of the `Summarise()` function

```
Summarise <- function(condition, results, fixed_objects = NULL) {
    # model parameters
    lambdas <- results[ , grepl('lambda', colnames(results))]
    pool_mean_lambdas <- mean(apply(lambdas, 2, mean)) # Equation 10 in F&C (2004)
    pool_SD_lambdas <- sqrt(mean( apply(lambdas, 2, var))) # Equation 11 in F&C
    (2004)
    RB_phi21 <- if (condition$factors == 2)
        bias(results$phi21, parameter=.3, type='relative', percent=TRUE) else NULL
    mean_se <- mean(results$mean_ses)

    # goodness-of-fit
    edr_05 <- EDR(results$pvalue, alpha = .05)
    mean_X2 <- mean(results$chisq)
    sd_X2 <- sd(results$chisq)
    RB_X2 <- bias(results$chisq, parameter=results$df, type='relative',
                  percent=TRUE, unname=TRUE)
    ret <- c(mean_X2=mean_X2, sd_X2=sd_X2, edr_05=edr_05,
             pool_mean_lambdas=pool_mean_lambdas,
             pool_SD_lambdas=pool_SD_lambdas, mean_se=mean_se,
             RB_X2=RB_X2, RB_phi21=RB_phi21)
    ret
}
```

cluded with an associated `stop()` call. If this logical test fails then an error is raised, recorded, and `Generate()` is called again to draw a new dataset.

Moving on to `Analyse()`, the user-defined `genLavaanSyntax()` function is again called, however this time the argument `analyse = TRUE` is passed to trigger the generation of a syntax suitable for estimation purposes. This syntax, along with the generated dataset `dat`, are supplied to `lavaan`'s function `cfa()` to estimate an item factor analysis model with ordinal response data (specified by the `ordered` argument) given some desired estimation criteria (`estimator`). After the model has been fitted, a number of checks are performed to ensure the quality of the model output; specifically, `lavInspect()` is used to extract whether the model converged successfully (otherwise, an error is thrown and the data are redrawn), whether the slope coefficient estimates are within $[-1, 1]$ (otherwise so-called “Heywood” cases are present), and for two-factor models a check is made regarding whether the absolute value of correlation estimate is larger than 1 (in which case the correlation matrix between the latent variables is not positive definite).

Following the estimation and quality checks of the fitted models, the last portion of `Analyse()` extracts the statistical information that is to be summarised across the independent replications. In Flora and Curran's 2004 simulation their interest was mainly in the properties of the (scaled) $\chi^2$ values and their associated $p$-values, the estimates of the factor loadings and correlation between the latent variables, and the general behaviour of the estimated standard errors relative to the standard deviation of the factor loadings across replications (of which the standard deviations and standard errors should be of similar magnitude). As such, all of the factor loadings (`lambda`) are returned from `Analyse()` so that their respective standard deviations can be obtained across all $R$ replications, and the average of the standard errors within a given replication were returned and later averaged in the `Summarise()` function. As well, for the single factor models the latent correlation estimate (`phi21`) is not applicable, and therefore `phi21` is set to a `NULL` placeholder in this situation to effectively drop it from the returned object. Notice here that with `SimDesign` the lengths of the returned objects from `Analyse()` (and also `Summarise()`) are allowed to differ across the simulation conditions, where in the final simulation object a set of `NA` placeholders will appear for elements that are not applicable to a given row condition.

Finally, `Summarise()` is defined to summarise the overall behaviour of the analysis properties within each unique row-condition in `Design`. This function first extracts all columns from the `results` object to create a more manageable subset called `lambdas`, which allows easier computations of the respective average estimates

**Listing 14** ■ Execution of Flora and Curran (2004) simulation through `runSimulation()`.

```
res <- runSimulation(design=Design, replications=500, generate=Generate,
                     analyse=Analyse, summarise=Summarise, max_errors=100,
                     packages='lavaan', parallel=TRUE,
                     filename='FloraCurran2004', save_results=TRUE)
res

## # A tibble: 300 x 20
##        N categories skewness_kurtos~ factors indicators estimator mean_X2 sd_X2
##    <dbl>      <dbl> <lst>              <dbl>      <dbl> <chr>       <dbl> <dbl
##  1   100          2 [0, 0]                 1          5 WLSMV        4.79  2.77
##  2   200          2 [0, 0]                 1          5 WLSMV        5.13  3.15
##  3   500          2 [0, 0]                 1          5 WLSMV        4.92  3.01
##  4  1000          2 [0, 0]                 1          5 WLSMV        5.13  3.31
##  5   100          5 [0, 0]                 1          5 WLSMV        5.12  3.35
##  6   200          5 [0, 0]                 1          5 WLSMV        5.05  3.02
##  7   500          5 [0, 0]                 1          5 WLSMV        4.97  3.16
##  8  1000          5 [0, 0]                 1          5 WLSMV        4.94  3.02
##  9   100          2 [0.75, 1.75]           1          5 WLSMV        4.85  2.79
## 10   200          2 [0.75, 1.75]           1          5 WLSMV        4.97  3.15
## # ... with 290 more rows, and 12 more variables: RB_X2 <dbl>, edr_05 <dbl>,
## #   pool_mean_lambdas <dbl>, pool_SD_lambdas <dbl>, mean_se <dbl>,
## #   RB_phi21 <dbl>, REPLICATIONS <int>, SIM_TIME <dbl>, COMPLETED <chr>,
## #   SEED <int>, ERRORS <int>, WARNINGS <int>
```

and the to-be-averaged variability terms (see Equations 10 and 11 in Flora & Curran, 2004, p. 474). The average of the standard errors across replications is also obtained, as well as the relative bias (as a `percent`) of the latent variable correlation estimate by using `SimDesign`'s built-in `bias()` function. Finally, the mean, standard deviation, and relative bias of the $\chi^2$ statistic given the respective degrees of freedom are obtained[17], and the empirical detection rates given $\alpha = .05$ are included.

### *Simulation Results*

Although most of the defaults arguments provided by `SimFunctions()` are useful out-of-the-box, for more complex MCSs the arguments to `runSimulation()` should generally be modified to accommodate the added complexity and computational intensity. In the following execution, the number of independent replications for each condition was reduced to $R = 500$ to match Flora and Curran (2004), the `lavaan` package was attached so that functions such as `simulateData()` and `cfa()` are available across nodes, the code is executed in `parallel` using all available cores, temporary simulation files are generated in case of unexpected crashes (`save = TRUE`; the default), the completed simulation object is saved to a file called "`FloraCurran2004.rds`", the number of allowable consecutive errors to occur was increased to 100 from 50 (`max_errors = 100`) to accommodate for the frequency of non-converging models in some conditions (see below), and finally the analysis results and associated information for each row-condition in `Design` are saved into a sub-directory for future reference and potential re-summarising (`save_results = TRUE`; see also `help(reSummarise)` for further details about re-summarizing the stored replication information).

`runSimulation()` was executed on a 48-core (2.6 GHz) Linux OS running Ubuntu 18.04.3 LTS, R version 3.6.1, and required just under one hour to complete. As before, further information regarding the R session and computer information can be obtained using `summary(res)`. After executing this R script, a number of error and warning messages were observed, where the frequency of the errors and warnings are printed in the `ERRORS` and `WARNINGS` columns. For ease of inspection, conditions with `ERRORS` greater than 500 (i.e., have a probability of convergence less than $\frac{500}{500+500} = 1/2$) are printed in Listing 15, where columns are selected using the `dplyr`

[17]Note the use of `bias(..., unname = TRUE)` when computing the relative bias for the $\chi^2$ statistic. This is included to remove any of R's labeling information that was carried over from the original object definitions so that the resulting object does not inherent confusing names in the subsequent computations.

**Listing 15** ■ Design conditions where the number of error messages raised was greater than 500.

```
library(dplyr)

# subset of res containing many error messages
res %>% select(N:estimator, ERRORS) %>%
    filter (ERRORS > 500)

## # A tibble: 11 x 7
##          N categories skewness_kurtosis factors indicators estimator ERRORS
##      <dbl>      <dbl> <lst>                <dbl>      <dbl> <chr>     <int
##  1   100          5 [1.25, 3.75]             1          5 WLSMV       681
##  2   100          5 [1.25, 3.75]             2          5 WLSMV       800
##  3   100          5 [1.25, 1.75]             1         10 WLSMV       777
##  4   100          5 [1.25, 3.75]             1         10 WLSMV      1803
##  5   100          5 [1.25, 1.75]             2         10 WLSMV       678
##  6   100          5 [1.25, 3.75]             2         10 WLSMV      2106
##  7   100          5 [1.25, 3.75]             1          5 WLS         999
##  8   100          5 [1.25, 3.75]             2          5 WLS        1114
##  9   100          5 [0.75, 1.75]             1         10 WLS         506
## 10   100          5 [1.25, 1.75]             1         10 WLS         837
## 11   100          5 [1.25, 3.75]             1         10 WLS        4753
```

package's column (`select()`) and row (`filter()`) subsetting functions via the pipe operator (`%>%`). As is clear from the output, models fitted with $N = 100$ and only five indicators are much less likely to converge when combined with non-zero skewness and kurtosis values for the latent variable distribution definitions when using the WLS and DWLS estimators, where the lowest convergence rate occurred in the most extreme distribution combination for the WLS estimator (with an estimated probability of convergence of $\frac{500}{500+4753} \approx .095$). After extracting the specific error messages by viewing the column names of the object `errors <- SimExtract(res, what = "errors")`, many of the error messages appeared to pertain to Heywood cases and categorical data generation issues, while most of the warning messages related to model estimation issues where the correlation matrix for the latent variables became non-positive definite during `lavaan`'s iterative parameter search.

At this point, summarizing the results from this simulation experiment becomes important to capture the meaningful variability in the results. For example, comparing the marginal empirical $p$-value estimates may be of interest to investigators, which can be computed using the verbs from the `dplyr` package. The following demonstrates two particular marginal results: averaging the empirical $p$-values over all combinations of the estimators and sample sizes, and again using these two experimental factors with the number of latent factors. The code in Listing 16 utilizes `dplyr`'s `group_by()` and `summarise()` verbs to create marginal tables of results. Of course, other data exploration techniques certainly could and should be investigated for these data as well, such as generating graphical representations by way of expressive packages such as `ggplot2` (Wickham, 2009), or by performing ANOVA-based decompositions of the meta-statistics to determine which experimental simulation factors (and the interaction of these factors) result in the largest observed differences; see `SimDesign::SimAnova()` for a brief demonstration of this type of post-analysis ANOVA exploration.

## *Reproduced Table and Figure*

Given the results from the above simulation study it is possible to quickly reproduce many of the tables and figures presented in Flora and Curran (2004). For instance, the pipe operator and additional data manipulation verbs from the `dplyr` package can be used to: add column variables (`mutate`), rename columns (`rename`), merge distinct objects by columns (`full_join`), and sort by row (`arrange`). Given these additional verbs, the R code in Listing 17 provides a replication of the information displayed in Table 5 of Flora and Curran (2004), p. 480.[18]

[18] In the following code, if users are familiar with the `purrr` package for extracting elements from `list`s then the `sapply()` calls could instead be replaced with `s = map_dbl(res$skewness_kurtosis, 1)` and `k = map_dbl(res$skewness_kurtosis, 2)`, respectively.

**Listing 16** ■ Marginal summary statistics for Flora and Curran's (2004) simulation experiment as implemented in SimDesign.

```
# marginalized detection rates given the estimator and sample size
res %>% group_by (estimator, N) %>%
    summarise (EDR = mean (edr_05))

## # A tibble: 8 x 3
## # A tibble: 8 x 3
## # Groups:   estimator [2]
##   estimator     N    EDR
##   <chr>     <dbl>  <dbl>
## 1 WLS         100 0.382
## 2 WLS         200 0.217
## 3 WLS         500 0.324
## 4 WLS        1000 0.214
## 5 WLSMV       100 0.0694
## 6 WLSMV       200 0.0570
## 7 WLSMV       500 0.0520
## 8 WLSMV      1000 0.0522

# marginalized detection rates given the estimator, factors, and sample size
res %>% group_by (estimator, factors, N)
    %>% summarise (EDR = mean (edr_05))

## # A tibble: 16 x 4
## # Groups:   estimator, factors [4]
##    estimator factors     N    EDR
##    <chr>       <dbl> <dbl>  <dbl>
##  1 WLS             1   100 0.250
##  2 WLS             1   200 0.151
##  3 WLS             1   500 0.0868
##  4 WLS             1  1000 0.0644
##  5 WLS             2   100 0.646
##  6 WLS             2   200 0.350
##  7 WLS             2   500 0.561
##  8 WLS             2  1000 0.364
##  9 WLSMV           1   100 0.0529
## 10 WLSMV           1   200 0.0487
## 11 WLSMV           1   500 0.0507
## 12 WLSMV           1  1000 0.0504
## 13 WLSMV           2   100 0.086
## 14 WLSMV           2   200 0.0654
## 15 WLSMV           2   500 0.0534
## 16 WLSMV           2  1000 0.0539
```

If investigators intend to use packages such as knitr (Xie, 2015) to dynamically prepare their documents using R objects then this table could be rendered into a publication ready table via packages such as xtable, pander, stargazer, and so on. Alternatively, if the investigator wishes to convert this information into a table manually then saving the information to an external text file via functions such as write.table() or write.csv() will provide a sufficient means to access the data (see the readr package for similar functions involving tibbles). Finally, as a practical example of building graphics using the results from SimDesign, the code in Listing 18 creates a comparable replication (Figure 2) of Figure 6 in Flora and Curran (2004), p. 483.

## Discussion

Sigal and Chalmers (2016) initially presented an overview of the SimDesign package for the purpose of demonstrating an intuitive pedagogical approach to learning and writing Monte Carlo simulation studies. However, the purpose of the SimDesign package is, and has always been, to provide a structure for optimally controlling real-world simulation experiments in a safe, flexible, intuitive, and efficient manner, where pedagogical applications are one fruitful byproduct. The purpose of this tutorial was to provide a detailed discussion of several coding features that ought to appear in many simulation experiments, emphasize attractive coding practices when using scripting lan-

**Listing 17** ■ Code to replicate Table 5 in Flora and Curran (2004).

```
# separate s and k values in skewness_kurtosis, create % variable,
# select only 5 categories, rename columns to match
res %>% mutate(s = sapply(res$skewness_kurtosis, function(x) x[1]),
               k = sapply(res$skewness_kurtosis, function(x) x[2]),
               "% reject" = edr_05 * 100) %>%
    filter(categories == 5) %>%
    rename(M='mean_X2', SD='sd_X2', RB='RB_X2') -> res_5

# construct Table 5 information from Flora and Curran (2004)
res_5 %>%
    filter(indicators == 10, factors == 1, estimator == 'WLS') %>%
    select(N, s, k, M, SD, RB, "% reject") -> WLS
res_5 %>%
    filter(indicators == 10, factors == 1, estimator == 'WLSMV') %>%
    select(N, s, k, RB, "% reject") -> WLSMV
full_join(WLS, WLSMV, by = c("N", "s", "k"), suffix= c('', '.DWLS')) %>%
    arrange(N, s, k) -> tab5
tab5

## # A tibble: 20 x 9
##        N     s     k     M    SD    RB `% reject` RB.DWLS `% reject.DWLS`
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>      <dbl>   <dbl>           <dbl>
##  1   100  0     0     57.8 18.7  65.2        62.4   6.37             7.
##  2   100  0.75  1.75  58.7 19.0  67.7        67.2   6.67             8.80
##  3   100  0.75  3.75  61.3 19.2  75.0        70.2   5.61             7.8
##  4   100  1.25  1.75  65.9 23.5  88.4        74.2  11.1             11
##  5   100  1.25  3.75  56.5 17.6  61.3        60.2   4.08             5
##  6   200  0     0     43.7 12.3  25.0        26.6   3.07             4.40
##  7   200  0.75  1.75  44.5 11.8  27.0        30.4   3.09             4.2
##  8   200  0.75  3.75  45.0 12.1  28.6        31.6   2.68             6.2
##  9   200  1.25  1.75  48.5 14.3  38.5        41.8   9.67            10.8
## 10   200  1.25  3.75  45.2 12.0  29.0        29.6   2.34             6.4
## 11   500  0     0     38.3  9.67  9.41       12.2  -0.449            3.4
## 12   500  0.75  1.75  38.4  9.68  9.69       12.6   1.62             4
## 13   500  0.75  3.75  38.3 10.2   9.36       13.8   0.737            5.2
## 14   500  1.25  1.75  41.6 10.3  19.0        20.8   6.83             9.6
## 15   500  1.25  3.75  38.2  9.49  9.23       10     1.09             4
## 16  1000  0     0     36.7  8.78  4.83        7.2   1.21             4.40
## 17  1000  0.75  1.75  36.5  8.79  4.27        7.6   0.898            6.8
## 18  1000  0.75  3.75  37.1  8.86  6.01        7.6   1.82             5.6
## 19  1000  1.25  1.75  38.8  9.23 10.9        12     7.00             7.2
## 20  1000  1.25  3.75  36.7  8.68  4.81        7.4   0.993            4.2
```

guages such as R, and to demonstrate how the `SimDesign` package can be used to naturally implement many important and desirable programming features when designing simulation experiments.

As with all discussions of software, the information presented herein is not intended to be exhaustive. Below are some additional features currently supported by `SimDesign` that interested readers may also wish to explore:

- In situations where more replications are required after the original simulation has been completed (e.g., at the request of an anonymous reviewer) then the additional replications requested in a new execution, combined with the use of `aggregate_simulations()`, can combine independent executions of the same simulation code into a single simulation. For instance, if a simulation was executed initially with $R = 500$, and at a later date the same simulation was independently performed with another $R = 500$ replications, then `aggregate_simulations()` could be used to combine these two independent experiments into single simulation as though $R = 1000$ were originally evaluated,
- In a similar spirit to `aggregate_simulations()`, if additional *levels* of a factor variable (and therefore subsequent conditions) were added to the `Design` object at a later date (e.g., adding $N = 5000$ to the Flora and Curran (2004) simulation above) then only the new row combinations in the `Design` object would need to be evaluated, and can be combined row-wise via the S3

**Listing 18** ■ Code to replicate Table 6 in Flora and Curran (2004).

```
library(ggplot2)

# generate summary data for Figure 6 in Flora and Curran (2004)
results_5 %>%
    filter(indicators == 5) %>%
    mutate(TypeI = edr_05 * 100,
    est_fact = factor(estimator):factor(factors)) %>%
    group_by(N, est_fact) %>%
    summarise(mean_TypeI = mean(TypeI)) -> fig6dat

labels <- c("Model 1 (full WLS estimation)", "Model 3 (full WLS estimation)",
            "Model 1 (robust WLS estimation)", "Model 3 (robust WLS estimation)")

# draw graphic using the ggplot2 package
ggplot(fig6dat,
       aes (x= factor(N), y=mean_TypeI, linetype=est_fact,
            shape=est_fact, group=est_fact)) +
       geom_line() + geom_point(size=4) +
       geom_hline(yintercept=5, colour='red', linetype='dashed') +
       xlab("Sample Size") + ylab("Type I Error Rate") +
       scale_linetype_discrete(labels=labels) +
       scale_shape_discrete(labels=labels) +
       ylim(0, 80) + theme_bw() +
       theme(legend.title= element_blank(),
             legend.position = c(.8, .7),
             axis.line = element_line(colour = "black"),
             panel.grid.major = element_blank(),
             panel.grid.minor = element_blank(),
             panel.border = element_blank(),
             panel.background = element_blank()
       )
]
```

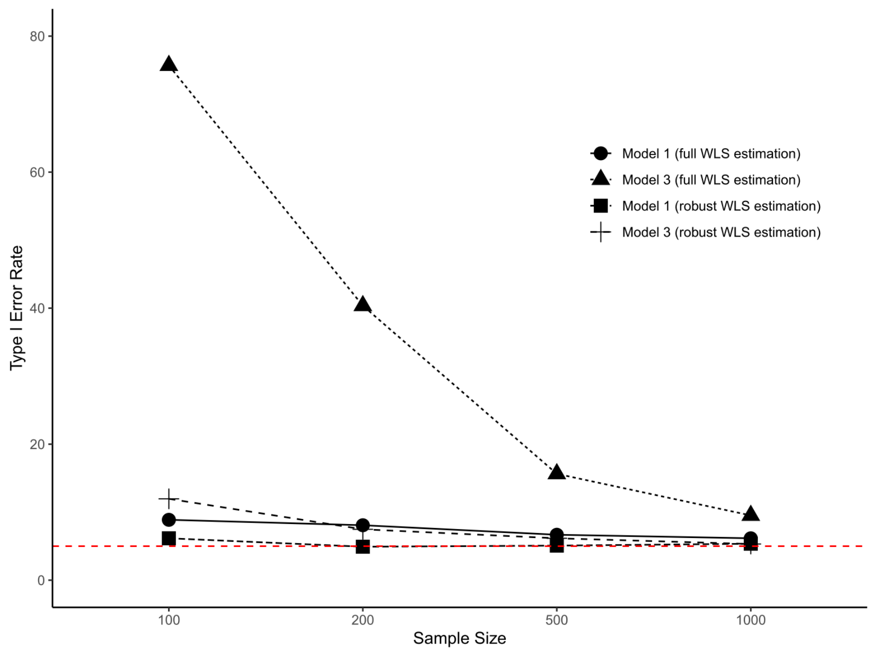
method `rbind()`,

- Additional data generation algorithms are included in the package, such as `rValeMaurelli()` and `rHeadrick()` to create multivariate non-normal distributions with specified skewness and kurtosis (Vale & Maurelli, 1983; Headrick, 2002), `rinvWishart()` for the inverse Wishart distribution, `rmgh()` for the multivariate $g$-and-$h$ distribution, `rmvt()` and `rmvnorm()` for the multivariate $t$ and Gaussian distribution, and `rejectionSampling()` to draw data from complex distributions via rejection sampling. See `extraDistr` (Wolodzko, 2019) for additional optimized distribution functions outside base R,
- Estimating the sampling variability of the simulation's meta-statistical results within each condition can be obtained via non-parametric bootstrapped confidence intervals (e.g., `runSimulation(..., boot_method = "basic")`, which may also be obtained at a later time via similar arguments to `reSummarise()` if `save_results = TRUE` were originally supplied to `runSimulation()`). This can used to gauge the sampling precision of the MCS estimates, and whether additional replications should be performed to reduce the sampling uncertainty in the obtained meta-statistics (see the first bullet point above),
- A means for creating self-contained parametric bootstrapping functions (e.g., Chalmers & Ng, 2017) is also possible, which comes complete with all of `SimDesign`'s built-in safety features, and
- A function called `SimShiny()` can be used to automatically create template files for interactive simulations via the `shiny` package. This allows investigators to publish their simulation's coding structure on-line using a GUI presentation format with minimal effort, and allows for full customization of the GUI output.

These and other features are included to streamline the creation and evaluation of MCSs, and to defensively anticipate many of the common stressors that appear throughout the process of designing, writing, and publishing simulation studies.

Finally, it is worth noting that the R programming environment is not without its share of third-party software packages for designing Monte Carlo simulation experiments (e.g., `simFrame` (Alfons, Templ, & Filzmoser, 2010), `simsalapar` (Hofert & Mächler,

**Figure 2** ■ Replication of Figure 6 in Flora and Curran (2004).



2016), simulator (Bien, 2016), ezsim (Chan, 2014), MonteCarlo (Leschinski, 2019), simstudy (Goldfeld, 2020), and more). However, while many of these available packages share similar characteristics, the overall philosophy of code safety, efficiency, flexibility, and readability are not strongly emphasized in these packages. SimDesign, on the other hand, aims at each of these important and desirable criteria. This is particularly important given the current research climate where open science practices are at a premium, whereby providing openly available MCS code (e.g., via online appendices or repositories) reflects an important consistency with the Open Science Framework's Transparency and Openness Protocol (Nosek et al., 2015). Although providing MCS code is often encouraged by journal editors and reviewers, this step alone is not sufficient for future investigators to ascertain the veracity of a simulation study. Fortunately, SimDesign makes publishing code more feasible, where future readers will require less effort in understanding and trusting the MCS code and subsequent results.

## Authors' note

Special thanks to James Mallari, Kevin Hallgren, and Alexandre Gellen-Kamel for providing constructive comments on earlier versions of this manuscript. Correspondence concerning this article should be addressed to R. Philip Chalmers, Department of Psychology, York University, 4700 Keele Street, Toronto, ON, M3J 1P3, Canada.

## References

Alfons, A., Templ, M., & Filzmoser, P. (2010). An object-oriented framework for statistical simulation: The R package simFrame. *Journal of Statistical Software*, *37*(3), 1–36. Retrieved from http://www.jstatsoft.org/v37/i03/

Bien, J. (2016). The simulator: An engine to streamline simulations. *Submitted*. Retrieved from http://faculty.bscb.cornell.edu/~bien/simulator.pdf

Burton, A., Altman, D. G., Royston, P., & Holder, R. L. (2006). The design of simulation studies in medical statistics. *Statistics in Medicine*, *25*, 4279–4292. doi:10.1002/sim.2673

Chalmers, R. P. (2012). mirt: A multidimensional item response theory package for the R environment. *Journal of Statistical Software*, *48*(6), 1–29. doi:10.18637/jss.v048.i06

Chalmers, R. P. (2020). *SimDesign: Structure for Organizing Monte Carlo Simulation Designs*. R package version 2.1. Retrieved from https://CRAN.R-project.org/package=SimDesign

Chalmers, R. P., & Ng, V. (2017). Plausible-value imputation statistics for detecting item misfit. *Applied Psychological Measurement*, *41*(5), 372–387. doi:10.1177/0146621617692079

Chan, T. J. (2014). *ezsim: Provide an easy to use framework to conduct simulation*. R package version 0.5.5. Retrieved from https://CRAN.R-project.org/package=ezsim

Flora, D. B., & Curran, P. J. (2004). An empirical evaluation of alternative methods of estimation for confirmatory factor analysis with ordinal data. *Psychological Methods*, *9*(4), 466–491. doi:10.1037/1082-989X.9.4.466

Goldfeld, K. (2020). Simstudy: Simulation of study data (Version R package version 0.1.16). Retrieved from https://CRAN.R-project.org/package=simstudy

Hallgren, K. A. (2013). Conducting simulation studies in the R programming environment. *Tutorials in Quantitative Methods for Psychology*, *9*(2), 43–60. doi:10.20982/tqmp.09.2.p043

Headrick, T. C. (2002). Fast fifth-order polynomial transforms for generating univariate and multivariate nonnormal distributions. *Computational Statistics and Data Analysis*, *40*, 685–711. doi:https://doi.org/10.1016/S0167-9473(02)00072-5

Hoaglin, D. C., & Andrews, D. F. (1975). The reporting of computation-based results in statistics. *The American Statistician*, *29*(3), 122–126.

Hofert, M., & Mächler, M. (2016). Parallel and other simulations in R made easy: An end-to-end study. *Journal of Statistical Software*, *69*(4), 1–44. doi:10.18637/jss.v069.i04

Jones, O., Maillardet, R., & Robinson, A. (2014). *Introduction to Scientific Programming and Simulation Using R*. CRC Press.

Lee, S., Sriutaisuk, S., & Kim, H. (2019). Using the tidyverse package in R for simulation studies in SEM. *Structural Equation Modeling: A Multidisciplinary Journal*. doi:https://doi.org/10.1080/10705511.2019.1644515

Leschinski, C. H. (2019). *MonteCarlo: Automatic parallelized monte carlo simulations*. R package version 1.0.6. Retrieved from https://CRAN.R-project.org/package=MonteCarlo

Mooney, C. Z. (1997). *Monte carlo simulations*. Thousand Oaks, CA: Sage.

Morris, T. P., White, I. R., & Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, *38*(11), 2074–2102. doi:https://doi.org/10.1002/sim.8086

Muthén, B. O. (1984). A general structural equation model with dichotomous, ordered categorical, and continuous latent variable indicators. *Psychometrika*, *49*, 115–132.

Muthén, L. K., & Muthén, B. O. (2008). Mplus (Version 5.0) [Computer Program]. Author.

Nosek, B. A., Alter, G., Banks, G. C., Borsboom, D., Bowman, S. D., Breckler, S. J., ... Yarkoni, T. (2015). Promoting an open research culture: Author guidelines for journals could help to promote transparency, openness, andreproducibility. *Science*, *6242*(348), 1422–1425. doi:10.1126/science.aab2374

Olsson, U. (1979). Maximum likelihood estimation of the polychoric correlation coefficient. *Psychometrika*, *44*(4), 443–460.

Paxton, P., Curran, P., Bollen, K. A., Kirby, J., & Chen, F. (2001). Monte carlo experiments: Design and implementation. *Structural Equation Modeling*, *8*(2), 287–312. doi:https://doi.org/10.1207/S15328007SEM0802_7

Rosseel, Y. (2012). lavaan: An R package for structural equation modeling. *Journal of Statistical Software*, *48*(2), 1–36. Retrieved from http://www.jstatsoft.org/v48/i02

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

Sobel, M. E. (1986). Some new results on indirect effects and their standard errors in covariance structure. *Sociological Methodology*, *16*, 159–186.

Vale, C. D., & Maurelli, V. A. (1983). Simulating multivariate nonnormal distributions. *Psychometrika*, *48*, 465–471.

Wickham, H. (2009). *ggplot2: Elegant graphics for data analysis*. New York: Springer.

Wickham, H. (2019). *Advanced R* (2nd). Boca Raton: CRC Press.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., ... Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. doi:10.21105/joss.01686

Wolodzko, T. (2019). *extraDistr: Additional univariate and multivariate distributions*. R package version 1.8.11. Retrieved from https://CRAN.R-project.org/package=extraDistr

Xie, Y. (2015). *Dynamic documents with R and knitr*. Boca Raton, FL: CRC Press.

## Appendix A: Mediation Analysis Simulation

Simulation presented by Hallgren (2013), re-expressed using the `SimDesign` package's generate-analyse-summarise functional framework. To execute this simulation, copy and paste the following into an active R console.

```
# Generate and edit the R template file 'Hallgren2013.R'
# SimDesign::SimFunctions('Hallgren2013')
# function returns data.frame of p-values, estimates, SEs, etc
sobel_test <- function(X, M, Y){
    M_X <- lm(M ~ X)
    Y_XM <- lm(Y ~ X + M)
    a <- coefficients(M_X)[2] # extract from numeric vector
    b <- coefficients(Y_XM)[3]
    stdera <- summary(M_X)$coefficients[2,2] # extract from list first then matrix
    stderb <- summary(Y_XM)$coefficients[3,2]
    sobelz <- a*b / sqrt(b^2 * stdera^2 + a^2 * stderb^2)
    sobelp <- pnorm( abs(sobelz), lower.tail = FALSE)*2
    ret <- data.frame(a=a, SE_a=stdera, b=b, SE_b=stdera,
                      z=sobelz, p=sobelp)
    ret
}

#-----------------------------------------------------------------
library (SimDesign)

# fully-crossed simulation experiment
Design <- createDesign(N = c(100, 300),
                       a = c(-.3, 0, .3),
                       b = c(-.3, 0, .3),
                       cp = c(-.2, 0, .2))

#-----------------------------------------------------------------
Generate <- function(condition, fixed_objects = NULL ) {
    Attach(condition) # make N, a, b, and cp accessable
    X <- rnorm(N)
    M <- a*X + rnorm(N)
    Y <- cp*X + b*M + rnorm(N)
    dat <- data.frame(X=X, M=M, Y=Y)
    dat
}

Analyse <- function(condition, dat, fixed_objects = NULL ) {
    Attach(dat) # make objects X, M, and Y directly accessible
    sobel <- sobel_test(X=X, M=M, Y=Y)$p
    sobel_incorrect <- sobel_test(X=X, M=Y, Y=M)$p
    ret <- c(sobel=sobel, sobel_incorrect=sobel_incorrect)
    ret # named vector of p-values
}

Summarise <- function(condition, results, fixed_objects = NULL ) {
    ret <- EDR(results, alpha = .05) # results object is a 'data.frame'
    ret # empirical detection rate returned
}

#-----------------------------------------------------------------
res <- runSimulation(design=Design, replications=1000, generate=Generate,
                     analyse=Analyse, summarise=Summarise)
res
```

## Appendix B: Modifications to the Mediation Simulation Example

The following reflects one possible extension of the mediation simulation presented in Appendix B, which adds: an additional experimental factor to the simulation `Design` object; provides a conditional discrete-continuous data generation form for the $X$ variable; and adds a bias-corrected bootstrap mediation $p$-value test in the `Analyse()` code block. Due to the greater intensity of this simulation the final results are saved to a file called "`modified_mediation`", the internal `results` objects passed to `Summarise()` are all written to the hard-disk for each associated condition in the `Design` rows conditions by passing `save_results = TRUE`, and the simulation is executed in parallel using all available cores by passing `parallel=TRUE`. Note the similarities and differences to the simulation code in Appendix .

```
sobel_test <- function(X, M, Y){
    ... # defined in Appendix A
}

boot_test <- function(X, M, Y, bootstrap = 5000){
    dat <- data.frame(X=X, M=M, Y=Y)
    syntax <- "
               M ~ a * X
               Y ~ b * M + cp * X
               # indirect and total effects
               ab := a * b
               total := cp + ab"
    mod <- lavaan:: sem(syntax, data=dat, se="bootstrap", bootstrap=bootstrap)
    # bias-corrected bootstrap
    cfs <- lavaan::parameterEstimates(mod, boot.ci.type = "bca.simple")
    ret <- cfs[cfs$label == 'ab', ]
    ret
}

#-------------------------------------------------------------
library (SimDesign)
Design <- createDesign(N = c(100, 300),
                        a = c(-.3, 0, .3),
                        b = c(-.3, 0, .3),
                        cp = c(-.2, 0, .2),
                        dist = c('norm', 'bern'))

#-------------------------------------------------------------
Generate <- function(condition, fixed_objects = NULL) {
    Attach(condition)
    X <- if (dist == 'norm') rnorm (N) else rbinom (N, size=1, prob=.5)
    M <- a*X + rnorm(N)
    Y <- cp*X + b*M + rnorm(N)
    dat <- data.frame(X=X, M=M, Y=Y)
    dat
}

Analyse <- function(condition, dat, fixed_objects = NULL) {
    Attach(dat)
    sobel <- sobel_test(X=X, M=M, Y=Y)$p
    sobel_incorrect <- sobel_test(X=X, M=Y, Y=M)$p
    boot <- boot_test(X=X, M=M, Y=Y)$p
    boot_incorrect <- boot_test(X=X, M=Y, Y=M)$p
    ret <- c(sobel=sobel, sobel_incorrect=sobel_incorrect,
             boot=boot, boot_incorrect=boot_incorrect)
    ret
}

Summarise <- function(condition, results, fixed_objects = NULL) {
    ret <- EDR(results, alpha = .05)
    ret
}

#---------------------------------------------------------------
# save temporary files, analysis results, and execute code in parallel
res <- runSimulation(design=Design, replications=1000, generate=Generate,
                     analyse=Analyse, summarise=Summarise,
                     parallel=TRUE, save_results=TRUE,
                     filename='modified_mediation')
res
```

**Appendix C: User-defined functions required for Flora and Curran simulation**

The following code contains a user-defined convenience function for the (Flora & Curran, 2004) replication study with `SimDesign`, and should be saved to an external `.R` file (e.g., "`FloraCurran2004-functions.R`"). This function can be loaded into R via the `source()` function, which should be passed to the R console prior to executing `runSimulation()`.

```
#' @param J number of variables
#' @param analyse logical; is syntax being used for data generation or analysis?
#' @examples
#' cat(genLavaanSyntax(factors=1, indicators=10))
#' cat(genLavaanSyntax(factors=2, indicators=10))
#' cat(genLavaanSyntax(factors=1, indicators=10, analyse = TRUE))
#' cat(genLavaanSyntax(factors=2, indicators=10, analyse = TRUE))
#'
genLavaanSyntax <- function(factors, indicators, analyse = FALSE){
    ret <- if (factors == 1){
        if (analyse){
             paste0(paste0('f1 =~ NA*x1 + ', paste0( paste0('x', 2:indicators),
             collapse=' + ')), '\nf1 ~~ 1*f1')
        } else {
             paste0( paste0('f1 =~ ', paste0( rep (.7, indicators),
                                       paste0('*x', 1:indicators),
                                       collapse=' + '), ' \n'),
             paste0( sprintf('x%s ~~ 0.51*x%s', 1:indicators, 1:indicators),
             collapse=' \n'))
        }
    } else if (factors == 2){
        if (analyse){
            paste0( paste0('f1 =~ NA*x1 + ',
                    paste0( paste0('x', 2:indicators), collapse=' + ')),
                    paste0( sprintf('\nf2 =~ NA*x%s + ', indicators+1),
                            paste0( paste0('x', 2:indicators + indicators),
    collapse=' + ')),
          '\nf1 ~~ 1*f1 \nf2 ~~ 1*f2 \nf1 ~~ f2')
      } else {
          paste0( paste0('f1 =~ ', paste0( rep (.7, indicators),
                                            paste0('*x', 1:indicators),
                                            collapse=' + '), ' \n'),
                  paste0('f2 =~ ', paste0( rep (.7, indicators),
                                            paste0('*x', 1:indicators + indicators),
                                            collapse=' + '), ' \n'), 'f1 ~~ .3*f2 \n',
                   paste0( sprintf('x%s~~0.51*x%s', 1:indicators, 1:indicators),
                   collapse=' \n'))
      }
    } else stop ('factors input is incorrect')
    ret
}
```

## Citation

Chalmers, R. P., & Adkins, M. C. (2020). Writing effective and reliable Monte Carlo simulations with the SimDesign package. *The Quantitative Methods for Psychology*, *16*(4), 248–280. doi:10.20982/tqmp.16.4.p248

Received: 13/01/2020 ~ Accepted: 12/06/2020

Table 1 follows.

**Table 1** ■ Meta-statistics useful for summarizing the behavior of Monte Carlo simulation replicates in `SimDesign`'s `Summarise()` function. In this table $\theta$ represents a population parameter, $\hat{\theta}$ a sample estimate, $\widehat{SD}(\cdot)$ an estimate of the standard deviation, $\widehat{CI}(\cdot)$ an estimate of the confidence interval, and $\#$ a counting or tally operator.

| Function in `SimDesign` | Name | `type = "..."` | Definition |
|---|---|---|---|
| `bias(...)` | bias | `bias` (default) | $\frac{1}{R}\sum(\hat{\theta}_r - \theta)$ |
| | relative bias | `relative` | $\frac{1}{R}\sum_{r=1}^{R}\frac{(\hat{\theta}_r-\theta)}{\theta}$ |
| | relative absolute bias (RAB) | `abs_relative` | $\frac{1}{R}\sum_{r=1}^{R}\frac{(\hat{\theta}_r-\theta)}{\lvert\theta\rvert}$ |
| | standardized bias | `standardized` | $\frac{1}{R}\sum_{r=1}^{R}\frac{\hat{\theta}_r-\theta}{\widehat{SD}(\hat{\theta})}$ |
| `RMSE(...)` | root mean-square error (RMSE) | `RMSE` (default) | $\sqrt{\frac{1}{R}\sum_{r=1}^{R}(\hat{\theta}_r - \theta)^2}$ |
| | normalized RMSE | `NRMSE` | $\frac{\text{RMSE}}{max(\hat{\theta})-min(\hat{\theta})}$ |
| | standardized RMSE | `SRMSE` | $\frac{\text{RMSE}}{\widehat{SD}(\hat{\theta})}$ |
| | coefficient of variation (CV) | `CV` | $\frac{\text{RMSE}}{\frac{1}{R}\sum_{r=1}^{R}\hat{\theta}_r}$ |
| | root mean-square log-error | `RMSLE` | $\sqrt{\frac{1}{R}\sum_{r=1}^{R}[\log(\hat{\theta}_r+1)-\log(\theta+1)]^2}$ |
| `MAE(...)` | mean absolute error (MAE) | `MAE` (default) | $\frac{1}{R}\sum_{r=1}^{R}\lvert\hat{\theta}_r - \theta\rvert$ |
| | normalized MAE | `NMAE` | $\frac{\text{MAE}}{max(\hat{\theta})-min(\hat{\theta})}$ |
| | standardized MAE | `SMAE` | $\frac{\text{MAE}}{\widehat{SD}(\hat{\theta})}$ |
| `IRMSE(...)` | integrated root mean-square error | — | $\int_{\psi}(f(\psi,\hat{\theta}) - f(\psi,\theta))^2 g(\psi)$ |
| `EDR(...)` | empirical detection rate | — | $\frac{1}{R}\sum_{r=1}^{R}\#(p_r \leq \alpha)$ |
| `ECR(...)` | empirical coverage rate | — | $1-\frac{1}{R}\sum_{r=1}^{R}\#[\widehat{CI}(\alpha/2)_r \geq \theta \cap \widehat{CI}(1-\alpha/2)_r \leq \theta]$ |
| `MSRSE(...)` | mean-square relative standard error | — | $\frac{\frac{1}{R}\sum_{r=1}^{R}\widehat{SE}(\hat{\theta}_r)}{\widehat{SD}(\hat{\theta})}$ |
| `RAB(...)` | relative absolute bias | — | $\frac{\text{RAB}_2}{\text{RAB}_1}$ |
| `RE(...)` | relative efficiency | — | $\left(\frac{\text{RMSE}_2}{\text{RMSE}_1}\right)^2$ |