Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Getting Rid of the Loops in Statistical Simulations: The R Package simTool

Marsel Scheer

Abstract

The **simTool** package is designed for statistical simulations that have two components: One component generates the data and the other one analyzes the data. The main aims of the **simTool** package are the reduction of the administrative source code (mainly loops and management code for the results) and a simple applicability of the package that allows the user to quickly learn how to work with the **simTool** package. Parallel and distributed computing is also supported. Finally, convenient functions are provided to summarize the simulation results.

Keywords: R, statistical simulations, parallel computing.

1. Introduction

In statistics there is a broad range of applications for simulation studies. Often they are conducted to assess the performance, robustness or the small sample behavior of a statistical method or simply compare different statistical methods. In the past, the author has implemented many small, sometimes quick and dirty, simulation studies and a few rather large (computationally intensive) studies. Basically, the task was to investigate or to compare different statistical methods under different distributional settings. After implementing the statistical methods and the functions for the data generation the most tedious and annoying parts are the construction of the loops and the organization of the result object. The concern of the **simTool** package is that annoying part. Actually, we only want to specify how the datasets have to be generated and analyzed. Calling the functions to generate and analyze the data and afterwards store the results is completely handled by the **simTool** package. To be more precise, suppose we have k functions, $g_1, \ldots, g_k$ that generate some data and ℓ functions, $f_1, \ldots, f_\ell$ that analyze the data. The core of **simTool** is the following "pseudo code":

```
init result object
for g in  {g_1, ..., g_k}
    data = g()
    for f in  {f_1, ..., f_ℓ}
        append f(data) to the result object
```

Thus, the general workflow is to define the sets $\{g_1, \ldots, g_k\}$ and $\{f_1, \ldots, f_\ell\}$, which is aided by the **simTool** package, and **simTool** takes care of the rest. Quite often the first reaction about this core functionality was doubting its usefulness, because it seems that one only saves two for-loops. But in general (without the **simTool** package), every parameter that is varied in a simulation study, for instance the variance in a normal model, introduces its own for-loop. Hence, changing the set of parameters usually requires some kind of adaption of the for-loops or the management of the result objects has to be revised. Also, often another debugging cycle is needed and probably the code that summarizes the results has to be adapted. The author's experience is that changing the set of parameters happens with probability one, even one year or later after the simulation study was conducted. Utilizing the **simTool** package no adapting, debugging or any kind of extra work is usually necessary after changing the set of parameters. One solely has to be sure that the set $\{g_1, \ldots, g_k\}$ and $\{f_1, \ldots, f_\ell\}$ are correct and work properly together. A nice side effect is that the two sets $\{g_1, \ldots, g_k\}$ and $\{f_1, \ldots, f_\ell\}$ are much easier to read and understand than a source code full of for-loops and temporary objects that disguise the content of the simulation study. Figure 1 illustrates how the **simTool** package could be embedded or utilized within a simulation study.

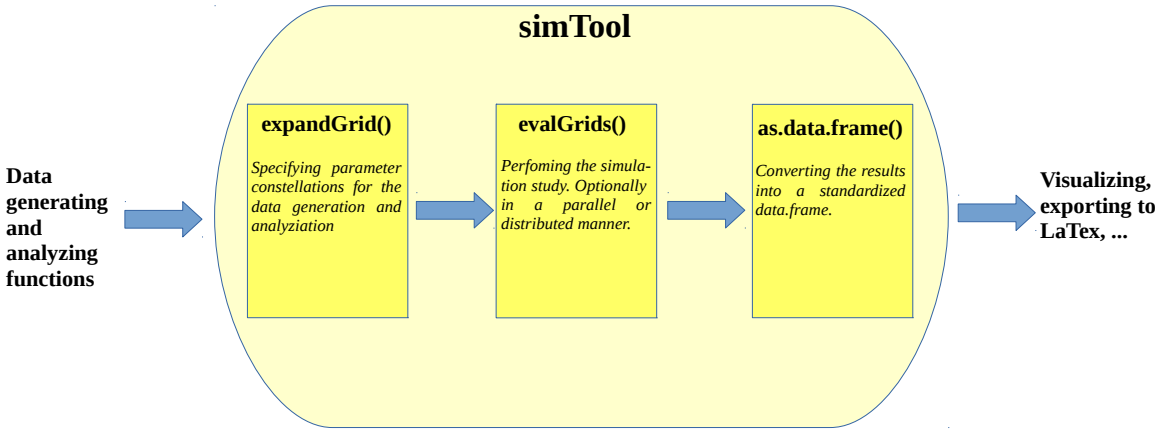


Figure 1: General workflow for the **simTool** package.

The outline of this publication is as follows. Section 2 compares the **simTool** package with packages hosted on CRAN that facilitate simulation studies. Very easy and self-explanatory examples that illustrate the workflow are presented in Section 3. In Section 4 the package is explained and discussed in detail. The practical usefulness of the **simTool** package is illustrated in Section 6 and 7 by reproducing simulation studies that were recently published in the Journal of Statistical Software. Finally, we state our concluding remarks in Section 8.

The **simTool** package is hosted on CRAN and the latest development version can be installed by calling `devtools::install_github("MarselScheer/simTool")`. Bugs, questions, feature requests and so on can be made through GitHub, where the package is also hosted or of course by simply contacting the author via the e-mail address stated in the affiliation.

2. Similar packages

A few packages can be found on CRAN that facilitate simulation studies. We briefly compare these packages with the **simTool** package.

In 2010, within the **mutoss** package the author already provided functions to facilitate simulation studies that have also the two components: data generation and data analyzation. But these functions are tailored to simulation studies in the context of multiple hypotheses testing, cf. Blanchard, Dickhaus, Hack, Konietschke, Rohmeyer, Rosenblatt, Scheer, and Werft (2014) and Blanchard, Dickhaus, Hack, Konietschke, Rohmeyer, Rosenblatt, Scheer, and Werft (2010) and does not support parallel or distributed computing. Whereas the **simTool** package may serve in much more general situations. However, the author believes that the **simTool** package is easier and more intuitive to handle. This enhancements are mainly a result of additional years of experience with R and simulation studies.

A further R-package is **harvestr**, cf. Redd (2014). Unfortunately, neither the examples nor the vignette of the **harvestr** package give us sufficient insight into the **harvestr** package. Hence, we no further comment this package.

Another R-package is **simFrame**. In Alfons, Templ, and Filzmoser (2010) the authors write themselves: *The package simFrame is intended to be as general as possible, but has initially been developed for close-to-reality simulation studies in survey statistics. Moreover, it is focused on simulations involving typical data problems such as outliers and missing values. Therefore, certain proportions of the data may be contaminated or set as missing in order to investigate the quality and behavior of, e.g., robust estimators or imputation methods.* In the author's opinion the **simFrame** package is rather hard to learn. Hence, it is not easy for the author to compare the **simFrame** package with the **simTool** package, but the **simFrame** package seems not to fit the needs of the author. For instance, it seems that it is only possible to specify one fixed data generating function and one fixed data analyzing function without any options to control these functions by varying parameters. For example, reproducing the simulation studies presented in the Sections 6 and 7 with the **simFrame** package seems not to be very intuitive and easy. Furthermore, our experience is that on the one hand it is sometimes handy to preserve the generated data, which is not supported by the **simFrame** package. On the other hand, if the simulation study is very memory consuming, it may be necessary to discard the generated data and summarize the results over the replications as soon as possible to spare memory. This seems also not to be supported by the **simFrame** package.

Finally, the **ezsim** package Chan (2014) is simple to learn and easy to handle. Similar to the **simTool** package, one can specify a data generating function and an estimator function. The parameter `parameter_def` controls the parameter for the data generating function. However, it is not as flexible as the **simTool** package. For instance, **ezsim** seems not to allow to vary parameters of the estimator function. It is only possible to specify one data generating function and control it by varying parameters. Hence, if one wants to use `rnorm` and `rexp` for

the data generation one has to write a wrapper function. But even with the wrapper function the handling may become cumbersome, because only a few argument names of `rnorm` and `rexp` coincide. Finally, preserving the generated data or discarding the generated data and summarizing the results over the replications as soon as possible to spare memory is not possible with the **ezsim** package.

3. A few simple examples

Before we start with the details, we give a few examples of increasing complexity. From a statistical point of view the examples are senseless. The point here is the self-explanatory character of the examples. For illustration purpose, the functions chosen for generating and analyzing the data are very elementary. More examples can be found in the vignette of this package, just call `vignette("simTool")` from the R-console. A reproduction of published simulation studies can be found in the Sections 6 and 7.

First, we define a set of data generating functions represented by a `data.frame`:

```
print(dg <- expandGrid(fun="rexp", n=10))
```

```
   fun  n
1 rexp 10
```

Next, we define a set of analyzing functions:

```
print(pg <- expandGrid(proc="min"))
```

```
  proc
1  min
```

Finally, we conduct the simulation study, i.e. generate 10 exponential distributed random variables by default with mean 1 and calculate the minimum of the sample. This is done 2 times.

```
set.seed(1234)
eg <- evalGrids(dg, pg, replications=2)
```

```
[1] "Estimated replications per hour:  83653708"
```

```
as.data.frame(eg)
```

```
  i j  fun  n proc replication      V1
1 1 1 rexp 10  min           1 0.06769
2 1 1 rexp 10  min           2 0.11797
```

Now, we extend the set of analyzing functions and repeat the simulation study.

```
pg <- expandGrid(proc=c("min", "max"))
set.seed(1234)
eg <- evalGrids(dg, pg, replications=2)

[1] "Estimated replications per hour:  54609383"

print(df <- as.data.frame(eg))

  i j  fun  n proc replication      V1
1 1 1 rexp 10  min           1 0.06769
2 1 1 rexp 10  min           2 0.11797
3 1 2 rexp 10  max           1 1.34852
4 1 2 rexp 10  max           2 2.40319
```

Note that the `name` attribute of the objects returned by the functions `min` and `max` is `NULL`. Hence, the resulting `data.frame` contains the column `V1` otherwise the `name` attribute of the returned object will be used. We now gradually increase the complexity briefly commenting the code.

Additionally, we define the function `minmax` for analyzing the data, which returns the minimum and maximum as a vector with a `name` attribute.

```
minmax = function(x) c(min=min(x), max=max(x))
pg <- expandGrid(proc=c("min", "max", "minmax"))
set.seed(1234)
eg <- evalGrids(dg, pg, replications=2)

[1] "Estimated replications per hour:  60157348"

print(df <- as.data.frame(eg))

  i j  fun  n   proc replication      V1     min   max
1 1 1 rexp 10    min           1 0.06769      NA    NA
2 1 1 rexp 10    min           2 0.11797      NA    NA
3 1 2 rexp 10    max           1 1.34852      NA    NA
4 1 2 rexp 10    max           2 2.40319      NA    NA
5 1 3 rexp 10 minmax           1      NA 0.06769 1.349
6 1 3 rexp 10 minmax           2      NA 0.11797 2.403
```

Now, we add a further data generating function

```
dg <- expandGrid(fun=c("rexp", "rnorm"), n=10)
set.seed(1234)
eg <- evalGrids(dg, pg, replications=2)

[1] "Estimated replications per hour:  32647555"
```

```
print(df <- as.data.frame(eg))
```

```
   i j   fun  n   proc replication       V1      min   max
1  1 1  rexp 10    min           1  0.06769       NA    NA
2  1 1  rexp 10    min           2  0.11797       NA    NA
3  1 2  rexp 10    max           1  1.34852       NA    NA
4  1 2  rexp 10    max           2  2.40319       NA    NA
5  1 3  rexp 10 minmax           1       NA  0.06769 1.349
6  1 3  rexp 10 minmax           2       NA  0.11797 2.403
7  2 1 rnorm 10    min           1 -2.09496       NA    NA
8  2 1 rnorm 10    min           2 -1.88456       NA    NA
9  2 2 rnorm 10    max           1  1.29298       NA    NA
10 2 2 rnorm 10    max           2  1.15447       NA    NA
11 2 3 rnorm 10 minmax           1       NA -2.09496 1.293
12 2 3 rnorm 10 minmax           2       NA -1.88456 1.154
```

Calculating the mean over the 2 replications can be simply done by

```
set.seed(1234)
eg <- evalGrids(dg, pg, replications=2, summary.fun=mean)
```

```
[1] "Estimated replications per hour:  67490"
```

```
print(df <- as.data.frame(eg))
```

```
  i j   fun  n   proc value       V1      min   max
1 1 1  rexp 10    min (all)  0.09283       NA    NA
2 1 2  rexp 10    max (all)  1.87586       NA    NA
3 1 3  rexp 10 minmax (all)       NA  0.09283 1.876
4 2 1 rnorm 10    min (all) -1.98976       NA    NA
5 2 2 rnorm 10    max (all)  1.22373       NA    NA
6 2 3 rnorm 10 minmax (all)       NA -1.98976 1.224
```

One can easily conduct the simulation study in a parallel manner.

```
# setting the seed by set.seed has no effect on the simulation
set.seed(1234)
# increasing the replications, because 2 replications can not be
# distributed on 4 CPUs.
eg <- evalGrids(dg, pg, replications=20, summary.fun=mean, ncpus=4)
```

```
[1] "Estimated replications per hour:  232449"
```

```
print(df <- as.data.frame(eg))
```

```
  i j   fun  n   proc value      V1     min   max
1 1 1  rexp 10    min (all)  0.1219      NA    NA
2 1 2  rexp 10    max (all)  2.7546      NA    NA
3 1 3  rexp 10 minmax (all)      NA  0.1219 2.755
4 2 1 rnorm 10    min (all) -1.2280      NA    NA
5 2 2 rnorm 10    max (all)  1.5582      NA    NA
6 2 3 rnorm 10 minmax (all)      NA -1.2280 1.558
```

Since the simulation is conducted in a parallel manner, the seed must be specified by the parameter `clusterSeed` of `evalGrids`, which by default equals `rep(12345, 6)`. For more details see the following section.

4. Package description

The package consists only of 3 functions:

- `expandGrid`: Convenient function to define the sets $\{g_1, \ldots, g_k\}$ and $\{f_1, \ldots, f_\ell\}$
- `evalGrids`: The workhorse that conducts the simulation study
- `as.data.frame.evalGrid`: Convenient function to summarize the results as a `data.frame`

Usually they are also called in that order. We will now discuss all 3 functions. Again as in the last section we use `dg` for the `data.frame` representing the data generating functions $\{g_1, \ldots, g_k\}$ and `pg` for the `data.frame` representing the functions $\{f_1, \ldots, f_\ell\}$ for analyzing the data.

4.1. Defining the sets for data generation and data evaluation

The function `evalGrids` that conducts the simulation expects two `data.frames`, say `dg` and `pg`. `evalGrids` will interpret these `data.frames` row-wise. The first column in both `data.frames` must contain the character names of the functions to be called and the other columns are the parameters that are passed to the function specified in the first column. If one of the other parameters is `NA`, then this parameter is not passed to the specified function. The following `data.frame` may be used as a representation of the set of data generating functions $\{g_1, \ldots, g_k\}$:

```
library("plyr")
print(dg <- rbind.fill(
  expandGrid(fun=c("rnorm"), n=c(10,20), mean=1:2),
  expandGrid(fun="rexp", n=10, rate=1:2)))
```

```
    fun  n mean rate
1 rnorm 10    1   NA
2 rnorm 20    1   NA
3 rnorm 10    2   NA
4 rnorm 20    2   NA
5  rexp 10   NA    1
6  rexp 10   NA    2
```

From a technical point of view, this `data.frame` will be automatically converted to 4 R-functions that generate normally distributed random variables of sample size 10 or 20 with mean 1 or 2 and 2 R-functions that generate exponentially distributed random variables of sample size 10 with mean 1 or 1/2. The function `expandGrid` is a very simple wrapper of the function `expand.grid` from the **base** package and is only a convenient function that of course may be replaced by the users favorite choice.

As already mentioned, `pg` the `data.frame` for the functions that analyze the generated data must follow the same rules as `dg`. The first column specifies the function to be called, the other columns determine parameters that are passed to the function, and `NA` is ignored. For instance, this `data.frame`

```
print(pg <- expandGrid(proc="mean", trim=c(0, 0.1)))
```

```
  proc trim
1 mean  0.0
2 mean  0.1
```

will be automatically converted to 2 R-functions. One function is the regular arithmetic mean and the other a trimmed mean with `trim` parameter set to 0.1. At this point we should clarify which parameter will be used for the generated data. Different functions may have different parameter names for the dataset, e.g. `x` is the argument for the data for the functions `mean`, `median`, `ecdf`, etc. and `data` is the argument for the function `lm`, `glm`, and so on. The **simTool** package passes the generated data to the first argument that is not specified by default and not specified by a column from `pg`. This may seem odd at the first glance, but after one year of development and practical working with the **simTool** package this seems to be a good choice in most cases. Hence, if this automatism fails, at least at the moment, one has to write a wrapper function to correct this. In such a case, the author would reserve the first argument of the wrapper function for the generated data. This was not necessary in any of the simulation studies the author has made with the **simTool** package. If the experience will show that such a wrapper function is often necessary a solution will be developed to get rid of such annoying necessity.

4.2. Conducting a simulation study

The workhorse `evalGrids` has the following simplified pseudo code:

```
1  convert dg to R-functions  {g_1, ..., g_k}
2  convert pg to R-functions  {f_1, ..., f_ℓ}
3  init result object
4  append dg and pg to the result object
5  t1 = current.time()
6  for g in  {g_1, ..., g_k}
7      for r in 1:replications (optionally in a parallel/distributed manner)
8          data = g()
9          for f in  {f_1, ..., f_ℓ}
10             append f(data) to the result object
11         optionally append data to the result object
```

```
12      optionally summarize the results over all replications, but
        separately for f_1, ..., f_ℓ
13     optionally save the result object to HDD
14 t2 = current.time()
15 Estimate the number of replications per hour from t1 and t2
```

In order to discuss the result object we define very simple `data.frames` `dg` and `pg`.

```
dg <- expandGrid(fun="rexp", n=c(5, 10))
pg <- expandGrid(proc="mean", trim=c(0.1,0.2))
set.seed(1234)
eg <- evalGrids(dg, pg, replications=10)
```

```
[1] "Estimated replications per hour:  16202912"
```

`evalGrids` returns a `list` of class `evalGrid`. For documentation purpose of the simulation study the elements `call`, `dataGrid`, `procGrid`, `summary.fun`, `est.reps.per.hour`, `sessionInfo` contain the function call, `dg`, `pg`, the functions that were used in command 12 (in the pseudo code at the beginning of this section) to summarize the results over all replications, the estimated number of replications that will be computable in one hour, and the `list` returned by `utils::sessionInfo`, respectively. The most interesting element is `simulation`, which itself is a list. `simulation[[i]][[r]]$data` contains the data generated by the parameter constellation of the ith row in `dg` in the rth replication and `simulation[[i]][[r]]$results[[j]]` contains the object returned by the function and parameter constellation of the jth row in `pg` applied to the element `simulation[[i]][[r]]$data`. For example,

```
eg[["simulation"]][[2]][[5]]$data
```

```
 [1] 2.14070 0.74931 0.34091 0.41084 0.62809 0.12774 0.85323 0.08172
 [9] 0.83608 0.73992
```

contains the dataset that was generated by

```
eg$dataGrid[2,]
```

```
   fun  n
2 rexp 10
```

in the 5th replication and

```
eg[["simulation"]][[2]][[5]]$results[[1]]
```

```
[1] 0.5858
```

is the object returned by

```
eg$procGrid[1,]
```

```
  proc trim
1 mean  0.1
```

Hence, we can simply reproduce this result by

```
with(eg[["simulation"]][[2]][[5]], mean(data, trim=0.1))
```

```
[1] 0.5858
```

Now, we discuss all arguments one by one, except `envir` because this parameter is only interesting in a few special cases. For this parameter we refer to the vignette.

```
args(evalGrids)
```

```
function (dataGrid, procGrid = expandGrid(proc = "length"), replications = 1,
    discardGeneratedData = FALSE, progress = FALSE, summary.fun = NULL,
    ncpus = 1L, cluster = NULL, clusterSeed = rep(12345, 6),
    clusterLibraries = NULL, clusterGlobalObjects = NULL, fallback = NULL,
    envir = globalenv())
NULL
```

The first three arguments should be clear by now. By default `evalGrids` saves ANY dataset generated by $\{g_1, \ldots, g_k\}$ and ALL result objects returned by the functions $\{f_1, \ldots, f_\ell\}$. This can be very memory consuming. For instance, if `replications=100`, $k = 6$, and $\ell = 3$, then `evalGrids` will save all $100 \cdot 6$ generated datasets and all $100 \cdot 6 \cdot 3$ result objects. Setting `discardGeneratedData=TRUE` a generated dataset is discarded right after every function contained in $\{f_1, \ldots, f_\ell\}$ has been applied to that dataset, confer command 11 in the pseudo code. Further, memory can be saved by summarizing the result objects through the parameter `summary.fun`. Passing a vector of univariate functions, e.g. `mean`, `sd`, `median`, etc., to `summary.fun` the objects returned by $\{f_1, \ldots, f_\ell\}$ are summarized (over the replications and for each combination of g_i $(i = 1, \ldots, k)$ and f_j $(j = 1, \ldots, \ell)$ separately) by the functions specified in `summary.fun` as soon as possible, confer command 12 in the pseudo code. This also automatically discards the generated datasets and all result objects created in command 10. A progress text bar in the console can be activated through `progress=TRUE`. It is updated, even under parallel computations, as the for-loop in command 6 chooses the next element. A cluster created with the **parallel** package can be passed to the parameter `cluster`. It will then be automatically used to distribute the replications over the cluster, confer command 6. In this case the random number generator proposed in L'Ecuyer (1999) is used. Reproducible results can be obtained by specifying `clusterSeed`. The seed must be a vector of 6 (signed) integer values. For further details confer the documentation of the parameter `clusterSeed`. By specifying an integer for `ncpus` a cluster on the local machine is created for the user and passed to the argument `cluster` of `evalGrids`. The parameter `clusterLibraries` and `clusterGlobalObjects` can be used to load libraries on the cluster and to transfer `R`-objects to the cluster that are necessary for the simulation. For instance, if the simulation study uses

the **boot** package and an object O from the global environment, then the cluster has to load the **boot** package and the object O must be transferred to the global environment of the cluster. Finally, the parameter `fallback` is for all users who are afraid of loosing results by server crashes, power black outs, and so on. Passing a character to `fallback` will cause `evalGrids` to save the results every time the for-loop in command 6 chooses the next element. Loading this file with the `load` function creates an R-object of class `evalGrid` called `fallBackObj`. A nice side effect is that one can load this object before the simulation study is finished and examine the results so far produced.

4.3. Converting results to a readable table

If all result objects (returned by $f_1, \ldots, f_l$) can be automatically transformed into a `data.frame`, then simple calling `as.data.frame` on an R-object of class `evalGrid` returns a `data.frame`.

```
head(df<-as.data.frame(eg))
```

```
  i j  fun n proc trim replication     V1
1 1 1 rexp 5 mean  0.1           1 0.6484
2 1 1 rexp 5 mean  0.1           2 0.4462
3 1 1 rexp 5 mean  0.1           3 1.2922
4 1 1 rexp 5 mean  0.1           4 0.7101
5 1 1 rexp 5 mean  0.1           5 1.5454
6 1 1 rexp 5 mean  0.1           6 0.4850
```

```
tail(df)
```

```
   i j  fun  n proc trim replication     V1
35 2 2 rexp 10 mean  0.2           5 0.6175
36 2 2 rexp 10 mean  0.2           6 1.3181
37 2 2 rexp 10 mean  0.2           7 0.6129
38 2 2 rexp 10 mean  0.2           8 1.5209
39 2 2 rexp 10 mean  0.2           9 0.8398
40 2 2 rexp 10 mean  0.2          10 0.5799
```

The first two columns indicate which row of `dg` and `pg` were the basis for obtaining the results displayed in the last columns. From there on the column names of `df` consist of the column names of `dg` followed by the column names of `pg` and the last column names are the `name` attribute of the result objects. If a result object does not have a name attribute, the results are displayed under `V1, V2`, and so one, as in our example.

As one can see, the results of every single replication are contained in the `df`. The column `replication` states in which replication the result was produced. Together with the column `i` it is very easy to extract the corresponding dataset that leads to the result. For instance,

```
eg[["simulation"]][[2]][[10]][["data"]]
```

```
 [1] 2.22616 0.83566 0.83674 0.55118 0.55822 0.29806 1.58251 0.09399
 [9] 0.04293 0.39925
```

leads to the last line in `df` by calculating the trimmed mean with `trim=0.2`.

The parameter `summary.fun` works just the same way it works within `evalGrids`. It summarizes the results over the replications but separately for all combinations of data generating and data analyzing functions.

```
as.data.frame(eg, summary.fun=c(mean, sd))
```

```
  i j  fun  n proc trim value V1_mean  V1_sd
1 1 1 rexp  5 mean  0.1 (all)  1.0227 0.4895
2 1 2 rexp  5 mean  0.2 (all)  0.9129 0.4575
3 2 1 rexp 10 mean  0.1 (all)  0.9429 0.3549
4 2 2 rexp 10 mean  0.2 (all)  0.8712 0.3526
```

Of course, if the results were already summarized by `evalGrid` a simple call of `as.data.frame` is enough to display the summarized results. Sometimes, the object returned by the data analyzing functions can not be automatically coerced into a `data.frame`. For this purpose it is possible to preprocess the result objects contained in the `evalGrid`-object by a function specified by the parameter `convert.result.fun` in order to convert the object to a `data.frame` and optionally summarize these further with the functions specified by `summary.fun`. How this works can be seen in Section 7 or by simply executing:

```
example(as.data.frame.evalGrid)
```

5. Reproducing published simulations

The applicability of the **simTool** package is illustrated by reproducing two simulation studies that were recently published in the Journal of Statistical Software. Searching the publication for newest to oldest we found Ritter, Jewell, and Hubbard (2014) and Jamshidian, Jalal, and Jansen (2014). We choose these two publications solely for one reason. The simulation studies presented there were easily reproducible by simply installing the corresponding packages from CRAN and running the source code from the supplementary.

Discussing these packages is beyond of the scope of this publication. Instead, we present and discuss the original source code and then show how the simulation studies can be conducted with the **simTool** package.

6. MissMech package

6.1. Original source code

The following source code (lines 131 till 167 from v56i06.R) is the basis for Table 1 in Jamshidian *et al.* (2014) and can be found in the corresponding supplementary.

```
#----- R code for Table 1 simulation results
# To reach the results in Table 1, uncomment the appropriate line and set
```

```
# the distribution parameters
library("MissMech")
set.seed(1010)
n <- 300
p <- 5
pctmiss <- 0.2
pval_Hw <- c()
pval_Non <- c()
pval_HwComp <- c()
pval_NonComp <- c()
df.t <- 3
shape.g <- 2
rep <- 1000/100
for (k in 1:rep){
     y <- matrix(rnorm(n * p), nrow = n)
    #y <- matrix(rt(n * p, df.t), nrow = n)
    #y <- matrix(rgamma(n * p, shape.g, 1) , nrow = n)
    #y <- matrix(runif(n * p) , nrow = n)

     ycomp <- y
     missing <- matrix(runif(n * p), nrow = n) < pctmiss
     y[missing] <- NA
     out <- TestMCARNormality(data = y, del.lesscases = 6, imputation.number = 1,
            method = "Auto", imputation.method = "Dist.Free", nrep = 10000,
            n.min = 30, seed = NA, alpha = 0.05, imputed.data = NA)
     ycomp <- ycomp[sort(out$caseorder), ]
     out.comp <- TestMCARNormality(data = out$analyzed.data, del.lesscases = 6,
            imputation.number = 1, method = "Auto", imputation.method = "Dist.Free",
            nrep = 10000, n.min = 30, seed = NA, alpha = 0.05, imputed.data = ycomp)
     pval_Hw <- c(out$pvalcomb, pval_Hw)
     pval_Non <- c(out$pnormality, pval_Non)
     pval_HwComp <- c(out.comp$pvalcomb, pval_HwComp)
     pval_NonComp <- c(out.comp$pnormality, pval_NonComp)
}
c(sum(pval_Hw < 0.05) / k, sum(pval_Non < 0.05) / k,
  sum(pval_HwComp < 0.05) / k, sum(pval_NonComp < 0.05) / k)
```

It seems that the source code was developed for one particular scenario with one distribution function and one parameter constellation. Afterwards, further parameter constellations and distribution functions were introduced without revising the source code. The author has done simulation studies in such a manner not only once and such an approach has some drawbacks:

1. Reproducing the results is cumbersome, especially for many different parameter constellations.
2. Temporary objects or variables that are only important under specific circumstances, e.g. `shape.g`, distract the reader from the important source code.

3. Extending and adapting the simulation study is cumbersome and error-prone, especially if this had to be done by a third person.
4. Transferring the results into the publication is error-prone.

Among other things, these reasons cause the author to develop the **simTool** package.

6.2. Reproduction by the simTool package

First define the functions that generate the data.

```
library("MissMech")
createMatrices = function(vec, n, p, pctmiss){
  completeMatrix = matrix(vec, nrow=n)
  incompleteMatrix = completeMatrix
  incompleteMatrix[matrix(runif(n * p), nrow = n) < pctmiss] = NA
  list(completeMatrix=completeMatrix, incompleteMatrix=incompleteMatrix)
}
matrix.rnorm = function(n, p, pctmiss) {
  createMatrices(rnorm(n*p), n, p, pctmiss)
}
matrix.rt = function(n, p, df, pctmiss){
  createMatrices(rt(n*p, df=df), n, p, pctmiss)
}
matrix.rgamma = function(n, p, shape, pctmiss){
  createMatrices(rgamma(n*p, shape=shape), n, p, pctmiss)
}
matrix.runif = function(n, p, pctmiss){
  createMatrices(runif(n*p), n, p, pctmiss)
}
```

Furthermore, we need the functions that analyze the generated data. The requirements of this particular simulation does not fit perfectly into the approach of the **simTool** package. The function `MissMech::TestMCARNormality` has a parameter `seed` and in order to reproduce Table 1 we need to set `seed=NA`. But with the **simTool** package it is not possible to pass `NA` to the parameter `seed`. Hence, a wrapper function is needed that sets `seed=NA` if `seed` is missing.

```
calcPValues = function(
  data, del.lesscases = 6, imputation.number = 1,
  method = "Auto", imputation.method = "Dist.Free", nrep = 10000,
  n.min = 30, seed, alpha = 0.05){

  if (missing(seed))
    seed = NA

  out <- TestMCARNormality(
    data = data$incompleteMatrix, del.lesscases = del.lesscases,
```

```
    imputation.number = imputation.number, method = method,
    imputation.method = imputation.method, nrep = nrep,
    n.min = n.min, seed = seed, alpha = alpha, imputed.data = NA)

  out.comp <- TestMCARNormality(
    data = out$analyzed.data,
    del.lesscases = del.lesscases, imputation.number = imputation.number,
    method = method,  imputation.method = imputation.method,
    nrep = nrep,  n.min = n.min, seed = seed, alpha = alpha,
    imputed.data = data$completeMatrix[sort(out$caseorder),])

   c(pval_Hw = out$pvalcomb, pval_Non = out$pnormality,
     pval_HwComp = out.comp$pvalcomb, pval_NonComp = out.comp$pnormality)
}
```

Note that this source code is more intelligible than the original source code merely by concentrating on the data generation and analyzation and the objects necessary for the particular situation. Now, as we have data generating and analyzing functions at hand we can reproduce Table 1 from Jamshidian *et al.* (2014). But first, we exactly reproduce the 7th row of Table 1 in Jamshidian *et al.* (2014).

```
dg = expandGrid(fun="matrix.runif", n=300, p=5, pctmiss=0.2)
pg <- expandGrid(proc="calcPValues", del.lesscases = 6,
  imputation.number = 1, method = "Auto",
  imputation.method = "Dist.Free", nrep = 10000,
  n.min = 30, alpha = 0.05)
set.seed(1010)
eg = evalGrids(dg, pg, replications=1000)

[1] "Estimated replications per hour:  1563"

as.data.frame(eg, summary.fun=function(x) round(100*mean(x<0.05),1))[,c("fun",
  "pval_Hw", "pval_Non", "pval_HwComp", "pval_NonComp")]

           fun pval_Hw pval_Non pval_HwComp pval_NonComp
1 matrix.runif      95      9.8        96.4          8.9
```

By the nature of the original source code in Section 6.1, exact reproduction of all results is clumsy, because one has to specify only one data generating function at once as shown above. However, since we will use 4 CPUs we can not make use of the `seed` chosen in Jamshidian *et al.* (2014). Thus, our results are only qualitative the same. Furthermore, the calculations require the library `MissMech` and the data generating function `createMatrices`, `matrix.rnorm`, etc. we have defined in the global environment. As mentioned in Section 4 by specifying `ncpus` larger than one a cluster is created for the user. In order to be able to conduct the simulation in a parallel manner these functions must be transferred to the cluster and the library `MissMech` must be loaded on the cluster. The parameters `clusterLibraries` and `clusterGlobalObjects` serve exactly this purpose.

```
dg = rbind.fill(
  expandGrid(fun="matrix.rnorm", n=300, p=5),
  expandGrid(fun="matrix.rt", n=300, p=5, df=c(3,5,7,9,20)),
  expandGrid(fun="matrix.runif", n=300, p=5),
  expandGrid(fun="matrix.rgamma", n=300, p=5, shape=c(2,5,10))
  )
print(dg <- cbind(dg, pctmiss=0.2))
```

```
             fun   n p df shape pctmiss
1   matrix.rnorm 300 5 NA    NA     0.2
2      matrix.rt 300 5  3    NA     0.2
3      matrix.rt 300 5  5    NA     0.2
4      matrix.rt 300 5  7    NA     0.2
5      matrix.rt 300 5  9    NA     0.2
6      matrix.rt 300 5 20    NA     0.2
7   matrix.runif 300 5 NA    NA     0.2
8  matrix.rgamma 300 5 NA     2     0.2
9  matrix.rgamma 300 5 NA     5     0.2
10 matrix.rgamma 300 5 NA    10     0.2
```

```
# pg is already defined correctly
eg = evalGrids(
  dg, pg, replications=1000,
  discardGeneratedData=TRUE, ncpus=4,
  clusterLibraries="MissMech",
  clusterGlobalObjects=c("createMatrices", unique(dg[, "fun"])))
```

```
[1] "Estimated replications per hour:  463"
```

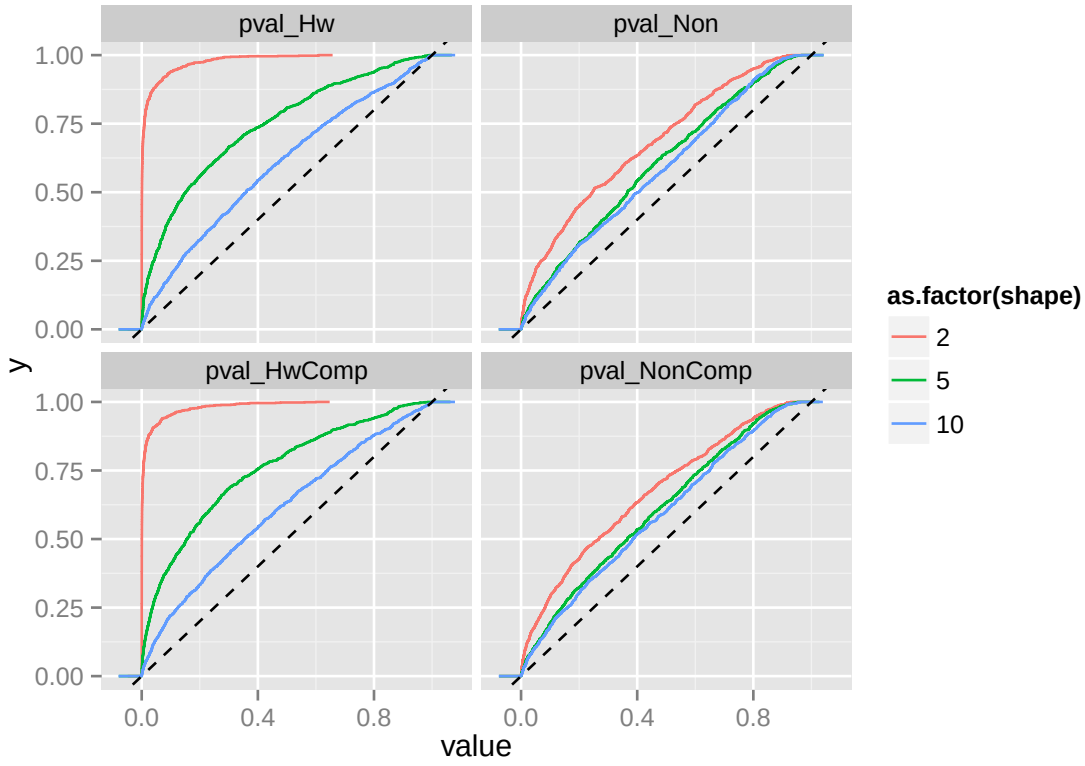
Obtaining Table 1 from Jamshidian *et al.* (2014) from our `evalGrid`-object is very simply.

```
table1 = as.data.frame(eg, summary.fun=function(x) round(100*mean(x<0.05), 1))
table1[,c("fun", "df", "shape", "pval_Hw", "pval_Non", "pval_HwComp",
          "pval_NonComp")]
```

```
             fun df shape pval_Hw pval_Non pval_HwComp pval_NonComp
1   matrix.rnorm NA    NA     4.5      6.9         5.4          6.6
2      matrix.rt  3    NA   100.0     12.4       100.0         11.8
3      matrix.rt  5    NA    88.6      8.4        91.2          8.1
4      matrix.rt  7    NA    55.1      8.1        57.4          7.6
5      matrix.rt  9    NA    35.6      7.8        35.4          6.7
6      matrix.rt 20    NA    13.0      6.7        10.5          6.5
7   matrix.runif NA    NA    95.2     13.0        95.8          9.2
8  matrix.rgamma NA     2    88.9     21.3        91.3         18.7
9  matrix.rgamma NA     5    27.7     12.0        29.6         11.1
10 matrix.rgamma NA    10    11.9     10.6        13.6         10.5
```

Although this is not part of Jamshidian *et al.* (2014) we illustrate that other representation, for instance as plots of the empirical cumulative distribution functions of the p-values, are also easy to create.

```
df = as.data.frame(eg)
library("reshape")
mdf = melt(df, measure.vars = c("pval_Hw", "pval_Non", "pval_HwComp",
    "pval_NonComp"))
mdf = subset(mdf, mdf[, "fun"]=="matrix.rgamma")
library("ggplot2")
ggplot(mdf, aes(x=value, colour=as.factor(shape))) +
  stat_ecdf() +
  geom_abline(slope=1, linetype="dashed") +
  facet_wrap(~variable)
```



7. multiPIM package

7.1. Original source code

We reproduce now Figure 1 from Ritter *et al.* (2014). The original source code for that Figure 1 can be found in v57i08.R in lines 39 till 174. Lines 39 till 119 code the actual simulation study and are partly displayed in the following code chunk:

```
for(a in length(ns):1) { ## do the big ns first
  cat("\n#########################################\nStarting on n =",
      ns[a], "\n#########################################\n\n")
  W <- gen.W(ns[a], num.covs, sigma)
  A <- gen.A(W)
  Y <- gen.Y(A, W, error.sd)
  for(b in 1:length(estimators)) {
    multiPIM.objects[[a]][[b]] <-
      multiPIM(Y, A, W, estimator = estimators[b],
               g.method = "main.terms.logistic", ## will be ignored for g-comp
               Q.method = "main.terms.linear",
               return.final.models = FALSE)
    param.estimates[a, b] <- multiPIM.objects[[a]][[b]]$param.estimates[1]
  }
}
```

The original source code before these two for-loops is concerned only with the initialization of the result object `multiPIM.objects` and definition of `ns`, `gen.W`, and so on. Again, extending the simulation study, for instance by varying `error.sd` or conducting more than one replication is error-prone and would also require to revise the original source code that creates Figure 1.

7.2. Reproduction by the simTool package

Again we start with the function that generates the data. The following three functions are copied from the supplementary of Ritter *et al.* (2014)

```
## function to generate W as multivariate normal
gen.W <- function(n, Sigma) {
  p = nrow(Sigma)
  W <- data.frame(mvrnorm(n = n, mu = rep(0, p), Sigma = Sigma))
  names(W) <- paste("w", 1:p, sep = "")
  return(W)
}

## function to generate A based on covs (W)
gen.A <- function(W) {
  A.probs <- plogis(0.2*rowSums(as.matrix(W)))
  A <- data.frame(a1 = rbinom(nrow(W), 1, A.probs))
}

## function to generate Y based on A and W
gen.Y <- function(A, W, error.sd) {
  Y <- data.frame(y1 = W[,1]*W[,2] + W[,3]*W[,4]
                  + rowSums(as.matrix(W)^2) * A[[1]]
                  + rnorm(nrow(W), 0, error.sd))
}
```

In order to utilize the **simTool** package two wrapper functions are needed. The function `multiPIM::multiPIM` that calculates the estimates expects 3 datasets `W`, `A`, and `Y`. Since `evalGrid` always passes only one dataset, we need a wrapper function that returns `W`, `A`, and `Y` at once and a wrapper function that passes the 3 datasets to `multiPIM::multiPIM`.

```
YAW = function(n, num.covs, covar, error.sd){
  sigma <- matrix(covar, num.covs, num.covs)
  diag(sigma) <- 1
  W = gen.W(n, sigma)
  A = gen.A(W)
  list(W = W,
       A = A,
       Y = gen.Y(A, W, error.sd))
}

est.fun = function(data, estimator, g.method, Q.method){
  multiPIM(data[["Y"]], data[["A"]], data[["W"]], estimator = estimator,
           g.method = g.method,
           Q.method = Q.method,
           return.final.models = FALSE)
}
```

We now reproduce all results necessary for Figure 1 from Ritter *et al.* (2014).

```
library("multiPIM")
library("MASS")

set.seed(23)
dg = expandGrid(
  fun="YAW",
  n=round(100*(2500^(1/99))^(99:0)),
  num.covs = 4, covar=0.2, error.sd=2)
pg = expandGrid(
  proc="est.fun", estimator=c("TMLE", "G-COMP"),
  g.method="main.terms.logistic",
  Q.method="main.terms.linear")
eg = evalGrids(dg, pg, discardGeneratedData = TRUE, replications = 1)
```

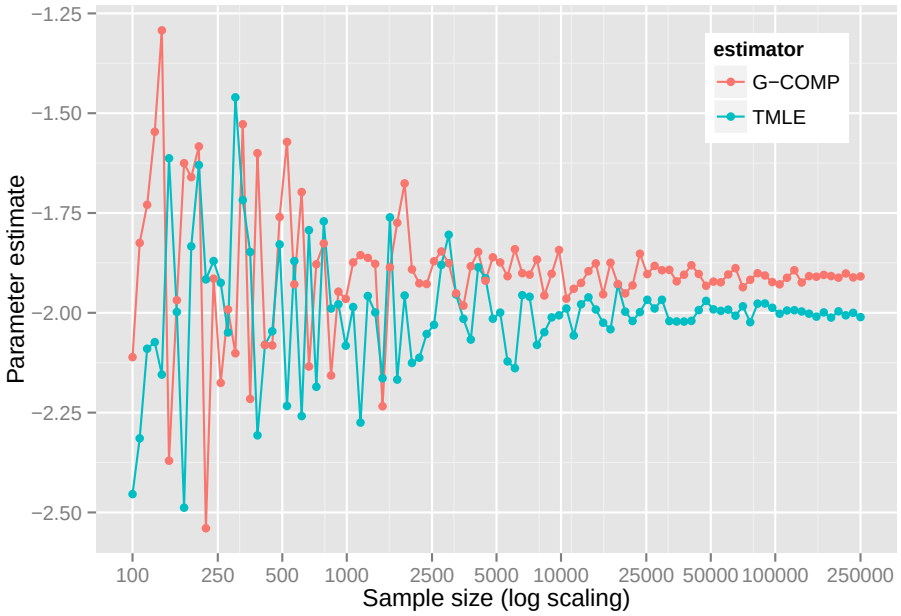
```
[1] "Estimated replications per hour:  50"
```

The object returned by `multiPIM` can not be automatically coerced into a `data.frame`. Hence, an appropriate function is passed to the argument `convert.result.fun` of `as.data.frame.evalGrid`.

```
df = as.data.frame(eg, convert.result.fun = function(result)
  c(estimate=result[["param.estimates"]][1]))
```

We use **ggplot2** for the visualization, but obtain a qualitative reproduction of Figure 1 in Ritter *et al.* (2014)

```
ggplot(df, aes(y=estimate, x=n, colour=estimator)) + geom_line() +
  geom_point() + ylim(min(df$estimate), max(df$estimate)) +
  theme(legend.position=c(0.85,0.85)) +
  ylab("Parameter estimate") + xlab("Sample size (log scaling)") +
  scale_x_log10(breaks=c(100*10^(0:3), 250*10^(0:3), 500*10^(0:2)))
```



From the call of `evalGrids` it is obvious that the replication equals only 1. It is very natural to increase the number of replications and to plot the mean of the estimates. Extending the original source code in this way requires some work and probably one or more debugging cycles. Utilizing the **simTool** package this is quite easy as we now show. We simple set `replications=400`. Furthermore, since we do not have any memory issues with this simulation we keep all individual results and summarize the data by setting `summary.fun=mean` within `as.data.frame.evalGrid` instead of `evalGrids`. Again, we use parallel computations. Hence, the libraries **multiPIM** and **MASS** must be loaded on the cluster and the functions `YAW`, `gen.A`, `gen.W`, and `gen.Y` must be transferred to the cluster.

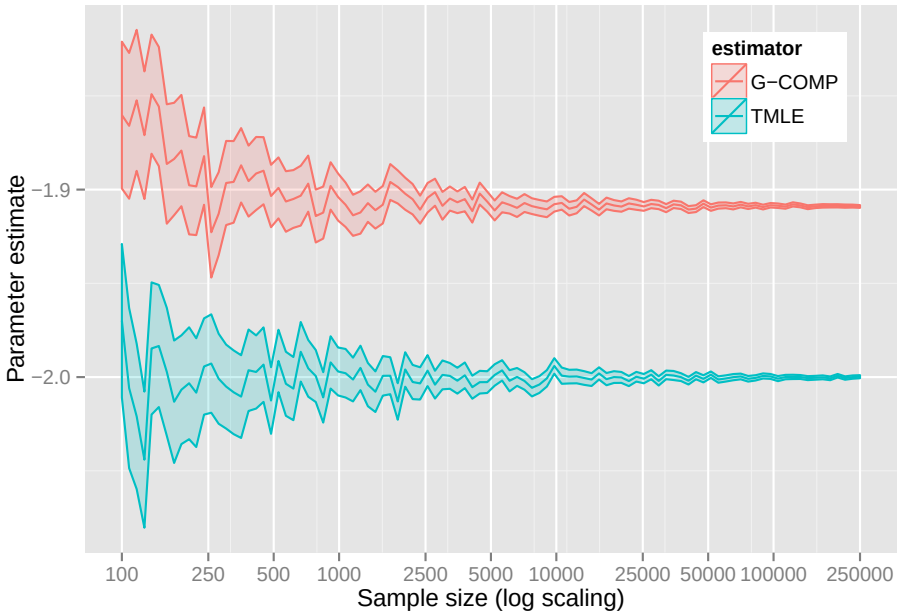
```
eg = evalGrids(dg, pg, discardGeneratedData = TRUE, replications = 400, ncpus=4,
  clusterLibraries=c("multiPIM", "MASS"),
  clusterGlobalObjects=c("YAW", "gen.A", "gen.W", "gen.Y"))
```

```
[1] "Estimated replications per hour:  151"
```

Calculating both the mean and the standard deviation is very easy and enables us to easily create a plot that is more informative than the original one. Note, we do not want to discuss the statistical issue of constructing confidence intervals for the estimator. This extended plot is solely an illustration of how easy and flexible the **simTool** package is.

```
df = as.data.frame(eg, convert.result.fun = function(result)
  c(estimate=result[["param.estimates"]][1]),
  summary.fun=c(mean, sd, length))

df$lower = with(df, estimate_mean - 1.96*estimate_sd/sqrt(estimate_length))
df$upper = with(df, estimate_mean + 1.96*estimate_sd/sqrt(estimate_length))
library("ggplot2")
ggplot(df, aes(y=estimate_mean, x=n, colour=estimator)) + geom_line() +
  geom_ribbon(aes(ymin=lower, ymax=upper, fill=estimator), alpha=0.2) +
  ylim(min(df$lower), max(df$upper)) +
  theme(legend.position=c(0.85,0.85)) +
  ylab("Parameter estimate") + xlab("Sample size (log scaling)") +
  scale_x_log10(breaks=c(100*10^(0:3), 250*10^(0:3), 500*10^(0:2)))
```



Note, instead of calculating the mean and the lower and upper bound by hand, the **simTool** package provides the convenient function `meanAndNormCI`.

8. Concluding remarks

A simulation study usually involve source code that is not directly connected to it, e.g. loops for the varying variables or organization of the result objects. The presented package disengages the researcher from such administrative source code, so the programmer can keep focused on the important aspects, i.e. the functions that generate and analyze the data. In order to conduct a simulation study the researcher has only to specify how the datasets have to be generated and analyzed in form of two `data.frames`, which also gives a nice overview of the defined simulation study. It is possible to keep the generated datasets and all individual result objects or simply the individual result objects or even only a summary of the

result objects. Keeping all information is handy for debugging or inspection of unusual or unexpected results, while discarding as many as possible may be necessary for very memory consuming simulation studies. Parallelizing the replications is very simple by solely specifying the number of CPUs. For researchers familiar with the **parallel** package it is even possible to distribute the simulation studies over many computers by simply passing the cluster to the workhorse function `evalGrids`. In sum, the **simTool** package is a flexible tool for small or large and memory consuming simulation studies. It is very easy to learn and intuitive to handle.

9. Session info

```
sessionInfo()
```

```
R version 3.0.2 (2013-09-25)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=de_DE.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=de_DE.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=de_DE.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=de_DE.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] splines   parallel  stats     graphics  grDevices utils     datasets
[8] methods   base

other attached packages:
 [1] MASS_7.3-29      multiPIM_1.4-1   rpart_4.1-5      polspline_1.1.9
 [5] penalized_0.9-42 survival_2.37-7  lars_1.2         MissMech_1.0.1
 [9] plyr_1.8.1       simTool_1.0.1    knitr_1.6

loaded via a namespace (and not attached):
 [1] colorspace_1.2-4 digest_0.6.4     evaluate_0.5.5   formatR_0.10
 [5] ggplot2_1.0.0    grid_3.0.2       gtable_0.1.2     htmltools_0.2.4
 [9] munsell_0.4.2    proto_0.3-10     Rcpp_0.11.2      reshape_0.8.5
[13] reshape2_1.4     rmarkdown_0.2.59 rticles_1.0      scales_0.2.4
[17] stringr_0.6.2    tools_3.0.2      yaml_2.1.13
```

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. ISSN 1548-7660. URL http://www.jstatsoft.org/v37/i03.

Blanchard G, Dickhaus T, Hack N, Konietschke F, Rohmeyer K, Rosenblatt J, Scheer M, Werft W (2010). "muTOSS - Multiple hypothesis testing in an open software system." *JMLR: Workshop and Conference Proceedings*, **11**, 12–29. URL http://jmlr.org/proceedings/papers/v11/blanchard10a/blanchard10a.pdf.

Blanchard G, Dickhaus T, Hack N, Konietschke F, Rohmeyer K, Rosenblatt J, Scheer M, Werft W (2014). ***mutoss**: Unified multiple testing procedures*. R package version 0.1-8, URL http://CRAN.R-project.org/package=mutoss.

Chan TJ (2014). ***ezsim**: provide an easy to use framework to conduct simulation*. R package version 0.5-5, URL http://CRAN.R-project.org/package=ezsim.

Jamshidian M, Jalal S, Jansen C (2014). "MissMech: An R Package for Testing Homoscedasticity, Multivariate Normality, and Missing Completely at Random (MCAR)." *Journal of Statistical Software*, **56**(6). ISSN 1548-7660. URL http://www.jstatsoft.org/v56/i06.

L'Ecuyer P (1999). "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators." *Operations Research*, **47**(1), 159–164. doi:10.1287/opre.47.1.159. http://pubsonline.informs.org/doi/pdf/10.1287/opre.47.1.159, URL http://pubsonline.informs.org/doi/abs/10.1287/opre.47.1.159.

Redd A (2014). ***harvestr**: A Parallel Simulation Framework*. R package version 0.6.0, URL http://CRAN.R-project.org/package=harvestr.

Ritter SJ, Jewell NP, Hubbard AE (2014). "R Package multiPIM: A Causal Inference Approach to Variable Importance Analysis." *Journal of Statistical Software*, **57**(8). ISSN 1548-7660. URL http://www.jstatsoft.org/v57/i08.

Affiliation:

Marsel Scheer

50968 Cologne
E-mail: scheer@freescience.de