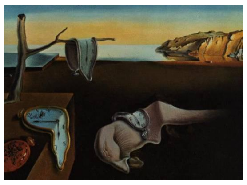

Journal of Statistical Software

February 2016, Volume 69, Issue 4. *doi: 10.18637/jss.v069.i04*

Parallel and Other Simulations in R Made Easy: An End-to-End Study

Marius Hofert
University of Waterloo

Martin Mächler
ETH Zurich

Abstract

It is shown how to set up, conduct, and analyze large simulation studies with the new R package **simsalapar** (= **sim**ulations **s**implified **a**nd **la**unched **par**allel). A simulation study typically starts with determining a collection of input variables and their values on which the study depends. Computations are desired for all combinations of these variables. If conducting these computations sequentially is too time-consuming, parallel computing can be applied over all combinations of select variables. The final result object of a simulation study is typically an array. From this array, summary statistics can be derived and presented in terms of flat contingency or LaTeX tables or visualized in terms of matrix-like figures.

The R package **simsalapar** provides several tools to achieve the above tasks. Warnings and errors are dealt with correctly, various seeding methods are available, and run time is measured. Furthermore, tools for analyzing the results via tables or graphics are provided. In contrast to rather minimal examples typically found in R packages or vignettes, an end-to-end, not-so-minimal simulation problem from the realm of quantitative risk management is given. The concepts presented and solutions provided by **simsalapar** may be of interest to students, researchers, and practitioners as a how-to for conducting realistic, large-scale simulation studies in R.

Keywords: R, simulation, parallel computing, data analysis.

1. Introduction

Realistic mathematical or statistical models are often complex and not analytically tractable, thus require to be evaluated by simulation. In many areas such as finance, insurance, or statistics, it is therefore necessary to set up, conduct, and analyze simulation studies. Apart from minimal examples which address particular tasks, one often faces more difficult setups. For example, if a comparably small simulation already reveals an interesting result, it is often desired to conduct a larger study, involving more parameters, a larger sample size, or more simulation replications. However, run time for sequentially computing results for all variable combinations may now be too large. It may thus be beneficial to apply parallel computing for select variable combinations, be it on a multi-core processor with several central processing units (*cores*), or on a network (*cluster*) with several computers (*nodes*). This adds another level of difficulty to solving the initial task. Users such as students, researchers, or practitioners are typically not primarily interested in the technical details of parallel computing, especially when it comes to more involved tasks such as correctly advancing a random number generator stream to guarantee reproducibility while having different seeds on different nodes. Furthermore, numerical issues often distort simulation results but remain undetected, especially if they happen rarely or are not captured correctly. These issues are either not, or not sufficiently addressed in examples, vignettes, or other packages one would consult when setting up a simulation study.

In what follows, we work with the statistical software R, see R Core Team (2015) and Venables, Smith, and R Core Team (2015) for an introduction. We introduce and present the new R package **simsalapar** – available from the Comprehensive R Archive Network at `https://CRAN.R-project.org/package=simsalapar` – and show how it can be used to set up, conduct, and analyze a simulation study. It extends the functionality of several other R packages, such as **simSummary** (see Gorjanc 2012), **ezsim** (see Chan 2014), **harvestr** (see Redd 2014), and **simFrame** (see Alfons, Templ, and Filzmoser 2010), where e.g., none of these capture warning messages, and only **harvestr** does save errors (for further treatment), where we excel at this. Our emphasis, more than in these packages, is to nicely deal with "multi-factor"[1] simulation setups,

In our view, a simulation study typically consists of the following parts:

1. *Setup*: The scientific problem; how to translate it to a setup of a simulation study; breaking down the problem into different layers and implementing the main, problem-specific function. These tasks are addressed in Sections 2.2–2.6 after introducing our working example in the realm of quantitative risk management in Section 2.1.

2. *Conducting the simulation*: Here, approaches of how to conduct computations in parallel with R are presented. They depend on whether the simulation study is run on one machine (node) with a multi-core processor or on a cluster with several nodes. This is addressed in Section 3.

3. *Analyzing the results*: How results of a simulation study can be presented with tables or graphics. This is done in Section 4.

After a conclusion in Section 5, we show additional and more advanced functions and computations in the appendix. They are not necessary for understanding the paper and rather emphasize what is going on "behind the scenes" of **simsalapar**, provide further functionality, explanations of our approach, and additional checks conducted. As a working example throughout the paper, we present a simulation problem from the realm of quantitative risk management. The example is minimal in the sense that it can still be run sequentially (non-parallel) on a standard computer. However, it is not too minimal in that it covers a wide range of possible variables and tasks a simulation study might depend on and involve. We

[1]Multi-factor: more than one "grid" variable in the variable list, see, e.g., Table 1.

believe this to be useful for users like students, researchers, and practitioners, who often need to implement simulation studies of similar kind, but miss guidance and accompanying tools such as an R package of how this can be achieved.

2. How to set up and conduct a simulation study

2.1. The scientific problem

As a simulation problem, we consider the task of estimating quantiles of the distribution function of the sum of dependent random variables. This is a statistical problem from the realm of quantitative risk management, where the distribution function under consideration is that of the losses a bank faces over a predetermined time horizon in the future. The corresponding quantile function is termed *value-at-risk*; see McNeil, Frey, and Embrechts (2005, p. 38). According to the Basel II rules of banking supervision, banks have to compute value-at-risk at certain high quantiles as a measure of risk they face and money they have to put aside in order to account for such losses and to avoid bankruptcy.

In the language of mathematics, this can be made precise as follows. Let $S_{t,j}$ denote the value of the jth of d stocks at time $t \ge 0$. The value of a portfolio with these d stocks at time t is thus

$$V_t = \sum_{j=1}^{d} \beta_j S_{t,j},$$

where $\beta_1, \dots, \beta_d$ denote the number of shares of stock j in the portfolio. Considering the logarithmic stock prices as *risk factors*, the *risk-factor changes* are given by

$$X_{t+1,j} = \log(S_{t+1,j}) - \log(S_{t,j}) = \log(S_{t+1,j}/S_{t,j}), \quad j \in \{1, \dots, d\}, \tag{1}$$

that is, as the log-returns. Assume that all quantities at time point t, interpreted as today, are known and we are interested in the time point $t+1$, for example one year ahead. The *loss* of the portfolio at $t+1$ can therefore be expressed as

$$\begin{aligned} L_{t+1} = -(V_{t+1} - V_t) &= -\sum_{j=1}^{d} \beta_j (S_{t+1,j} - S_{t,j}) = -\sum_{j=1}^{d} \beta_j S_{t,j} (\exp(X_{t+1,j}) - 1) \\ &= -\sum_{j=1}^{d} w_{t,j} (\exp(X_{t+1,j}) - 1), \end{aligned} \tag{2}$$

that is, in terms of the known weights $w_{t,j} = \beta_j S_{t,j}$ and the unknown risk-factor changes $X_{t+1,j}$, $j \in \{1, \dots, d\}$. *Value-at-risk* (VaR_α) of L_{t+1} at *level* $\alpha \in (0,1)$ is given by

$$\mathrm{VaR}_\alpha(L_{t+1}) = F^-_{L_{t+1}}(\alpha) = \inf\{x \in \mathbb{R} : F_{L_{t+1}}(x) \ge \alpha\}, \tag{3}$$

that is, the α-quantile of the distribution function $F_{L_{t+1}}$ of L_{t+1}; see Embrechts and Hofert (2013) for more details about such functions.

For simplicity, we drop the time index $t+1$ in what follows. Let $\boldsymbol{X} = (X_1, \dots, X_d)$ be the d-dimensional vector of (possibly) dependent risk-factor changes. By Sklar (1959), its distribution function H can be expressed as

$$H(\boldsymbol{x}) = C(F_1(x_1), \dots, F_d(x_d)), \quad \boldsymbol{x} \in \mathbb{R}^d,$$

for a copula C and the marginal distribution functions $F_1, \dots, F_d$ of H. A *copula* is a distribution function with standard uniform univariate margins; for an introduction to copulas, see Nelsen (2006). Our goal is to simulate losses L for margins $F_1, \dots, F_d$ (assumed to be standard normal), a given vector $\boldsymbol{w} = (w_1, \dots, w_d)$ of weights (assumed to be $\boldsymbol{w} = (1, \dots, 1)$), and different

- sample sizes n;
- dimensions d;
- copula families C (note that we slightly abuse notation here and in what follows, using C to denote a parametric copula family, not only a fixed copula);
- copula parameters, expressed in terms of the concordance measure Kendall's tau τ;

and then to compute $\mathrm{VaR}_\alpha(L)$ for different levels α. This is a common setup and problem from quantitative risk management. Since neither F_L, nor its quantile function and thus $\mathrm{VaR}_\alpha(L)$ are known explicitly, we estimate $\mathrm{VaR}_\alpha(L)$ empirically based on n simulated losses L_i, $i \in \{1, \dots, n\}$, of L. This method for estimating $\mathrm{VaR}_\alpha(L)$ is also known as *Monte Carlo simulation method*; see McNeil *et al.* (2005, Section 2.3.3). We repeat it N_{sim} times to be able to provide an error measure of the estimation via bootstrapped percentile confidence intervals.

2.2. Translating the scientific problem to R

To summarize, our goal is to simulate, for each sample size n, dimension d, copula family C, and strength of dependence Kendall's tau τ, N_{sim} times n losses L_{ki}, $k \in \{1, \dots, N_{sim}\}$, $i \in \{1, \dots, n\}$, and to compute in the kth of the N_{sim} replications $\mathrm{VaR}_\alpha(L)$ as the empirical α-quantile of L_{ki}, $i \in \{1, \dots, n\}$, for each α considered. Since different α-quantiles can (and should!) be estimated based on the same simulated losses, we do not have to generate additional samples for different values of α, $\mathrm{VaR}_\alpha(L)$ can be estimated simultaneously for all α under consideration.

Variable	Expression	Type	Value
`n.sim`	N_{sim}	N	32
`n`	n	grid	64, 256
`d`	d	grid	5, 20, 100, 500
`varWgts`	$\mathbf{w}$	frozen	1, 1, 1, 1
`qF`	F^{-1}	frozen	qF
`family`	C	grid	Clayton, Gumbel
`tau`	τ	grid	0.25, 0.50
`alpha`	α	inner	0.950, 0.990, 0.999

Table 1: Variables which determine our simulation study.

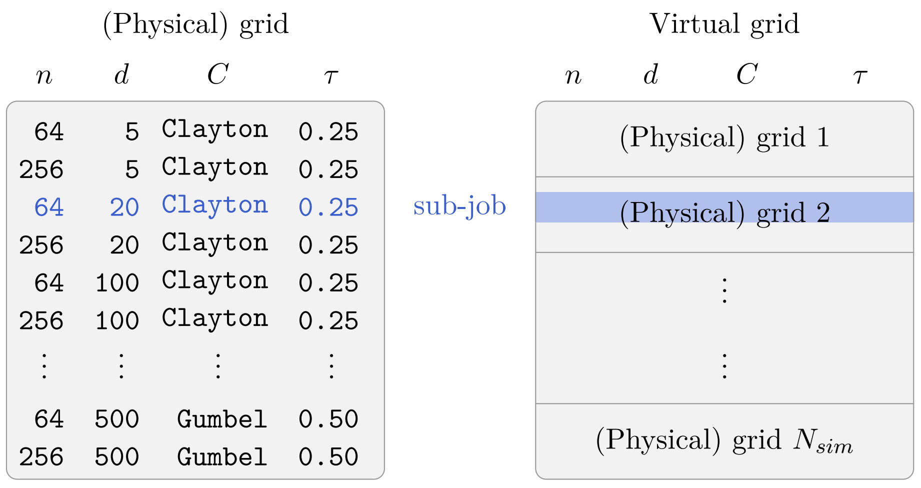

Figure 1: (Physical) vs. virtual grid.

Table 1 provides a summary of all variables involved in our simulation study, their names in R, LaTeX expressions, type, and the corresponding values we choose. Note that this table is produced entirely with **simsalapar**'s `toLatex(varList, ....)`; see page 6. For the moment, let us focus on the type. Available are:

N: The variable N_{sim} gives the number of simulation ("bootstrap") replications in our study. This variable is present in many statistical simulations and allows one to provide an error measure of a statistical quantity such as an estimator. Because of this special meaning, it gets the type "N", and there can be only one variable of this type in a simulation study. If it is not given, it will implicitly be treated as 1.

frozen: The variable $\boldsymbol{w}$ is a list of length equal to the number of dimensions considered, where each entry is a vector (in our case a value which will be sufficiently often recycled by R) of length equal to the corresponding dimension. Variables such as $\boldsymbol{w}$ (or the marginal distribution functions) remain the same throughout our whole simulation study, but one might want to change them if the study is conducted again. Variables of this type are assigned the type "frozen", since they remain fixed throughout the whole study.

grid: Variables of type "grid" are used to build a *(physical) grid*. In R this grid is implemented as a data frame. Each row in this data frame contains a unique combination of variables of type "grid". The number of rows n_G of this grid, is thus the product of the lengths of all variables of type "grid". The simulation will iterate N_{sim} times over all n_G rows and conduct the required computations. Conceptually, this corresponds to visiting each of the $N_{sim} \times n_G$ rows of a *virtual grid* seen as N_{sim} copies of the grid pasted together; see Figure 1. The computations for one row in this virtual grid are viewed as one *sub-job*. If computing all sub-jobs sequentially turns out to be time-consuming and profiling of the code did not reveal major possible improvements such as removing deeply nested 'for' loops, we can apply parallel computing and distribute the sub-jobs over several cores of a multi-core processor or several machines (nodes) in a cluster.

inner: Finally, variables of type "inner" are all dealt with within a sub-job for reasons such as convenience, speed, or load balancing. As mentioned before, in our example, α plays such a role since $\mathrm{VaR}_\alpha(L)$ can be estimated simultaneously for all α under consideration based on the same simulated losses.

As result object of a simulation, we naturally obtain an array. This array has one dimension for each variable of type "grid" or "inner", and one additional dimension if $N_{sim} > 1$. Besides the variable names, their type, and their values, we also define R expressions for each variable. These expressions are later used to label tables or plots when the simulation results are analyzed.

We are now ready to start writing an R script which can be run on a single computer or on a computer cluster. Since cluster types and interfaces are quite different, we only focus on how to write the R script here[2]. The first task is to implement the variable list presented above. Note that `varlist()` is a constructor for an object of the S4 class `"varlist"`, which is only little more than the usual `list()` in R. For more details, use `require("simsalapar")`, then `?varlist`, `getClass("varlist")`, or `class?varlist`. Given a user-provided variable list of class `"varlist"`, a table such as Table 1 can be automatically generated with the `toLatex.varlist` method.

```
R> require("simsalapar")
R> varList <- varlist(
+    n.sim = list(type = "N", expr = quote(N[sim]), value = 32),
+    n = list(type = "grid", value = c(64, 256)),
+    d = list(type = "grid", value = c(5, 20, 100, 500)),
+    varWgts = list(type = "frozen", expr = quote(bold(w)),
+      value = list("5" = 1, "20" = 1, "100" = 1, "500" = 1)),
+    qF = list(type = "frozen", expr = quote(F^{-1}),
+      value = list(qF = qnorm)),
+    family = list(type = "grid", expr = quote(C),
+      value = c("Clayton", "Gumbel")),
+    tau = list(type = "grid", value = c(0.25, 0.5)),
+    alpha = list(type = "inner", value = c(0.95, 0.99, 0.999)))
R> toLatex(varList, label = "tab:var",
+    caption = "Variables which determine our simulation study.")
```

One actually does not need to specify a type for `n.sim` or variables of type "frozen" as the default chosen is "frozen" unless the variable is `n.sim` in which case it is "N".

The function `getEl()` can be used to extract elements of a certain type from a variable list (defaults to all values).

[2]As an example of how to run an R script `simu.R` on different nodes on a computer cluster, let us consider the cluster *Brutus* at ETH Zurich. It runs an LSF batch system. Once logged in, one can submit the script `simu.R` via `bsub -N -W 01:00 -n 48 -R "select[model==Opteron8380]" -R "span[ptile=16]" mpirun -n 1 R CMD BATCH simu.R`, for example, where the meaning of the various options is as follows: `-N` sends an email to the user when the batch job has finished; `-W 01:00` submits the job to the queue with maximal wall-clock run time of one hour; the option `-n 48` asks for 48 cores, one is used as manager and 47 as workers; `-R "select[model==Opteron8380]"` specifies X86_64 nodes with AMD Opteron 8380 CPUs for the sub-jobs to be run (this is important if run-time comparisons are required, since one has to make sure that the same architecture is used when computations are carried out in parallel); the option `-R "span[ptile=16]"` specifies that 16 cores are used on a single node; `mpirun` specifies an Open MPI job which runs only one copy (`-n 1`) of the program; and finally, `R CMD BATCH simu.R` is the standard call of the R script `simu.R` in batch mode.

```
R> str(getEl(varList, "grid"))
```

```
List of 4
 $ n     : num [1:2] 64 256
 $ d     : num [1:4] 5 20 100 500
 $ family: chr [1:2] "Clayton" "Gumbel"
 $ tau   : num [1:2] 0.25 0.5
```

```
R> str(getEl(varList, "inner"))
```

```
List of 1
 $ alpha: num [1:3] 0.95 0.99 0.999
```

To have a look at the (physical) grid for our working example which contains all combinations of variables of type "grid", the function `mkGrid()` can be used as follows.

```
R> pGrid <- mkGrid(varList)
R> str(pGrid)
```

```
'data.frame':        32 obs. of  4 variables:
 $ n     : num  64 256 64 256 64 256 64 256 64 256 ...
 $ d     : num  5 5 20 20 100 100 500 500 5 5 ...
 $ family: chr  "Clayton" "Clayton" "Clayton" "Clayton" ...
 $ tau   : num  0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 ...
```

2.3. The result of a simulation

Our route from here is to conduct the simulations required for each line of the virtual grid. As an important point, note that each computational *result* naturally consists of the following components:

`value`: The actual value. This is can be a scalar, numeric vector, or numeric array whose dimensions depend on variables of type "inner". The computed entries also depend on variables of type "frozen", but they do not enter the array as additional dimensions.

`error`: It is important to adequately track errors during simulation studies. If one computation fails, we lose all results computed so far and thus have to do the work again (fix the error, move the files to the cluster, wait for the simulation job to start, wait for it to fail or to finish successfully in this next trial run etc.). To avoid this, we capture the errors to be able to deal with them after the simulation has been conducted. This also allows us to compute statistics about errors, such as percentages of runs producing errors etc.

`warning`: Similar to errors, warnings are important to catch. They may indicate non-convergence of an algorithm or a maximal number of iterations reached and therefore impact reliability of the results.

`time`**:** Measured run time can also be an indicator of reliability in the sense that if computations are too fast or too slow, there might be a programming error such as a wrong logical condition leading the program to end up in a wrong case. If the value computed from this case is not suspicious, and if there were no warnings and errors, then run time is the only indicator of a possible bug in the code. Furthermore, measuring run time is also helpful for benchmarking or determining whether a computation or algorithm runs sufficiently fast on a notebook.

`.Random.seed`**:** The random seed right before the user-specified computations are carried out. This is useful for reproducing single results for debugging purposes.

In many simulation studies, also on an academic level, focus is put on `value` only. We therefore particularly stress all of these components, since they become more and more important for obtaining reliable results the larger the conducted simulation study is. Furthermore, `error`, `warning`, and `.Random.seed` are important to consider especially during experimental stage of the simulation, for checking an implementation, and testing it for numerical stability.

The paradigm of **simsalapar** is that the user only has to take care of how to compute the `value`, that is, the statistic the user is most interested in. All other components addressed above are automatically dealt with by **simsalapar**. We will come back to this in Section 2.5, after having thought about how to compute the `value` for our working example in the following section.

2.4. Implementing the problem-specific function `doOne()`

Programming in R is about writing *functions*. Our goal is now to implement the workhorse of the simulation study: `doOne()`. This function has to be designed for the particular simulation problem at hand. It computes the component `value` (a numeric vector of VaR_α) for the given arguments `n` (sample size), `d` (dimension), `qF` (marginal quantile function), `family` (copula family), `tau` (Kendall's tau), `alpha` (vector of VaR_α levels), and `varWgts` (vector of weights). For functions `doOne()` for other simulation examples, we refer to the demos of **simsalapar**, see for example `demo(TGforecasts)` for reproducing the simulation conducted by Gneiting (2011).

```
R> doOne <- function(n, d, qF, family, tau, alpha, varWgts, names = FALSE)
+  {
+    w <- varWgts[[as.character(d)]]
+    stopifnot(require("copula"), sapply(list(w, alpha, tau, d), is.numeric))
+    simRFC <- function(n, d, qF, family, tau) {
+      theta <- getAcop(family)@iTau(tau)
+      cop <- onacopulaL(family, list(theta, 1:d))
+      qF(rCopula(n, cop))
+    }
+    X <- simRFC(n, d = d, qF = qF[["qF"]], family = family, tau = tau)
+    L <- -rowSums(expm1(X) * matrix(rep(w, length.out = d),
+      nrow = n, ncol = d, byrow = TRUE))
+    quantile(L, probs = alpha, names = names)
+  }
```

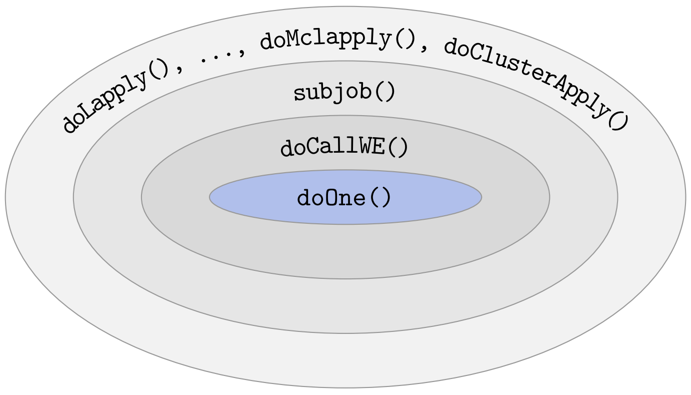

Figure 2: Layers of functions involved in a simulation study. **simsalapar** provides all but `doOne()`.

2.5. Putting the pieces together: The `do*()` functions

To conduct the main simulation, we only need one more function which iterates over all sub-jobs and calls `doOne()`. There are several options: sequential (see Section 2.6) versus various approaches for parallel computing (see Section 3), for which we provide the `do*()` functions explained below. Since these functions are quite technical and lengthy, we will present the details in the appendix. For the moment, our goal is to understand the functions they call in order to understand how the simulation works. Figure 2 visualizes the main functions involved in conducting the simulation. These functions break down the whole task into smaller pieces. This improves readability of the code and simplifies debugging when procedures fail.

We have already discussed the innermost, user-provided function `doOne()`. The auxiliary function `doCallWE()` captures the values computed by `doOne()`, errors, warnings, and run times when calling `doOne()`. This already provides us with a list of four of the five components of a result as addressed in Section 2.3. The component `.Random.seed` may[3] then be added by the function which calls `doCallWE()`, namely `subjob()`. The aim of `subjob()` is to compute one sub-job, that is, one row of the virtual grid. A large part of this function deals with correctly setting the seed. It also provides a monitor feature.

As mentioned before, there are several choices available for the outermost layer of functions, depending on whether and what kind of parallel computing should be used to deal with the rows of the virtual grid. In particular, **simsalapar** provides the following functions:

`doLapply()`: a wrapper for the non-parallel function `lapply()`. This is useful for testing the code with a small number of different parameters so that the simulation still runs locally on the computer at hand.

`doForeach()`: a wrapper for the function `foreach()` of the R package **foreach** to conduct computations in parallel on several cores or nodes; see Revolution Analytics and Weston (2015b) and Kane, Emerson, and Weston (2013).

`doRmpi()`: a wrapper for the function `mpi.apply()` or its load-balancing version `mpi.applyLB()`

[3] `subjob`'s default `keepSeed=FALSE` has been chosen to avoid large result objects.

(default) from the R package **Rmpi** for parallel computing on several cores or nodes; see Yu (2002).

`doMclapply()`: a wrapper for the function `mclapply()` (with (default) or without load-balancing) of the base R package **parallel** for parallel computing on several cores; see `vignette("parallel")`. Note that this approach is not available on Windows.

`doClusterApply()`: a wrapper for the function `clusterApply()` or its load-balancing version `clusterApplyLB()` (default) of the R package **parallel** for parallel computing on several cores or nodes.

The user of **simsalapar** can call one of the above functions `do*()` to finally run the whole simulation study; see Sections 2.6 and 3. To this end, these functions iterate over all sub-jobs and finally call the function `saveSim()`. `saveSim()` tries to convert the resulting list of lists of length four or five containing the components `value`, `error`, `warning`, `time`, and, possibly, `.Random.seed` to an array of lists of length four or five and saves it in the `.rds` file specified by the argument `sfile`. If this non-trivial conversion fails[4], the raw list of lists of length four or five is saved instead, so that results are not lost. This behavior can also be obtained by directly specifying `doAL = FALSE` when calling the `do*()` functions. To further avoid that the conversion fails, the functions `do*()` conduct a basic check of the correctness of the return value of `doOne()` by calling the function `doCheck()`. This can also be called by the user after implementing `doOne()` to verify the correctness of `doOne()`; see, for example, `demo(VaRsuperadd)`.

2.6. Running the simulation sequentially: `doLapply()` based on `lapply()`

In Section 3 and the appendix, we will compare different approaches for parallel computing in R. To make this easier to follow, we start with `doLapply()` which is a wrapper for the sequential function `lapply()` to iterate over all rows of the virtual grid. This approach is often the first choice to try in order to check whether the simulation actually does what it should or for debugging purposes based on a smaller number of parameter combinations. If sequential computations based on `lapply()` turn out to be too slow, one can easily use one of the parallel computing approaches described in Section 3, since they share the same interface. This is one major advantage of the functionality provided by **simsalapar**.

We now demonstrate the use of `doLapply()` to run the whole simulation. Note that `names` is an optional argument to our `doOne()` and the argument `monitor`, passed to `subjob()`, allows progress monitoring.

```
R> res <- doLapply(varList, sfile = "res_lapply_seq.rds", doOne = doOne,
+    names = TRUE, monitor = interactive())
```

The structure of the resulting object can briefly be analyzed as follows.

```
R> str(res, max.level = 2)
```

[4]Our flexible approach allows one to implement a function `doOne()` such that the order in which the "inner" variables appear does not correspond to the order in which they appear in the variable list. Therefore, the user-provided workhorse `doOne()` has to be written with care.

```
List of 1024
 $ :List of 4
  ..$ value  : num [1:3(1d)] 3.18 3.6 4.02
  .. ..- attr(*, "dimnames")=List of 1
  ..$ error  : NULL
  ..$ warning: NULL
  ..$ time   : num 11
 $ :List of 4
  ..$ value  : num [1:3(1d)] 3.36 4.35 4.68
  .. ..- attr(*, "dimnames")=List of 1
  ..$ error  : NULL
  ..$ warning: NULL
  ..$ time   : num 1
 .......
 .......
  [list output truncated]
 - attr(*, "dim")= Named int [1:5] 2 4 2 2 32
  ..- attr(*, "names")= chr [1:5] "n" "d" "family" "tau" ...
 - attr(*, "dimnames")=List of 5
  ..$ n     : chr [1:2] "64" "256"
  ..$ d     : chr [1:4] "5" "20" "100" "500"
  ..$ family: chr [1:2] "Clayton" "Gumbel"
  ..$ tau   : chr [1:2] "0.25" "0.50"
  ..$ n.sim : NULL
 - attr(*, "fromFile")= logi TRUE
```

```
R> str(dimnames(res))
```

```
List of 5
 $ n     : chr [1:2] "64" "256"
 $ d     : chr [1:4] "5" "20" "100" "500"
 $ family: chr [1:2] "Clayton" "Gumbel"
 $ tau   : chr [1:2] "0.25" "0.50"
 $ n.sim : NULL
```

3. Parallel computing in R

For a tutorial on parallel computing in R, we refer the interested reader to Eugster, Knaus, Porzelius, Schmidberger, and Vicedo (2011). Roughly speaking, the main idea behind parallelization is to run independent parts of a program on several cores or nodes simultaneously in order to reduce the overall run time. In this section, we show how **simsalapar** can be used to conduct a simulation in parallel.

3.1. A word of warning

As a first remark, we would like to stress that parallel computing is not a panacea for all computational problems. Indeed, a good design avoiding time bombs such as nested for-loops, proper profiling, and maybe implementing certain parts in C (such as rejection algorithms) can already lead to a significantly reduced run time. If running a program sequentially is feasible, this is the preferred way. If not, or if a user simply does not possess the knowledge to implement parts in C, parallelization becomes interesting. However, if not applied properly, parallel computing can even increase the run time if, for example, each sub-job takes less time than the overhead of the communication between the workers; see also Appendix A.4.

Also, we would like to mention that there are other approaches to parallel computing not discussed here, **simsalapar** simply considers the most common approaches for parallel computing in R which do not require a significantly deeper knowledge outside R. An approach not discussed here is based on Hadoop; see, for example, `http://www.datadr.org/` or `http://www.rdatamining.com/tutorials/r-hadoop-setup-guide`.

3.2. Parallel computing with simsalapar

In the same way that `doLapply()` wraps around `lapply()`, **simsalapar** provides convenient wrapper functions `do*()` for conducting computations *in parallel*. These functions use different approaches for parallel computing. One should only use *one* of them in the same R session as mixing several different ways of conducting parallel computations in the same R process may lead to weird errors, conflicts of various kinds, or unreliable results at best.

The different approaches for parallel computing are useful under different setups and may depend on the available computer architecture or different specifications of the simulation study considered. The paradigm **simsalapar** follows is that one just needs to replace `doLapply()` above (Section 2.6) by one of its *"parallelized"* `do*()` versions listed in Section 2.5 in order to conduct the computations in parallel. We will take `doClusterApply()` as an example here and refer to Section A for a more in-depth analysis and comparison of the results obtained from these different approaches to those from `doLapply()` to check their correctness, consistency, and efficiency.

```
R> res5 <- doClusterApply(varList, sfile = "res5_clApply_seq.rds",
+     doOne = doOne, names = TRUE)
```

`doClusterApply()` produces the same result as `doLapply()` which can be verified with the function `doRes.equal()` from **simsalapar**.

```
R> stopifnot(doRes.equal(res5, res))
```

Note that if there are special requirements, the (more technically inclined) user can implement his/her own `do*()` function; see Appendix A.6 for an example.

4. Data analysis

After having conducted the main simulation, the final task is to analyze the data and present the results. It seems difficult to provide a general solution for this part of a simulation study. Besides the solutions provided by **simsalapar** however, it might therefore be required to implement additional problem-specific functions. In this case, functions from **simsalapar** may at least serve as a good starting point.

The function `getArray()` is a function from **simsalapar** which, given the result object of the simulation and one of the components "value" (the default), "error", "warning", or "time" creates an array containing the corresponding results. This is typically more convenient than working with an array of lists as returned by one of the `do*()` functions. For the components being "error" or "warning", the array created contains by default boolean variables indicating whether there was an error or warning, respectively; this behavior can be changed by providing a suitable argument `FUN` to `getArray()`. Additionally, `getArray()` allows for an argument `err.value`, defaulting to `NA`, for replacing values in case there was an error. As mentioned before, each "value", can be a scalar, a numeric vector, or a numeric array, often with `dimnames`, for example, resulting from the outer product of variables of type "inner". Note that for conducting the simulation, variables sometimes can be declared as "inner" or "frozen" interchangeably. However, this changes the dimension of the result object for the analysis in the sense that variables of type "inner" appear as additional dimensions in the result array and can thus serve as a proper quantity or dimension in a table or plot, whereas variables of type "frozen" do not.

In the following, we work with the R object `res` as returned by `doLapply()`[5]. To extract values, error and warning indicators, and run times (in ms) from it, we simply apply `getArray()` to `res`,

```
R> val  <- getArray(res)
R> err  <- getArray(res, "error")
R> warn <- getArray(res, "warning")
R> time <- getArray(res, "time")
```

and a `data.frame` can easily be produced from our array of values via `array2df()`:

```
R> df <- array2df(val)
R> str(df)
```

```
'data.frame':        3072 obs. of  7 variables:
 $ alpha : Factor w/ 3 levels "95%","99%","99.9%": 1 2 3 1 2 3 1 2 3 1 ...
 $ n     : Factor w/ 2 levels "64","256": 1 1 1 2 2 2 1 1 1 2 ...
 $ d     : Factor w/ 4 levels "5","20","100",..: 1 1 1 1 1 1 2 2 2 2 ...
 $ family: Factor w/ 2 levels "Clayton","Gumbel": 1 1 1 1 1 1 1 1 1 1 ...
 $ tau   : Factor w/ 2 levels "0.25","0.50": 1 1 1 1 1 1 1 1 1 1 ...
 $ n.sim : Factor w/ 32 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ value : num  3.18 3.6 4.02 3.36 4.35 ...
```

As a first part of the analysis, we are interested in how reliable our results are. We thus consider possible errors and warnings of the computations conducted. Flat contingency tables allow us to conveniently get an overview of errors or warnings.

```
R> rv <- c("family", "d")
R> cv <- c("tau", "n")
R> ftable(100 * (err + warn), row.vars = rv, col.vars = cv)
```

[5]We have shown above that the other `do<...>()` functions produce the same object as `res`.

```
            tau 0.25     0.50
            n     64 256   64 256
family  d
Clayton 5          0   0    0   0
        20         0   0    0   0
        100        0   0    0   0
        500        0   0    0   0
Gumbel  5          0   0    0   0
        20         0   0    0   0
        100        0   0    0   0
        500        0   0    0   0
```

Since we do not have errors or warnings in our numerically non-critical example study here, let us briefly consider the run times.

```
R> ftable(time, row.vars = rv, col.vars = cv)
```

```
            tau 0.25      0.50
            n     64  256   64  256
family  d
Clayton 5         50   55   34   42
        20        43   81   42   84
        100       86  268   87  264
        500      316 1380  318 1390
Gumbel  5         37   47   34   48
        20        46   84   45   87
        100       95  288   96  277
        500      345 1574  444 1413
```

```
R> dtime <- array2df(time)
R> summary(dtime)
```

```
   n         d          family       tau          n.sim
 64 :512   5  :256   Clayton:512   0.25:512   1      : 32
 256:512   20 :256   Gumbel :512   0.50:512   2      : 32
           100:256                            3      : 32
           500:256                            4      : 32
                                              5      : 32
                                              6      : 32
                                              (Other):832
     value
 Min.   :  0.000
 1st Qu.:  1.000
 Median :  3.000
 Mean   :  9.277
 3rd Qu.:  9.000
 Max.   :145.000
```

In what follows, we exclusively focus on the actual computed values, hence the array `val`. We apply tools from **simsalapar** that allow us to create flexible LaTeX tables and sophisticated graphs for representing these results.

4.1. Creating LaTeX tables

In this section, we create LaTeX tables of the simulation results. Our goal is to make this process modular and flexible. We thus leave tasks such as formatting of table entries as much as possible to the user. Note that there are already R packages available for generating LaTeX tables, for example the well-known **xtable** or the rather new **tables**; see Dahl (2016) and Murdoch (2014), respectively. However, they do not fulfill the above requirements and come with other unwanted side effects concerning the table headers or formatting of entries we do not want to cope with. We therefore present new tools for constructing tables with **simsalapar**. For inclusion in LaTeX documents, only the LaTeX package **tabularx**, and (optionally, if using the default `booktabs = TRUE` in the following functions), the LaTeX package **booktabs** have to be loaded in the `.tex` document. Much more sophisticated alignment of column entries for LaTeX tables than we show here, possibly including units, can be achieved in combination with the LaTeX package **siunitx**. Note that these packages all come with standard LaTeX distributions.

After having computed arrays of robust value-at-risk estimates and robust standard deviations via

```
R> non.sim.margins <- setdiff(names(dimnames(val)), "n.sim")
R> huber. <- function(x) MASS::huber(x)$mu
R> VaR <- apply(val, non.sim.margins, huber.)
R> VaR.mad <- apply(val, non.sim.margins, mad)
```

we format and merge the arrays. As just mentioned, we specifically leave this task to the user to guarantee flexibility. As an example, we put the robust standard deviations in parentheses and colorize[6] all entries corresponding to the largest level α.

```
R> fval <- formatC(VaR, digits = 1, format = "f")
R> fmad <- paste0("(", format(round(VaR.mad, 1),
+    scientific = FALSE, trim = TRUE), ")")
R> nc <- nchar(fmad)
R> sm <- nc == min(nc)
R> fmad[sm] <- paste0("\\ \\,", fmad[sm])
R> fres <- array(paste(fval, fmad), dim = dim(fval),
+    dimnames = dimnames(fval))
R> ia <- dim(fval)[1]
R> fres[ia, , , , ] <- paste("\\color{white!40!black}", fres[ia, , , , ])
```

Next, we create a flat contingency table from the array of formatted results `fres`. The arguments `row.vars` and `col.vars` of `ftable()` specify the basic layout of Table 2 below.

[6]This requires the LaTeX package **xcolor** with the option `table` to be loaded in the LaTeX document. The latter option even allows to use `\cellcolor` to modify the background colors of select table cells.

		τ	0.25			0.50		
C	n	$d \mid \alpha$	95%	99%	99.9%	95%	99%	99.9%
Clayton	64	5	3.1 (0.4)	3.8 (0.4)	4.0 (0.5)	3.6 (0.3)	4.2 (0.2)	4.4 (0.2)
		20	10.6 (1.4)	13.5 (1.5)	14.8 (2.2)	14.2 (1.6)	16.7 (1.0)	17.4 (1.0)
		100	46.1 (9.1)	63.5 (11.6)	68.5 (13.6)	70.7 (8.6)	83.7 (3.9)	86.7 (4.2)
		500	224.8 (50.6)	307.8 (61.5)	336.0 (66.8)	350.0 (40.5)	418.6 (22.3)	434.0 (21.4)
	256	5	3.2 (0.2)	4.1 (0.2)	4.4 (0.2)	3.9 (0.2)	4.4 (0.1)	4.6 (0.1)
		20	10.9 (1.0)	15.3 (1.2)	17.0 (0.9)	15.3 (0.7)	17.6 (0.5)	18.5 (0.6)
		100	49.0 (5.5)	72.1 (7.7)	82.5 (4.8)	76.0 (3.4)	87.9 (2.7)	92.3 (3.0)
		500	240.4 (27.0)	349.7 (35.3)	408.5 (24.3)	378.8 (17.4)	439.4 (12.7)	461.7 (14.2)
Gumbel	64	5	2.7 (0.3)	3.3 (0.4)	3.4 (0.5)	3.3 (0.3)	3.8 (0.3)	4.0 (0.2)
		20	7.3 (1.1)	9.4 (1.2)	10.1 (1.5)	12.2 (0.6)	14.0 (1.2)	14.6 (1.2)
		100	26.0 (4.2)	35.8 (4.7)	38.5 (5.6)	57.7 (5.1)	67.7 (4.8)	70.3 (5.4)
		500	117.2 (12.5)	154.4 (19.0)	167.5 (18.2)	288.2 (18.0)	333.7 (23.0)	347.9 (20.7)
	256	5	2.7 (0.2)	3.3 (0.2)	3.7 (0.2)	3.4 (0.2)	3.9 (0.1)	4.2 (0.1)
		20	7.4 (0.5)	9.9 (0.8)	11.5 (0.9)	12.5 (0.4)	14.7 (0.7)	16.0 (0.6)
		100	27.8 (2.8)	38.4 (3.1)	44.7 (3.2)	60.4 (2.3)	70.9 (2.5)	76.9 (3.5)
		500	126.8 (10.3)	171.9 (11.2)	202.3 (13.5)	299.1 (13.7)	353.8 (13.2)	380.0 (9.7)

Table 2: Simulation results; the table is the automatic Sweave rendering of `tabL <- toLatex(ft, ..)`, above.

```
R> ft <- ftable(fres, row.vars = c("family", "n", "d"),
+    col.vars = c("tau", "alpha"))
```

Table 2 shows the results of applying our `toLatex()` method[7]

```
R> tabL <- toLatex(ft, vList = varList, fontsize = "scriptsize",
+    caption = "Simulation results; the table is the automatic Sweave
+      rendering of \\code{tabL <- toLatex(ft, ..)}, above.",
+    label = "tab:ft")
```

To summarize, using functions from **simsalapar** and packages from LaTeX, one can create flexible LaTeX tables. If the simulation results become sufficiently complicated, creating (at least parts of) LaTeX tables from R reduces a lot of work, especially if the simulation study has to be repeated due to bug fixes, improvements, or changes in the implementation. Note that the table header typically constitutes the main complication when constructing tables. It might still be required to manually modify it in case our carefully chosen defaults are insufficient. Package **simsalapar** provides many other functions not presented here, including the currently non-exported function `ftable2latex()` and the exported and documented functions `fftable()`, `tablines()` and `wrapLaTable()`. These auxiliary functions can be useful if one encounters very specific requirements not covered by `toLatex.ftable()`.

A crucial step in the development of `tablines()` was the correct formatting of an `ftable` without introducing empty rows or columns. For this we introduced four different methods of "compactness" of a formatted `ftable` which are available in `format.ftable()` from R version 3.0.0 and for earlier versions in **simsalapar**.

4.2. Graphical analysis

Next we show how **simsalapar** can be applied to visualize the results of our study. In modern

[7]The `toLatex()` method for `"ftable"`'s, i.e., `toLatex.ftable()`.

statistics, displaying results with graphics is typically preferred to tables, since it is easier to see the story the data would like to tell us. For example, in a table, the human eye can only compare two numbers at a time, in well-designed graphics much more information is visible.

There are various different approaches of how to create graphics in R, for example, with the traditional **graphics** package, the **lattice**, or the **ggplot2** package; see R Core Team (2015), Sarkar (2008), and Wickham (2009), respectively. The most flexible approach is based on **grid** graphics; see Murrell (2006). In what follows, we apply **simsalapar**'s `mayplot()` for creating a plot matrix from an array of values. Within each cell of this *conditioning plot* a traditional graphic is drawn to visualize the results; this is achieved using **grid** and **graphics** via **gridBase**, see Murrell (2014).

In our example study, the strength of dependence in terms of Kendall's tau determines the columns of the matrix-like plot and the copula family determines its rows. In each cell, there is an x and a y axis. For making comparisons easier, one typically would like to have the same limits on the y axes across different rows of the plot matrix. Sometimes it makes sense to have separate scales for the y axes in different rows. This behavior can be determined with the argument `ylim` (being `"global"` (the default) or `"local"`) of `mayplot()`. For our working example, the x axis provides the different significance levels α. We thus naturally can depict three different input variables in such a layout (copula families, Kendall's taus, and significance levels α). The y axis may show point estimates or boxplots of the simulated value-at-risk values as given in `val`.

All other variables (sample sizes n, dimensions d) then have to be depicted in the same cell, visually distinguished by different line types or colors, for example; currently one such variable is allowed, we chose d and fix $n = 256$. If more variables are involved, one might even want to put more variables in one cell, rethink the design, or split different values of a variable over separate plots. N_{sim}, if available, enters the scene through a second label on the right side of the graphic.

With `mayplot()` it is easy to create a graphical result in the form of a pdf file, for example[8]. Figures 3 and 4 display the results for $n = 256$.

```
R> v256 <- val[, n = "256",,,, ]
R> dimnames(v256)[["tau"]] <- paste0("tau==", dimnames(v256)[["tau"]])

R> mayplot(v256, varList, row.vars = "family", col.vars = "tau", xvar = "d",
+    ylab = bquote(widehat(VaR)[alpha](italic(L))),
+    pcol = c("black", "blue", "red"))
```

The former shows boxplots of all the N_{sim} simulated value-at-risk estimates $\widehat{\mathrm{VaR}}_\alpha(L)$, whereas the latter depicts corresponding robust Huber "means" and also demonstrates `mayplot()` for $N_{sim} = 1$ or, equivalently, no N_{sim} at all. Overall, we see that a graphic such as Figure 3 is easier to grasp and to infer conclusions from than Table 2.

```
R> varList. <- set.n.sim(varList, 1)
R> dimnames(VaR)[["tau"]] <- paste0("tau==", dimnames(VaR)[["tau"]])
```

[8]Note that we use the system tool `pdfcrop` to crop the graph after it is generated. This allows one to perfectly align the graph in a LaTeX (`.tex`) or Sweave (`.Rnw`) document.

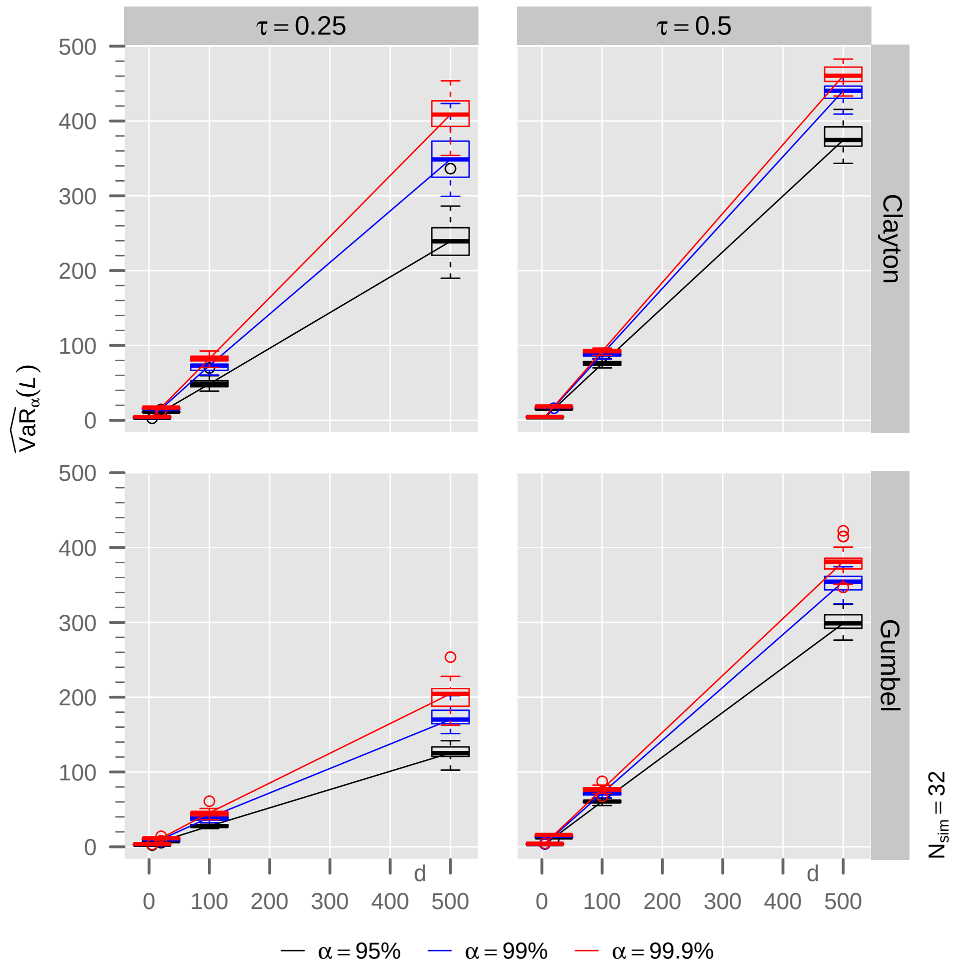

Figure 3: Boxplots of the N_{sim} simulated $\mathrm{VaR}_\alpha(L)$ values for $n = 256$.

```
R> mayplot(VaR[, n = "256",,, ], varList., row.vars = "family",
+     col.vars = "tau", xvar = "d", type = "b", log = "y",
+     axlabspc = c(0.15, 0.08), ylab = bquote(widehat(VaR)[alpha](italic(L))),
+     pcol = c("black", "blue", "red"))
```

4.3. Interpreting the results

Let us briefly interpret the results we have obtained in the context of our working example from Section 2.1. For this, consider Figure 4. Up to the Monte Carlo error, we can infer the following properties:

α: $\mathrm{VaR}_\alpha(L)$ is increasing in α. This is easily verified as higher levels α correspond to larger α-quantiles, thus $\mathrm{VaR}_\alpha(L)$.

τ: $\mathrm{VaR}_\alpha(L)$ is increasing in τ. This can be verified numerically via:

```
R> stopifnot(VaR[, n = "256",,, "tau==0.25"] <=
+     VaR[, n = "256",,, "tau==0.50"])
```

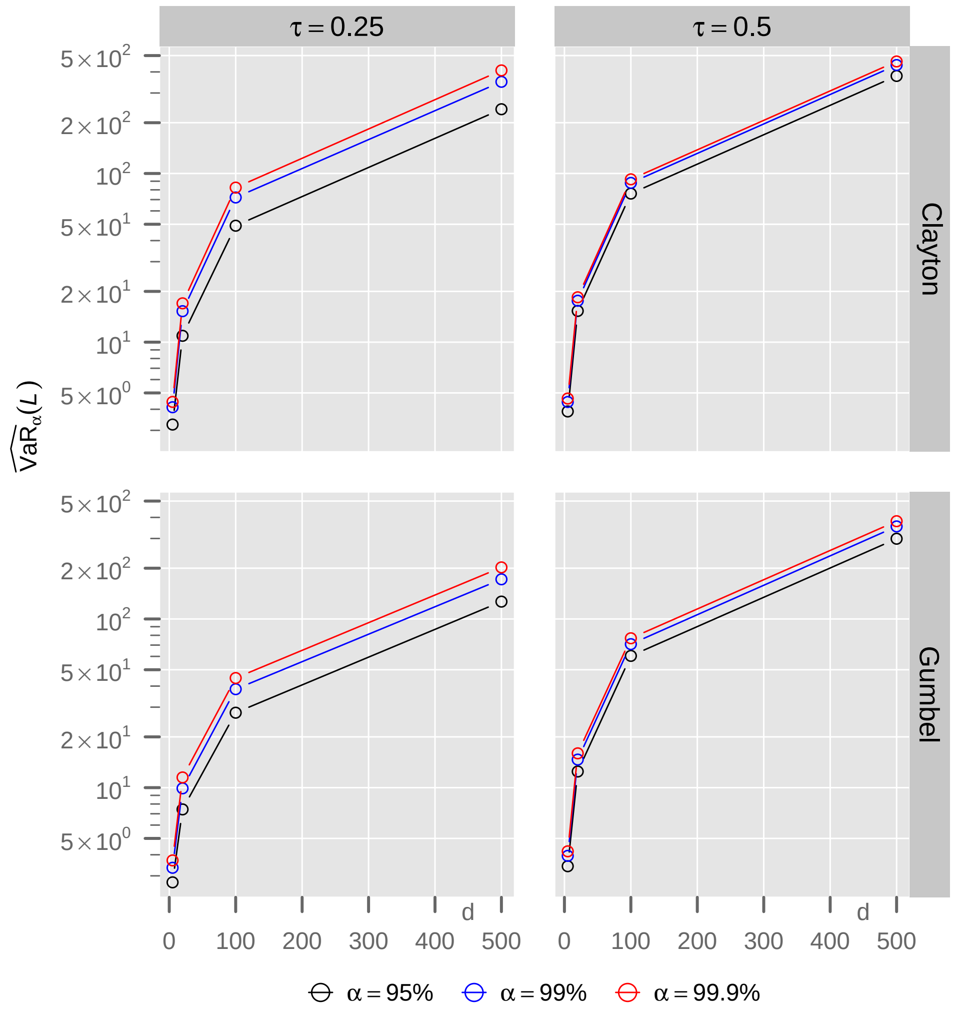

Figure 4: Plot of robust $\mathrm{VaR}_\alpha(L)$ estimates in log scale, that is, Huber "means" of N_{sim} values of Figure 3 for $n = 256$.

To explain this behavior, note that for θ converging to ∞, both the Clayton and the Gumbel copula converge (pointwise) to the upper Fréchet–Hoeffding bound (the copula of a comonotone random vector). Since larger Kendall's tau imply larger parameters θ for the two copulas, a larger tau increases the probability that the components of $\boldsymbol{X} = (X_1, \dots, X_d)$ are simultaneously small or large. By (2),

$$L = \sum_{j=1}^{d} (1 - \exp(X_j)), \tag{4}$$

which directly implies that the tails of F_L get heavier with larger tau. This in turn implies a larger $\mathrm{VaR}_\alpha(L)$.

C: $\mathrm{VaR}_\alpha(L)$ for Clayton is larger than for Gumbel. This can be verified numerically via:

```
R> stopifnot(VaR[, n = "256",, "Gumbel",] <=
+     VaR[, n = "256",, "Clayton",])
```

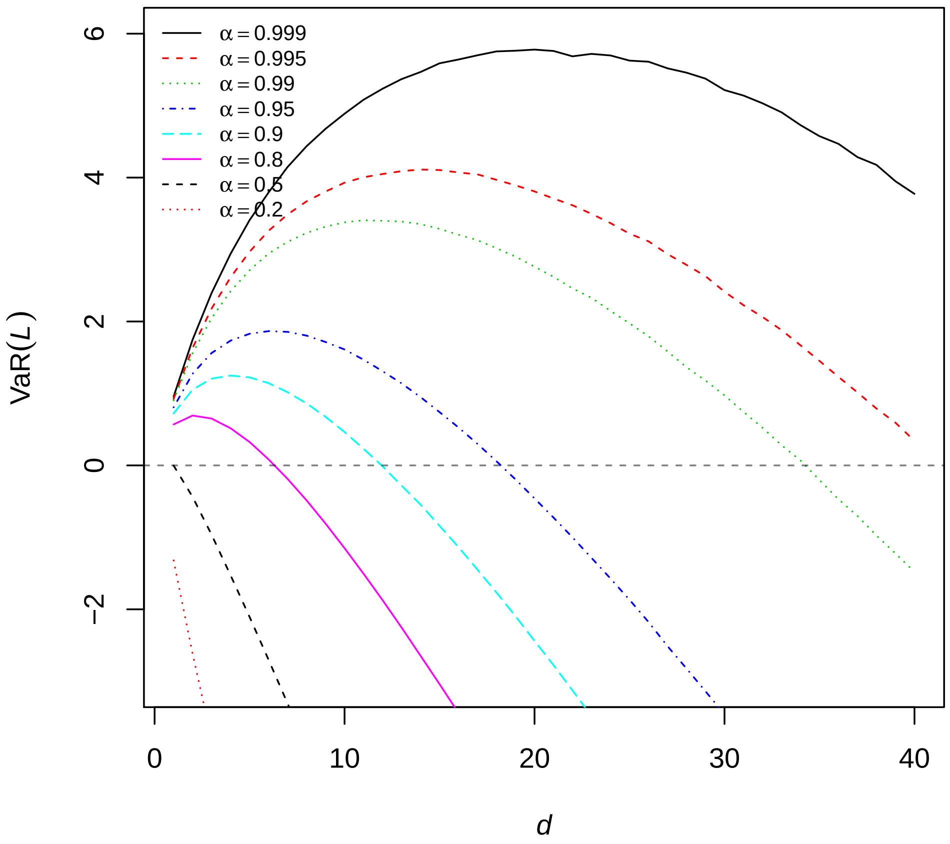

Figure 5: $\mathrm{VaR}_\alpha(L)$ estimates for $d \in \{1, 2, \dots, 40\}$ (Monte Carlo sample size 10^6) for independent $X_1, \dots, X_d$.

It follows from (4) that if all components of $\boldsymbol{X}$ are small simultaneously, then large losses L result, which leads to a large $\mathrm{VaR}_\alpha(L)$. The former condition is also known as *lower tail dependence* and is present for the Clayton, but not for the Gumbel copula.

d**:** $\mathrm{VaR}_\alpha(L)$ is increasing in d. This can be verified numerically via:

```
R> stopifnot(
+    VaR[, n = "256",   "5", , ] <= VaR[, n = "256",  "20", , ],
+    VaR[, n = "256",  "20", , ] <= VaR[, n = "256", "100", , ],
+    VaR[, n = "256", "100", , ] <= VaR[, n = "256", "500", , ])
```

This result is somewhat surprising as there is no reasonable explanation for this behavior. Indeed, the mean of L is even *decreasing* in d as

$$\mathbb{E}[L] = \sum_{j=1}^{d} (1 - \mathbb{E}[\exp(X_j)]) = d(1 - e^{1/2}) \approx -0.65d,$$

where we used $X_j \sim \mathcal{N}(0, 1)$. Furthermore, one can show numerically that also the median is decreasing in d. For the extreme case of (complete) comonotonicity—corresponding to Kendall's tau being 1, and almost surely identical X_j— we see that $\mathrm{VaR}_\alpha(L)$ is increasing in d for $\alpha > \frac{1}{2}$, since

$$L = \sum_{j=1}^{d}(1-\exp(X_j)) \overset{\mathrm{d}}{=} \sum_{j=1}^{d}(1-\exp(X)) = d(1-\exp(\Phi^{-1}(U))),$$

which implies $\mathrm{VaR}_\alpha(L) = d(1-\exp(-\Phi^{-1}(\alpha)))$; this is increasing in d for $\alpha > \frac{1}{2}$. Consequently, a sufficiently large dependence may lead to $\mathrm{VaR}_\alpha(L)$ being increasing in d. For independence, the other "extreme", F_L is not simply tractable as a convolution, however we can approximate it or rather its inverse, the quantiles $\mathrm{VaR}_\alpha(L)$ by the central limit theorem,

$$\begin{aligned} q_{L_\infty}(\alpha) &= \mathbb{E}[L] + \sigma(L)\cdot z_\alpha = \\ &= d(1-e^{1/2}) + \sqrt{d(e^2-e)}z_\alpha \approx -0.6487d + 2.161 z_\alpha\sqrt{d}, \end{aligned} \tag{5}$$

which is indeed increasing at first, before decreasing, when z_α is clearly positive, i.e., $\alpha > \frac{1}{2}$.[9] Figure 5 shows $\mathrm{VaR}_\alpha(L)$ estimates for $d \in \{1,2,\dots,40\}$ (evaluated via Monte Carlo based on $n = 10^6$ samples). Interestingly, $\mathrm{VaR}_\alpha(L)$ is increasing for small d at first, then for larger d, it is decreasing, for $\alpha > \frac{1}{2}$ (where "small" and "large" depend on α).

5. Conclusion

The R package **simsalapar** allows one to easily set up, conduct, and analyze large-scale simulations studies in R. The user of the package can proceed as follows to conduct his or her own simulation study:

Step 1: Formulate the simulation problem and determine the input variables; see Section 2.1.

Step 2: Determine the type of each input variable and create the variable list with `varlist()`; see Section 2.2.

Step 3: Implement the problem-specific function `doOne()` which computes the statistic of interest for one combination of input variables; see Section 2.4.

Step 4: Run the simulation sequentially with `doLapply()` or in parallel with one of `doForeach()`, `doRmpi()`, `doMclapply()`, or `doClusterApply()`; see Section 3 and the appendix. This can involve replicates via a variable of type "N" as our `n.sim` (N_{sim}).

Step 5: Analyze the results (values, errors, warnings, run time); see auxiliary functions and the high-level functions `toLatex()` and `mayplot()` for creating sophisticated LaTeX tables and matrix-like figures of the results presented in Section 4 and the appendix.

We explained and guided the reader through a working example end-to-end, which highlights all of the above major steps. More explanations, tests, and further examples can be found in

[9] Requiring the derivative wrt d of $q_{L_\infty}(\alpha)$ to be non-negative, is equivalent to the condition $d \le (e^2 - e)/(2(e^{1/2}-1))^2 {z_\alpha}^2 \approx 2.77{z_\alpha}^2$, showing a maximum when $\alpha > 0.8$ or so.

the package itself, notably in the demos of **simsalapar**; use `demo(package = "simsalapar")` to list them.

One of the main features of **simsalapar** is that important aspects of a simulation study such as catching of errors and warnings, measuring run time, or dealing with seeds are automatically taken care of or easily adjusted if required. This simplifies parallel computing significantly, makes it accessible to a wider audience, and largely reduces the risk of obtaining unreliable results due to a wrong implementation. Furthermore, the `do*()` functions provide very similar "front-ends" to quite different approaches for parallel computing. This is a major advantage if one has to repeat a simulation with a different operating system or hardware architecture, or if an extension to the simulation study suddenly requires it to be run on a computer cluster rather than multiple cores locally. Moreover, **simsalapar** provides useful tools to analyze and present the results. Given how challenging the development of these tools is, there is more to be expected in **simsalapar** in the future, notably resulting from feedback by package users interested in even more approaches for parallelization.

Acknowledgements

We would like to thank Matthias Kirchner (ETH Zurich) and Dr. Robin Nittka (Google Zurich) for proofreading the manuscript and our past and current master and Ph.D. students for motivating this work.

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. `doi:10.18637/jss.v037.i03`.

Chan TJ (2014). ***ezsim**: Provide an Easy to Use Framework to Conduct Simulation.* R package version 0.5.5, URL `https://CRAN.R-project.org/package=ezsim`.

Dahl DB (2016). ***xtable**: Export Tables to LaTeX or HTML.* R package version 1.8-2, URL `https://CRAN.R-project.org/package=xtable`.

Embrechts P, Hofert M (2013). "A Note on Generalized Inverses." *Mathematical Methods of Operations Research*, **77**(3), 423–432. `doi:10.1007/s00186-013-0436-7`.

Eugster MJA, Knaus J, Porzelius C, Schmidberger M, Vicedo E (2011). "Hands-on Tutorial for Parallel Computing with R." *Computational Statistics*, **26**, 219–239. `doi:10.1007/s00180-010-0206-4`.

Gneiting T (2011). "Making and Evaluating Point Forecasts." *Journal of the Americal Statistical Association*, **106**, 746–762. `doi:10.1198/jasa.2011.r10138`.

Gorjanc G (2012). ***simSummary**: Simulation Summary.* R package version 0.1.0, URL `https://CRAN.R-project.org/package=simSummary`.

Kane MJ, Emerson J, Weston S (2013). "Scalable Strategies for Computing with Massive Data." *Journal of Statistical Software*, **55**(14), 1–19. `doi:10.18637/jss.v055.i14`.

Mächler M, others (2016). ***sfsmisc**: Utilities from Seminar für Statistik ETH Zurich.* R package version 1.0-29, URL `https://CRAN.R-project.org/package=sfsmisc`.

McNeil AJ, Frey R, Embrechts P (2005). *Quantitative Risk Management: Concepts, Techniques, Tools.* Princeton University Press.

Murdoch D (2014). ***tables**: Formula-Driven Table Generation.* R package version 0.7.79, URL `https://CRAN.R-project.org/package=tables`.

Murrell P (2006). *R Graphics.* Chapman & Hall/CRC. `doi:10.1201/9781420035025`.

Murrell P (2014). ***gridBase**: Integration of Base and **grid** Graphics.* R package version 0.4-7, URL `https://CRAN.R-project.org/package=gridBase`.

Nelsen RB (2006). *An Introduction to Copulas.* Springer-Verlag. `doi:10.1007/0-387-28678-0`.

R Core Team (2015). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL `http://www.R-project.org/`.

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

Revolution Analytics (2013). ***doSNOW**: Foreach Parallel Adaptor for the Snow Package.* R package version 1.0.9, URL `https://CRAN.R-project.org/package=doSNOW`.

Revolution Analytics, Weston S (2015a). ***doParallel**: Foreach Parallel Adaptor for the Parallel Package.* R package version 1.0.14, URL `https://CRAN.R-project.org/package=doParallel`.

Revolution Analytics, Weston S (2015b). ***foreach**: Foreach Looping Construct for R.* R package version 1.4.3, URL `https://CRAN.R-project.org/package=foreach`.

Sarkar D (2008). ***lattice**: Multivariate Data Visualization with R.* Springer-Verlag, New York.

Sklar A (1959). "Fonctions De Répartition à n Dimensions et Leurs Marges." *Publications de L'Institut de Statistique de L'Université de Paris*, **8**, 229–231.

Venables WN, Smith DM, R Core Team (2015). *An Introduction to R.* URL `https://CRAN.R-project.org/`.

Weston S (2015). *Nesting `foreach` Loops.* R package version 1.4.3, `nested` vignette, URL `https://CRAN.R-project.org/package=foreach`.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis.* Springer-Verlag.

Yu H (2002). ***Rmpi**: Parallel Statistical Computing in R.* URL `https://CRAN.R-project.org/doc/Rnews/Rnews_2002-2.pdf`.

A. Behind the scenes: Advanced features of simsalapar

A.1. Select functions for conducting the simulation

The function `doCallWE()`

The R package **simsalapar** provides the following auxiliary function `doCallWE()` for computing the components `value`, `error`, `warning`, and `time` as addressed in Section 2.3. It is called from `subjob()` and based on `tryCatch.W.E()` which is part of R's `demo(error.catching)` for catching both warnings and errors.

```
doCallWE <- function(f, argl, timer = mkTimer(gcFirst = FALSE)) {
  tim <- timer(res <- tryCatch.W.E(do.call(f, argl))) # compute f(<argl>)
  is.err <- is(val <- res$value, "simpleError") # logical indicating an error
  list(value = if(is.err) NULL else val, # value (or NULL in case of error)
    error = if(is.err) val else NULL, # error (or NULL if okay)
    warning = res$warning, # warning (or NULL)
    time = tim) # time
}
```

The function `subjob()`

`subjob()` calls `doOne()` via `doCallWE()` for computing a sub-job, that is, a row of the virtual grid. It is called by the `do*()` functions. Besides catching errors and warnings, and measuring run time via calling `doCallWE()`, the main duty of `subjob()` is to correctly deal with the seed. It also provides a monitor feature.

```
subjob <- function(i, pGrid, nonGrids, n.sim, seed, keepSeed = FALSE,
  repFirst = TRUE, doOne,
  timer = mkTimer(gcFirst = FALSE), monitor = FALSE, ...)
{
  ## i |-> (i.sim, j) :
  ## determine corresponding i.sim and row j in the physical grid
  if(repFirst) {
    i.sim <- 1 + (i-1) %%  n.sim ## == i  when n.sim == 1
    j          <- 1 + (i-1) %/% n.sim ## row of pGrid
    ## Note: this case first iterates over i.sim, then over j:
    ## (i.sim,j) = (1,1), (2,1), (3,1),..., (1,2), (2,2), (3,2), ...
  } else {
    ngr <- nrow(pGrid) # number of rows of the (physical) grid
    j          <- 1 + (i-1) %%  ngr ## row of pGrid
    i.sim <- 1 + (i-1) %/% ngr
    ## Note: this case first iterates over j, then over i.sim:
    ## (i.sim,j) = (1,1), (1,2), (1,2),..., (2,1), (2,2), (2,3), ...
  }

  ## seeding
  if(is.null(seed)) {
    if(!exists(".Random.seed")) runif(1) # guarantees .Random.seed exists
    ## => this is typically not reproducible
  } else if(is.numeric(seed)) {
    if(length(seed) != n.sim) stop("'seed' has to be of length ", n.sim)
    set.seed(seed[i.sim]) # same seed for all runs within the same i.sim
    ## => calculations based on same random numbers as much as possible
  }
  ## else if(length(seed) == n.sim * ngr && is.numeric(seed)) {
  ##   set.seed(seed[i]) # different seed for *every* row of the virtual grid
  ##   always (?) suboptimal (more variance than necessary)
  ## }
  else if(is.list(seed)) { # (currently) L'Ecuyer-CMRG
    if(length(seed) != n.sim) stop("'seed' has to be of length ", n.sim)
    if(!exists(".Random.seed"))
        stop(".Random.seed does not exist - in l'Ecuyer setting")
    assign(".Random.seed", seed[[i.sim]], envir = globalenv())
  } else if(is.na(seed)) {
    keepSeed <- FALSE
  } else {
    if(!is.character(seed)) stop(.invalid.seed.msg)
    switch(match.arg(seed, choices = c("seq")),
      "seq" = { # sequential seed :
        set.seed(i.sim) #same seed for all runs within the same i.sim
        ## => calculations based on the same random numbers
      },
        stop("invalid character 'seed': ", seed))
  }
  ## save seed, compute and return result for one row of the virtual grid
  if(keepSeed) rs <- .Random.seed # save here in case it is advanced in doOne

  ## monitor checks happen already in caller!
  if(isTRUE(monitor)) monitor <- printInfo[["default"]]

  ## doOne()'s arguments, grids, non-grids, and '...':
  args <- c(pGrid[j, , drop=FALSE],
    ## [nonGrids is never missing when called from doLapply() etc.]
    if(missing(nonGrids) || length(nonGrids) == 0)
    list(...) else c(nonGrids, ...))
  nmOne <- names(formals(doOne))
  if(!identical(nmOne, "..."))
    args <- args[match(names(args), nmOne)] # adjust order for doOne()

  r4 <- doCallWE(doOne, args, timer = timer)

  ## monitor (after computation)
  if(is.function(monitor)) monitor(i.sim, j = j, pGrid = pGrid,
    n.sim = n.sim, res4 = r4)

  c(r4, if(keepSeed) list(.Random.seed = rs)) # 5th component .Random.seed
}
```

For the different seeding methods implemented, see `?subjob`. If `keepSeed=TRUE` and `seed` is not `NA`, `subjob()` saves `.Random.seed` as the fifth component of the output vector (besides the four components returned by `doCallWE()`). This is useful for reproducing the result of the corresponding call of `doOne()` for debugging purposes, for example.

The default seeding method in the `do*()` functions is `"seq"`. This is a comparably simple default which guarantees reproducibility. Note, however, that for very large simulations, there is no guarantee that the random-number streams are sufficiently "apart". For this, we recommend l'Ecuyer's random number generator `L'Ecuyer-CMRG`; see Section A.3 for an example.

The function `doLapply()`

As mentioned before, `doLapply()` is essentially a wrapper for `lapply()` to iterate (sequentially) over all rows in the virtual grid, that is, over all sub-jobs. As an important ingredient, `saveSim()`, explained below, is used to deal with the raw result list.

```
doLapply <- function(vList, seed = "seq", repFirst = TRUE, sfile = NULL,
  check = TRUE, doAL = TRUE, subjob. = subjob, monitor = FALSE, doOne, ...)
{
  if (!is.null(r <- maybeRead(sfile))) return(r)
  stopifnot(is.function(subjob.), is.function(doOne))
  if (!(is.null(seed) || is.na(seed) || is.numeric(seed) ||
    (is.list(seed) && all(vapply(seed, is.numeric, NA))) ||
    is.character(seed)))
    stop(.invalid.seed.msg)
  if (check) doCheck(doOne, vList, nChks = 1, verbose = FALSE)

  ## monitor checks {here, not in subjob()!}
  if (!(is.logical(monitor) || is.function(monitor)))
    stop(gettextf("'monitor' must be logical or a function like %s",
      "printInfo[[\"default\"]]"))

  ## variables
  pGrid <- mkGrid(vList)
  ngr <- nrow(pGrid)
  ng <- get.nonGrids(vList) # => n.sim >= 1
  n.sim <- ng$n.sim # get n.sim

  ## actual work
  res <- lapply(seq_len(ngr * n.sim), subjob., pGrid = pGrid,
    nonGrids = ng$nonGrids, repFirst = repFirst, n.sim = n.sim,
    seed = seed, doOne = doOne, monitor = monitor, ...)

  ## convert result and save
  saveSim(res, vList = vList, repFirst = repFirst, sfile = sfile,
    check = check, doAL = doAL)
}
```

The functions `saveSim()` *and* `maybeRead()`

After having conducted the main simulation with one of the `do*()` functions, we would like to create and store the result array. It can then be loaded and worked on for the analysis of the study which is often done on a different computer. For creating, checking, and saving the array, **simsalapar** provides the function `saveSim()`.

If possible, `saveSim()` creates an array of lists (via `mkAL()`), where each element of the array is a list of length four or five as returned by `subjob()`. If this fails, `saveSim()` simply takes its input list. It then stores this array (or list) in the given `.rds` file (via `saveRDS()`) and returns it for further usage. In our working example, the array itself is five-dimensional, the dimensions corresponding to n, d, C, τ, and N_{sim}.

```
saveSim <- function(x, vList, repFirst, sfile, check = TRUE, doAL = TRUE) {
  if (doAL) {
    a <- tryCatch(mkAL(x, vList, repFirst = repFirst, check = check),
      error = function(e) e)
    if (inherits(a, "error")) {
      warning(paste("Relax..: The simulation result 'x' is being saved;",
        "we had an error in 'mkAL(x, *)' ==> returning 'x' (argument, a list).",
        "  you can investigate mkAL(x, ..) yourself. The mkAL() err.message:",
        conditionMessage(a), sep = "\n"))
      a <- x
    }
  } else a <- x
  if (!is.null(sfile)) saveRDS(a, file = sfile)
  a
}
```

For creating the array, `saveSim()` calls `mkAL()` which is implemented as follows:

```
mkAL <- function(x, vList, repFirst, check = TRUE) {
  grVars <- getEl(vList, "grid", NA)
  n.sim <- get.n.sim(vList)
  ngr <- prod(vapply(lapply(grVars, `[[`, "value"), length, 1L)) # nrow(pGrid)
  lx <- n.sim * ngr
  if (check) {
    stopifnot(is.list(x))
    if (length(x) != lx)
      stop("varlist-defined grid variable dimensions do not match length(x)")
    if (length(x) >= 1) {
      x1 <- x[[1]]
      stopifnot(is.list(x1),
        c("value", "error", "warning", "time") %in% names(x1))
    }
  }
  if (repFirst) ## reorder x
    x <- x[as.vector(matrix(seq_len(lx), ngr, n.sim, byrow = TRUE))]
  iVals <- getEl(vList, "inner")
  xval <- lapply(x, `[[`, "value")
  iLen <- vapply(iVals, length, 1L)
  n.inVals <- prod(iLen)
  if (check) {
    ## vector of all 'value' lengths
    v.len <- vapply(xval, length, 1L)
    ## NB: will be of length zero, when an error occured !!

    ##' is N a true multiple of D? includes equality, but we also true vector
    is.T.mult <- function(N, D) N >= D & {
      q <- N/D
      q == as.integer(q)
    }

    if (!all(eq <- is.T.mult(v.len, n.inVals))) {
```

```
    ## (!all(len.divides <- v.len %% n.inVals == 0)) {
    not.err <- vapply(lapply(x, `[[`, "error"), is.null, NA)
    if (!identical(eq, not.err)) {
      msg <- gettextf(
        "some \"value\" lengths differ from 'n.inVals'=%d without error",
        n.inVals)
      if (interactive()) {
        ## warning() instead of stop():
        ## had *lots* of computing till here --> want to
        ## investigate
        warning(msg, domain = NA, immediate. = TRUE)
        cat("You can investigate (v.len, xval, etc) now:\n")
        browser()
      } else stop(msg, domain = NA)
    }
    if (all(v.len == 0))
      warning(gettextf(
        "All \"%s\"s are of length zero. The first error message is\n %s",
        "value", dQuote(conditionMessage(x[[1]][["error"]]))), domain = NA)
  }
}

if (length(iVals) > 0 && length(xval) > 0) {
  ## ensure that inner variable names are 'attached' to x's 'value's :
  if (noArr <- is.null(di <- dim(xval[[1]])))
    di <- length(xval[[1]])
  rnk <- length(di)  # true dim() induced rank
  nI <- length(iLen)  # = number of inner Vars;  iLen are their lengths
  for (i in seq_along(xval)) {
    n. <- length(xi <- xval[[i]])
    if (n. == 0) # 'if (check)' above has already ensured this is an 'error'
      xi <- NA_real_
    ## else if (n. != n.inVals) warning(gettext('x[[%d]] is of wrong
    ## length (=%d) instead of %d', i, n., n.inVals), domain = NA)
    dn.i <- if (noArr) {
      if (nI == 1)
        list(names(xi)) else rep.int(list(NULL), nI)
    } else if (is.null(dd <- dimnames(xi)))
      rep.int(list(NULL), rnk) else dd
    ## ==> rnk := length(di) == length(dn.i)
    if (rnk == nI) { # = length(iVals) = length(iLen) -- simple matching case
      names(dn.i) <- names(iLen)
    } else { # more complicated as doOne() returned a full vector, matrix ...
      if (rnk != length(dn.i)) warning(
        "dim() rank, i.e., length(dim(.)), does not match dimnames() rank")
      if (nI > rnk) { # or rather error?
        warning("nI=length(iVals) larger than length(<dimnames>)")
      } else {
        # nI<rnk==length(di)==length(dn.i) =>
        # find matching dim() assume inner variables
        # match the *end* of the array
        j <- seq_len(rnk - nI)
        j <- which(di[nI + j] == iLen[j])
```

```
          if (is.null(names(dn.i)))
          names(dn.i) <- rep.int("", rnk)
          names(dn.i)[nI + j] <- names(iLen)[j]
        }
      }
      x[[i]][["value"]] <- array(xi, dim = if (noArr)
        iLen else di, dimnames = dn.i)
    }
  }

  gridNms <- mkNms(grVars, addNms = TRUE)
  dmn <- lapply(gridNms, sub, pattern = ".*= *", replacement = "")
  dm <- vapply(dmn, length, 1L)
  if (n.sim > 1) {
    dm <- c(dm, n.sim = n.sim)
    dmn <- c(dmn, list(n.sim = NULL))
  }
  ## build array
  array(x, dim = dm, dimnames = dmn)
}
```

For reading a saved object of a simulation study, **simsalapar** provides the function `maybeRead()`. If the provided `.rds` file exists, `maybeRead()` reads and returns the object. Otherwise, `maybeRead()` does nothing (hence the name). This is useful for reading and analyzing the result object at a later stage by executing the same R script containing both the simulation and its analysis[10].

```
maybeRead <- function(sfile, msg = TRUE) {
  if (is.character(sfile) && file.exists(sfile)) {
    if (msg)
      message("getting object from ", sfile)
    structure(readRDS(sfile), fromFile = TRUE)
  }
}
```

A.2. Select functions for the analysis

The function `getArray()`

As promised in Section 4, we now present the implementation of the function `getArray()`. This function receives the result array of lists, picks out a specific component of the lists, and returns an array containing these components. This is especially useful when analyzing the results of a simulation.

```
getArray <- function(x, comp = c("value", "error", "warning", "time"),
  FUN = NULL, err.value = NA)
{
  comp <- match.arg(comp)
  if (comp == "value") return(valArray(x, err.value = err.value, FUN = FUN))
  ## else :
  dmn <- dimnames(x)
```

[10]Note that the first part of this paper is itself such an example.

```
  dm <- dim(x)
  if (is.null(FUN)) {
    FUN <- switch(comp, error = ,
      warning = function(x) !vapply(x, is.null, NA), time = ul)
  } else stopifnot(is.function(FUN))
  array(FUN(lapply(x, `[[`, comp)), dim = dm, dimnames = dmn)
}
```

The method toLatex.ftable *and related functions*

The ftable method toLatex.ftable for creating LaTeX tables calls several auxiliary functions, detailed below.

First, the function ftable2latex() is called. It takes the provided flat contingency table, converts R expressions in the column and row variables to LaTeX expressions, and, unless they are LaTeX math expressions, escapes them (per default with the function escapeLatex()). Furthermore, ftable2latex() takes the table entries and converts R expressions (and only those) to LaTeX expressions (which are escaped in case x.escape=TRUE; this is not the default).

```
ftable2latex <- function(x, vList = NULL, x.escape,
  exprFUN = expr2latex, escapeFUN = escapeLatex)
{
  ## checks
  stopifnot(is.function(exprFUN), is.function(escapeFUN))
  cl <- class(x)
  dn <- c(r.v <- attr(x, "row.vars"), c.v <- attr(x, "col.vars"))
  if (is.null(vList)) {
    nvl <- names(vList <- dimnames2varlist(dn))
  } else {
    stopifnot(names(dn) %in% (nvl <- names(vList)))
  }
  vl <- .vl.as.list(vList)
  ## apply escapeORmath() to expressions of column and row variables
  names(c.v) <- lapply(lapply(vl[match(names(c.v), nvl)], `[[`, "expr"),
    escapeORmath, exprFUN = exprFUN, escapeFUN = escapeFUN)
  names(r.v) <- lapply(lapply(vl[match(names(r.v), nvl)], `[[`, "expr"),
    escapeORmath, exprFUN = exprFUN, escapeFUN = escapeFUN)
  ## for the entries of 'x' itself, we cannot apply exprFUN(.) everywhere,
  ## only ``where expr''
  exprORchar <- function(u) {
    lang <- vapply(u, is.language, NA)  # TRUE if 'name', 'call' or 'expression'
    u[lang] <- exprFUN(u[lang])  # apply (per default) expr2latex()
    u[!lang] <- as.character(u[!lang])  # or format()?
    u
  }
  x <- exprORchar(x)  # converts expressions (and only those) to LaTeX
  if (x.escape) x <- escapeFUN(x)  # escapes LaTeX expressions
  ## now the transformed row and col names
  attr(x, "row.vars") <- lapply(r.v, escapeFUN)
  attr(x, "col.vars") <- lapply(c.v, escapeFUN)
  class(x) <- cl
  x
}
```

The second function called, `fftable()`, formats the resulting flat contingency table (applying a new version of `format.ftable()` which is available in base R from 3.0.0) and returns a flat contingency table with two attributes `ncv`, `nrv` indicating the number of column variables and the number of row variables, respectively.

Next, `tablines()` is called. It receives a character matrix with attributes `ncv`, `nrv` (typically) obtained from `fftable()`. It then creates and returns a list with the components `body`, `body.raw`, `head`, `head.raw`, `align`, and `rsepcol`. By default, `body` is a vector of character strings containing the full rows (including row descriptions, if available) of the body of the table, table entries (separated by the column separator `csep`), and the row separator as specified by `rsep`. `body.raw` provides the row descriptions (if available) and the table entries as a character matrix. Similar for `head.raw` which is a character matrix containing the entries of the table header (the number of rows of this matrix is essentially determined by `ncv`); typically, this is the header of the flat contingency table created by `fftable()`. `head` contains a "collapsed" version of `head.raw` but in a much more sophisticated way. `\multicolumn` statements for centering of column headings and title rules for separating groups of columns are introduced (`\cmidrule` if `booktabs = TRUE`; otherwise `\cline`). The list component `align` is a string which contains the alignment of the table entries (as accepted by LaTeX's `tabular` environment). The default implies that all columns containing row names are left-aligned and all other columns are right-aligned. The component `rsepcol` is a vector of characters which contain the row separators `rsep` or, additionally, `\addlinespace` commands for separating blocks of rows belonging to the same row variables or groups of such. The default chooses a larger space between groups of variables which appear in a smaller column number. In other words, the "largest" group is determined by the variables which appear in the first column, the second-largest by those in the second column etc. up to the second-last column containing row variables. For more details we refer to the source code of `tablines()` in **simsalapar**.

Finally, the method `toLatex.ftable` calls `wrapLaTable()`. This function wraps a LaTeX `table` and `tabular` environment around, which can be put in a LaTeX document.

```
toLatex.ftable <- function(object, vList = NULL, x.escape = FALSE,
  exprFUN = expr2latex, escapeFUN = escapeLatex, align = NULL, booktabs = TRUE,
  head = NULL, rsep = "\\\\", sp = if (booktabs) 3 else 1.25, rsep.sp = NULL,
  csep = " & ", quote = FALSE, lsep = " \\textbar\\ ", do.table = TRUE,
  placement = "htbp", center = TRUE, fontsize = "normalsize", caption = NULL,
  label = NULL, ...)
{
  ## convert expressions, leave rest:
  ft <- ftable2latex(object, vList, x.escape = x.escape, exprFUN = exprFUN,
    escapeFUN = escapeFUN)
  ## ftable -> character matrix (formatted ftable) with attributes 'ncv' and
  ## 'nrv'
  ft <- fftable(ft, quote = quote, lsep = lsep, ...)
  ## character matrix -> latex {head + body}:
  tlist <- tablines(ft, align = align, booktabs = booktabs, head = head,
    rsep = rsep, sp = sp, rsep.sp = rsep.sp, csep = csep)
  ## wrap table and return 'Latex' object:
  wrapLaTable(structure(tlist$body, head = tlist$head), do.table = do.table,
    align = tlist$align, placement = placement, center = center,
    booktabs = booktabs, fontsize = fontsize, caption = caption, label = label)
}
```

Function `mayplot()` *to visualize a 5D array*

We will now present a bit more details about the function `mayplot()` for creating matrix-like plots of arrays up to dimension five. Due to space limitations, we only describe `mayplot()` verbally here and refer to the source code of **simsalapar** for the exact implementation.

`mayplot()` utilizes the function `grid.layout()` to determine the matrix-like layout, including spaces for labels; call `mayplot()` with `show.layout=TRUE` to see how the layout looks like. `pushViewport()` is then used to put the focus on a particular cell of the plot matrix (or several cells simultaneously, see the global y axis label, for example). The focus is released via `popViewport()`. Within a particular cell of the plot matrix a panel function is chosen for plotting. This is achieved by **gridBase**. The default panel function is either `boxplot.matrix()` or `lines()` depending on whether `n.sim` exists. We also display a background with grid lines similar to the style of **ggplot2**. Axes (for the y axis in logarithmic scale using `eaxis` from **sfsmisc**; see Mächler and others (2016)) are then printed depending on which cell the focus is on; similar for the row and column labels of the cells, again in **ggplot2**-style. Due to the flexibility of **grid**, we can also create a legend in the same way as in the plot. Finally, we save initial graphical parameters with `opar <- par(no.readonly=TRUE)` and restore them on function exit in order to not change graphical parameters for possible subsequent plots.

Overall, `mayplot()` is quite flexible in visualizing results contained in arrays of dimensions up to five, see the corresponding help file for more customizations.

A.3. Alternative varlists and simulations

In addition to the basic example in Section 2.6, we now call `doLapply()` under various other setups, seeding methods, etc., including the case of *no* replications, that is, `n.sim = 1`:

```
R> res0. <- doLapply(varList, seed = NULL, sfile = "res0_lapply_NULL.rds",
+     doOne = doOne) # seed = NULL (typically not reproducible)
R> raw0 <- doLapply(varList, sfile = "raw0_lapply_NULL.rds",
+     doAL = FALSE, # do not call mkAL() --> keep "raw" result
+     doOne = doOne, names = TRUE) # seed = "seq" (default)
R> varList.1 <- set.n.sim(varList, 1) # n.sim = 1
R> res01 <- doLapply(varList.1, sfile = "res01_lapply_seq.rds",
+     doOne = doOne, names = TRUE)
R> varList.2 <- set.n.sim(varList, 2) # n.sim = 2
R> LE.seed <- c(2, 11, 15, 27, 21, 26)
R> old.seed <- .Random.seed
R> set.seed(LE.seed, kind = "L'Ecuyer-CMRG")
R> (n.sim <- get.n.sim(varList.2))

[1] 2

R> seedList <- LEseeds(n.sim)
R> system.time(
+  res02 <- doLapply(varList.2, seed = seedList,
+     sfile = "res02_lapply_LEc.rds", doOne = doOne, names = TRUE,
+     monitor = interactive()))
```

```
   user  system elapsed 
  0.002   0.000   0.002 
```

```
R> RNGkind()
```

```
[1] "L'Ecuyer-CMRG" "Inversion"
```

```
R> old.seed -> .Random.seed
R> RNGkind()
```

```
[1] "Mersenne-Twister" "Inversion"
```

A.4. A comment on load-balancing

In terms of load-balancing, one advantage of our approach is that each repeated simulation has the same expected run time. Note, however, that thousands of fast sub-jobs might lead to a comparably large overall run time due to both the waiting times for the jobs to start on a cluster and due to the overhead in communication between the manager and the workers. It might therefore be more efficient to send blocks of sub-jobs to the same core or node. This feature is provided by the argument `block.size` in the `do*()` functions `doForeach()`, `doRmpi()`, `doMclapply()`, or `doClusterApply()`.

A.5. Using `foreach`

The wrapper `doForeach()` is based on the function `foreach()` of the package **foreach**. It allows to carry out parallel computations on multiple nodes or cores. In principle, different parallel backends can be used to conduct parallel computations with **foreach**'s `foreach()`; see Weston (2015) or the various parallel adaptors (e.g., **doMC**, **doMPI**, **doParallel**, **doRedis**, **doSNOW**). For example, SNOW cluster types could be specified with `registerDoSNOW()` from the package **doSNOW**; see Revolution Analytics (2013). We use the package **doParallel** here which provides an interface between **foreach** and the R package **parallel**; see Revolution Analytics and Weston (2015a). The number of nodes can be specified via `cluster.spec` (defaulting to 1) and the number of cores via `cores.spec` (defaulting to **parallel**'s `detectCores()`). For more details, we refer to the package source code and the vignettes of **foreach** and **doParallel**.

```
doForeach <- function(vList, cluster = makeCluster(detectCores(), type = "PSOCK"),
  cores = NULL, block.size = 1, seed = "seq", repFirst = TRUE, sfile = NULL,
  check = TRUE, doAL = TRUE, subjob. = subjob, monitor = FALSE, doOne,
  extraPkgs = character(), exports = character(), ...)
{
  ## Unfortunately, imports() ends not finding 'iter' from pkg 'iterators': -->
  ## rather strictly require things here:
  getPkg <- function(pkg) if (!require(pkg, character.only = TRUE))
    stop(sprintf(
      "You must install the CRAN package '%s' before you can use doForeach()",
      pkg), call. = FALSE)
  getPkg("foreach")
  getPkg("doParallel")
  if ((is.null(cluster) && is.null(cores)) || (!is.null(cluster) &&
    !is.null(cores))) stop("Precisely one of 'cluster' or 'cores' has to be not NULL")
  if (!is.null(r <- maybeRead(sfile))) return(r)
  stopifnot(is.function(subjob.), is.function(doOne))
  if (!(is.null(seed) || is.na(seed) || is.numeric(seed) || (is.list(seed) &&
    all(vapply(seed, is.numeric, NA))) || is.character(seed)))
    stop(.invalid.seed.msg)
  if (check) doCheck(doOne, vList, nChks = 1, verbose = FALSE)
  if (!is.null(cluster))
    on.exit(stopCluster(cluster))  # shut down cluster and execution environment

  ## monitor checks {here, not in subjob()!}
  if (!(is.logical(monitor) || is.function(monitor))) stop(gettextf(
    "'monitor' must be logical or a function like %s", "printInfo[[\"default\"]]"))

  ## variables
  pGrid <- mkGrid(vList)
  ngr <- nrow(pGrid)
  ng <- get.nonGrids(vList)  # => n.sim >= 1
  n.sim <- ng$n.sim
  stopifnot(1 <= block.size, block.size <= n.sim)
  if (n.sim%%block.size != 0)
    stop("block.size has to divide n.sim")

  ## Two main cases for parallel computing multiple cores ?registerDoParallel ->
  ## Details -> Unix + multiple cores => 'fork' is used
  if (!is.null(cores)) {
    stopifnot(is.numeric(cores), length(cores) == 1)
    registerDoParallel(cores = cores) # register doParallel to be used with foreach
  } else registerDoParallel(cluster) # multiple nodes;
  # register doParallel to be used with foreach
  if (check) cat(sprintf("getDoParWorkers(): %d\n", getDoParWorkers()))

  ## actual work
  n.block <- n.sim%/%block.size
  i <- NULL  ## <- required for R CMD check ...
  res <- ul(foreach(i = seq_len(ngr * n.block), .packages = c("simsalapar",
    extraPkgs), .export = c(".Random.seed", "iter", "mkTimer", exports)) %dopar%
    {
      lapply(seq_len(block.size), function(k) subjob.((i - 1) * block.size +
        k, pGrid = pGrid, nonGrids = ng$nonGrids, repFirst = repFirst,
        n.sim = n.sim, seed = seed, doOne = doOne, monitor = monitor,
        ...))
    })
  ## convert result and save
  saveSim(res, vList, repFirst = repFirst, sfile = sfile, check = check, doAL = doAL)
}
```

Let us call `doForeach()` for our working example, with `seed = NULL`, and `n.sim = 1`, respectively.

```
R> res1 <- doForeach(varList, sfile = "res1_foreach_seq.rds",
+    doOne = doOne, names = TRUE)
```

```
R> system.time(
+  res1. <- doForeach(varList, seed = NULL, sfile = "res1_foreach_NULL.rds",
+    doOne = doOne))

   user  system elapsed
  0.005   0.066   1.714

R> res11 <- doForeach(varList.1, sfile = "res11_foreach_seq.rds",
+    doOne = doOne, names = TRUE)
```

Next, we demonstrate how l'Ecuyer's random number generator can be used.

```
R> old.seed <- .Random.seed
R> set.seed(LE.seed, kind = "L'Ecuyer-CMRG")
R> n.sim <- get.n.sim(varList.2)
R> seedList <- LEseeds(n.sim)
R> system.time(res12 <- doForeach(varList.2, seed = seedList,
+    sfile = "res12_lapply_LEc.rds", doOne = doOne, names = TRUE,
+    monitor = interactive()))

   user  system elapsed
  0.000   0.062   1.892

R> old.seed -> .Random.seed
```

To see that `doForeach()` and `doLapply()` lead the same result, let us check for equality of `res1` with `res`. We also check equality of `res12` with `res02` which shows the same for l'Ecuyer's random number generator.

```
R> stopifnot(doRes.equal(res1 , res), doRes.equal(res12, res02))
```

A.6. Using `foreach` with nested loops

The approach we present next is similar to `doForeach()`. However, it uses nested `foreach()` loops to iterate over the grid variables and replications; see the vignettes of **foreach** for the technical details. Since this is context-specific, `doNestForeach()` is not part of **simsalapar**. Unfortunately, it is not possible to execute statements between different `foreach()` calls. This would be interesting for efficiently computing those quantities which remain fixed in subsequent `foreach()` loops only once. Note that this is also not possible for the other methods for parallel computing and thus not a limitation of this method alone.

```
doNestForeach <- function(vList, cluster = makeCluster(detectCores(),
  type = "PSOCK"), cores = NULL, block.size = 1, seed = "seq", repFirst = TRUE,
  sfile = NULL, check = TRUE, doAL = TRUE, subjob. = subjob, monitor = FALSE,
  doOne, extraPkgs = character(), exports = character(), ...)
{
  stopifnot(require("doSNOW"), require("foreach"), require("doParallel"))
  if((is.null(cluster) && is.null(cores)) || (!is.null(cluster) && !is.null(cores)))
```

```
    stop("Precisely one of 'cluster' or 'cores' has to be not NULL")
  if (!is.null(r <- maybeRead(sfile)))
    return(r)
  stopifnot(is.function(doOne))
  if (!(is.null(seed) || is.na(seed) || is.numeric(seed) || (is.list(seed) &&
    all(vapply(seed, is.numeric, NA))) || is.character(seed)))
    stop(.invalid.seed.msg)
  if (check) doCheck(doOne, vList, nChks = 1, verbose = FALSE)
  if (!is.null(cluster)) on.exit(stopCluster(cluster))

  ## monitor checks {here, not in subjob()!}
  if (!(is.logical(monitor) || is.function(monitor)))
    stop(gettextf("'monitor' must be logical or a function like %s",
      "printInfo[[\"default\"]]"))

  ## variables
  pGrid <- mkGrid(vList)
  ngr <- nrow(pGrid)
  ng <- get.nonGrids(vList)  # => n.sim >= 1
  n.sim <- ng$n.sim
  stopifnot(1 <= block.size, block.size <= n.sim)
  if (n.sim%%block.size != 0)
    stop("block.size has to divide n.sim")

  ## Two main cases for parallel computing multiple cores ?registerDoParallel ->
  ## Details -> Unix + multiple cores => 'fork' is used
  if (!is.null(cores)) {
    stopifnot(is.numeric(cores), length(cores) == 1)
    registerDoParallel(cores = cores)  # register doParallel to be used with foreach
  } else registerDoParallel(cluster)  # multiple nodes; register doParallel
  ## For using Rmpi, call with: cluster=makeCluster(max(2,
  ## Rmpi::mpi.universe.size()), type=type) would have to be shut down with:
  ## on.exit({ stopCluster(cluster) if(!interactive()) Rmpi::mpi.exit() })
  if (check)
    cat(sprintf("getDoParWorkers(): %d\n", getDoParWorkers()))

  ## need all problem-specific variables here 'grid' variables
  grVals <- getEl(vList, type = "grid")
  nn <- length(n <- grVals$n)
  nd <- length(d <- grVals$d)
  nfamily <- length(family <- grVals$family)
  ntau <- length(tau <- grVals$tau)

  ## 'inner' variables
  inVals <- getEl(vList, type = "inner")
  alpha <- inVals$alpha

  ## actual work (note, we use a different construction here)
  n.block <- n.sim/block.size
  xpObj <- c(".Random.seed", "iter", "mkTimer", exports)
  xpPkgs <- c("simsalapar", extraPkgs)
  res <- ul(foreach(j = seq_along(tau), .packages = xpPkgs, .export = xpObj) %:%
    foreach(k = seq_along(family), .packages = xpPkgs, .export = xpObj) %:%
    foreach(l = seq_along(d), .packages = xpPkgs, .export = xpObj) %:%
    foreach(m = seq_along(n), .packages = xpPkgs, .export = xpObj) %:%
    foreach(i. = seq_len(n.block), .packages = xpPkgs, .export = xpObj) %dopar%
    {
      i <- i. + n.block * ((m - 1) + nn * ((l - 1) + nd * ((k - 1) + nfamily *
        (j - 1))))
      lapply(seq_len(block.size), function(k.) subjob((i - 1) * block.size + k.,
        pGrid = pGrid, nonGrids = ng$nonGrids, repFirst = repFirst, n.sim = n.sim,
        seed = seed, doOne = doOne, ...))
    })
  ## Now, res is a list with res[[]][[]][[]][[]][[]] corresponding to (tau, family,
  ## d, n, n.sim) ==> need to unlist (exactly the correct number of times)
  res <- ul(ul(ul(ul(res))))
  ## convert result and save
  saveSim(res, vList, repFirst = repFirst, sfile, check = check, doAL = doAL)
}
```

Let us call `doNestForeach()` for our working example, with `seed=NULL`, and `n.sim=1`, respectively.

```
R> res2 <- doNestForeach(varList, sfile = "res2_nested_seq.rds",
+    doOne = doOne, names = TRUE)

R> system.time(res2. <- doNestForeach(varList, seed = NULL,
+    sfile = "res2_nested_NULL.rds", doOne = doOne)

   user  system elapsed
  0.016   0.061   1.698

R> res21 <- doNestForeach(varList.1, sfile = "res21_nested_seq.rds",
+    doOne = doOne, names = TRUE)
```

Next, we demonstrate how l'Ecuyer's random number generator can be used.

```
R> old.seed <- .Random.seed
R> set.seed(LE.seed, kind = "L'Ecuyer-CMRG")
R> n.sim <- get.n.sim(varList.2)
R> seedList <- LEseeds(n.sim)
R> system.time(res22 <- doNestForeach(varList.2, seed = seedList,
+    sfile = "res22_lapply_LEc.rds", doOne = doOne, names = TRUE))

   user  system elapsed
  0.000   0.071   1.699

R> old.seed -> .Random.seed
```

To see that `doNestForeach()` and `doLapply()` lead the same result, let us check for equality of `res2` with `res`. Finally, we check equality of `res22` with `res02` which shows the same for l'Ecuyer's random number generator.

```
R> stopifnot(doRes.equal(res2, res), doRes.equal(res22, res02))
```

A.7. Using Rmpi

The following wrapper function `doRmpi()` utilizes only tools from the R package **Rmpi** for parallel computing on multiple nodes or cores in R via MPI. With `load.balancing = TRUE` (the default), the load-balancing version `mpi.applyLB()` is utilized (otherwise `mpi.apply()`) which sends the next sub-job to a worker who just finished one.

```
doRmpi <- function(vList, nslaves = if ((sz <- Rmpi::mpi.universe.size()) <= 1)
  detectCores() else sz, load.balancing = TRUE, block.size = 1, seed = "seq",
  repFirst = TRUE, sfile = NULL, check = TRUE, doAL = TRUE, subjob. = subjob,
  monitor = FALSE, doOne, exports = character(), ...)
{
  ## see
  ## http://cran.r-project.org/doc/manuals/r-devel/R-exts.html#Suggested-packages
  if (!requireNamespace("Rmpi", quietly = TRUE))
    stop("You must install the CRAN package 'Rmpi' before you can use doRmpi()")

  if (!is.null(r <- maybeRead(sfile)))
    return(r)
  stopifnot(is.function(subjob.), is.function(doOne))
  if (!(is.null(seed) || is.na(seed) || is.numeric(seed) || (is.list(seed) &&
    all(vapply(seed, is.numeric, NA))) || is.character(seed)))
    stop(.invalid.seed.msg)
  if (check) doCheck(doOne, vList, nChks = 1, verbose = FALSE)

  ## monitor checks {here, not in subjob()!}
  if (!(is.logical(monitor) || is.function(monitor)))
    stop(gettextf("'monitor' must be logical or a function like %s",
      "printInfo[[\"default\"]]"))

  ## variables
  pGrid <- mkGrid(vList)
  ngr <- nrow(pGrid)
  ng <- get.nonGrids(vList)  # => n.sim >= 1
  n.sim <- ng$n.sim
  stopifnot(1 <= block.size, block.size <= n.sim)
  if (n.sim%%block.size != 0)
    stop("block.size has to divide n.sim")

  ## use as many workers as available Note: mpi.comm.size(comm) returns the total
  ## number of members in a comm
  comm <- 1  ## communicator number
  if (!Rmpi::mpi.comm.size(comm)) {
    ## <==> no workers are running
    Rmpi::mpi.spawn.Rslaves(nslaves = nslaves)
  }
  ## quiet = TRUE would omit successfully spawned workers
  on.exit(Rmpi::mpi.close.Rslaves())  # close workers spawned by mpi.spawn.Rslaves()
  ## pass global required objects to cluster (required by mpi.apply())
  Rmpi::mpi.bcast.Robj2slave(.Random.seed)
  Rmpi::mpi.bcast.Robj2slave(mkTimer)
  for (e in exports) {
    ee <- substitute(Rmpi::mpi.bcast.Robj2slave(EXP), list(EXP = as.symbol(e)))
    eval(ee)
  }

  ## instead of initExpr, this needs a 'initFunction' + 'initArgs'
  ## if(!missing(initExpr)) do.call(mpi.bcast.cmd, c(list(initFunction), ...))

  ## actual work
  n.block <- n.sim%/%block.size
  res <- ul((if (load.balancing)
    Rmpi::mpi.applyLB else Rmpi::mpi.apply)(seq_len(ngr * n.block),
      function(i) lapply(seq_len(block.size),
      function(k) subjob.((i - 1) * block.size + k, pGrid = pGrid,
      nonGrids = ng$nonGrids, repFirst = repFirst, n.sim = n.sim, seed = seed,
      doOne = doOne, monitor = monitor, ...))))

  ## convert result and save
  saveSim(res, vList, repFirst = repFirst, sfile = sfile, check = check, doAL = doAL)
}
```

Similar as before, we now call `doRmpi()` for our working example, with `seed = NULL`, and `n.sim = 1`, respectively. We also show here, that `seed = NULL` is typically non-reproducible.

```
R> res3  <- doRmpi(varList, sfile = "res3_Rmpi_seq.rds",
+                  doOne = doOne, names = TRUE)

R> system.time(res3. <- doRmpi(varList, seed = NULL,
+    sfile = "res3_Rmpi_NULL.rds", doOne = doOne))

   user  system elapsed
  0.016   0.000   0.019

R> ## shows that seed = NULL is non-reproducible here ==> warnings (2x)
R> set.seed(101)
R> system.time(res3N1 <- doRmpi(varList, seed = NULL,
+    sfile = "res3_RmpiN1_NULL.rds", doOne = doOne))

   user  system elapsed
  0.012   0.000   0.017

R> set.seed(101)
R> system.time(res3N2 <- doRmpi(varList, seed = NULL,
+    sfile = "res3_RmpiN2_NULL.rds", doOne = doOne))

   user  system elapsed
  0.012   0.000   0.013

R> if(identical(res3N1, res3N2)) stop("identical accidentally ??")
R> str(all.equal(res3N1, res3N2)) # => differ quite a bit!
```

```
 chr [1:1607] "Component 1: Component 4: Mean relative difference:
 0.01057269" ...
```

```
R> res31 <- doRmpi(varList.1, sfile = "res31_Rmpi_seq.rds",
+    doOne = doOne, names = TRUE)
R> old.seed <- .Random.seed
R> set.seed(LE.seed, kind = "L'Ecuyer-CMRG")
R> n.sim <- get.n.sim(varList.2)
R> seedList <- LEseeds(n.sim)
R> system.time(res32 <- doRmpi(varList.2, seed = seedList,
+    sfile = "res32_lapply_LEc.rds", doOne = doOne, names = TRUE,
+    monitor = interactive()))
```

```
   user  system elapsed
  0.002   0.000   0.003
```

```
R> old.seed -> .Random.seed
```

To see that `doRmpi()` and `doLapply()` lead the same result, let us check for equality of `res3` with `res`. We also check equality of `res32` with `res02` which shows the same for l'Ecuyer's random number generator.

```
R> stopifnot(doRes.equal(res3, res), doRes.equal(res32, res02))
```

A.8. Using parallel with `mclapply()`

Our next wrapper `doMclapply()` is based on the function `mclapply()` of the recommended R package **parallel**. Although it only parallelizes over multiple cores, it is especially interesting to use if a larger computer cluster is not available or if such a cluster requires complicated setup procedures. Since a cluster is not required for `mclapply()` and thus `doMclapply()` to work, tools like MPI need not be installed on the computer at hand. As a drawback, this method relies on forking and hence is not available on Windows (unless the number of cores is specified as 1 and therefore calculations are not parallel anymore).

```
doMclapply <- function(vList, cores = if (.Platform$OS.type == "windows") 1 else
  detectCores(), load.balancing = TRUE, block.size = 1, seed = "seq",
  repFirst = TRUE, sfile = NULL, check = TRUE, doAL = TRUE, subjob. = subjob,
  monitor = FALSE, doOne, ...)
{
  if (!is.null(r <- maybeRead(sfile))) return(r)
  stopifnot(is.function(subjob.), is.function(doOne))
  if (!(is.null(seed) || is.na(seed) || is.numeric(seed) || (is.list(seed) &&
    all(vapply(seed, is.numeric, NA))) || is.character(seed)))
    stop(.invalid.seed.msg)
  if (check) doCheck(doOne, vList, nChks = 1, verbose = FALSE)

  ## variables
  pGrid <- mkGrid(vList)
  ngr <- nrow(pGrid)
  ng <- get.nonGrids(vList)  # => n.sim >= 1
  n.sim <- ng$n.sim
  stopifnot(1 <= block.size, block.size <= n.sim)
  if (n.sim%%block.size != 0) stop("block.size has to divide n.sim")

  ## monitor checks
  if (!(is.logical(monitor) || is.function(monitor)))
    stop(gettextf("'monitor' must be logical or a function like %s",
      "printInfo[[\"default\"]]"))

  ## actual work
  n.block <- n.sim%/%block.size
  res <- ul(mclapply(seq_len(ngr * n.block), function(i) lapply(seq_len(block.size),
    function(k) subjob.((i - 1) * block.size + k, pGrid = pGrid,
    nonGrids = ng$nonGrids, repFirst = repFirst, n.sim = n.sim, seed = seed,
    doOne = doOne, monitor = monitor, ...)), mc.cores = cores,
    mc.preschedule = !load.balancing, mc.set.seed = FALSE))

  ## convert result and save
  saveSim(res, vList, repFirst = repFirst, sfile = sfile, check = check, doAL = doAL)
}
```

Let us call `doMclapply()` for our working example, with `seed = NULL`, and `n.sim = 1`, respectively.

```
R> ## not if it is only 'detectCores(): require(parallel)
R> options(mc.cores = parallel::detectCores())

R> res4 <- doMclapply(varList, sfile = "res4_mclapply_seq.rds",
+    doOne = doOne, names = TRUE)

R> system.time(res4. <- doMclapply(varList, seed = NULL,
+    sfile = "res4_mclapply_NULL.rds", doOne = doOne))

   user  system elapsed
  0.007   0.000   0.014

R> res41 <- doMclapply(varList.1, sfile = "res41_mclapply_seq.rds",
+    doOne = doOne, names = TRUE)
```

Next, we demonstrate how l'Ecuyer's random number generator can be used.

```
R> old.seed <- .Random.seed
R> set.seed(LE.seed, kind = "L'Ecuyer-CMRG")
R> n.sim <- get.n.sim(varList.2)
R> seedList <- LEseeds(n.sim)
R> system.time(res42 <- doMclapply(varList.2, seed = seedList,
+    sfile = "res42_lapply_LEc.rds", doOne = doOne, names = TRUE,
+    monitor = interactive()))
```

```
   user  system elapsed
  0.001   0.000   0.011
```

```
R> old.seed -> .Random.seed
```

To see that `doMclapply()` and `doLapply()` yield the same result, let us check for equality of `res4` with `res`. We also check equality of `res42` with `res02` which shows the same for l'Ecuyer's random number generator.

```
R> stopifnot(doRes.equal(res4, res), doRes.equal(res42, res02))
```

A.9. Using parallel with `clusterApply()`

The final wrapper `doClusterApply()` is based on the function `clusterApply()` which is the workhorse of various functions (`parLapply()`, `parSapply()`, `parApply()`, etc.) in the R package **parallel** for parallel computations across different nodes or cores. In our setup, this is more efficient than calling the more well-known wrapper function `parLapply()`; see the vignette of **parallel**. With `load.balancing=TRUE` (the default), the load-balancing version `doClusterApplyLB()` is utilized.

```
doClusterApply <- function(vList, cluster = makeCluster(detectCores(),
  type = "PSOCK"), load.balancing = TRUE, block.size = 1, seed = "seq",
  repFirst = TRUE, sfile = NULL, check = TRUE, doAL = TRUE, subjob. = subjob,
  monitor = FALSE, doOne, initExpr, exports = character(), ...)
{
  if (!is.null(r <- maybeRead(sfile))) return(r)
  stopifnot(is.function(subjob.), is.function(doOne))
  if (!(is.null(seed) || is.na(seed) || is.numeric(seed) || (is.list(seed) &&
    all(vapply(seed, is.numeric, NA))) || is.character(seed)))
    stop(.invalid.seed.msg)
  if (check) doCheck(doOne, vList, nChks = 1, verbose = FALSE)
  on.exit(stopCluster(cluster)) # shut down cluster and execution environment

  ## variables
  pGrid <- mkGrid(vList)
  ngr <- nrow(pGrid)
  ng <- get.nonGrids(vList) # => n.sim >= 1
  n.sim <- ng$n.sim
  stopifnot(1 <= block.size, block.size <= n.sim)
  if (n.sim%%block.size != 0) stop("block.size has to divide n.sim")

  ## monitor checks
  if (!(is.logical(monitor) || is.function(monitor)))
    stop(gettextf("'monitor' must be logical or a function like %s",
      "printInfo[[\"default\"]]"))

  clusterExport(cluster, varlist = c(".Random.seed", "mkTimer", exports))
  if (!missing(initExpr))
    clusterCall(cluster, eval, substitute(initExpr))

  ## actual work
  n.block <- n.sim%/%block.size
  res <- ul((if (load.balancing)
    clusterApplyLB else clusterApply)(cluster, seq_len(ngr * n.block),
    function(i) lapply(seq_len(block.size),
    function(k) subjob.((i - 1) * block.size + k, pGrid = pGrid,
    nonGrids = ng$nonGrids, repFirst = repFirst, n.sim = n.sim, seed = seed,
    doOne = doOne, monitor = monitor, ...))))

  ## convert result and save
  saveSim(res, vList, repFirst = repFirst, sfile = sfile, check = check, doAL = doAL)
}
```

Let us call `doClusterApply()` with `seed=NULL` and `n.sim=1`, respectively; note that we have already called it for our working example in Section 3.

```
R> system.time(res5. <- doClusterApply(varList, seed = NULL,
+    sfile = "res5_clApply_NULL.rds", doOne = doOne))

   user  system elapsed
  0.006   0.004   0.010

R> res51 <- doClusterApply(varList.1, sfile = "res51_clApply_seq.rds",
+    doOne = doOne, names = TRUE)
```

Next, we demonstrate how l'Ecuyer's random number generator can be used.

```
R> old.seed <- .Random.seed
R> set.seed(LE.seed, kind = "L'Ecuyer-CMRG")
R> n.sim <- get.n.sim(varList.2)
R> seedList <- LEseeds(n.sim)
R> system.time(res52 <- doClusterApply(varList.2, seed = seedList,
+    sfile = "res52_clApply_LEc.rds", doOne = doOne, names = TRUE,
+    monitor = interactive()))

   user  system elapsed
  0.001   0.000   0.002

R> old.seed -> .Random.seed
```

We already checked in Section 3 that `doClusterApply()` and `doLapply()` lead the same result, so we only have left to check equality for l'Ecuyer's random number generator.

```
R> stopifnot(doRes.equal(res52, res02))
```

B. Timing comparison of different parallelization methods

Note that we have measured elapsed times for our running example on half a dozen Linux platforms with between 4 and 24 cores, always only on one (multi-core) machine. Our running example is clearly *unrealistically small* and hence, parallelization may well cost more than running a single thread by `doLapply()`.

Running everything on a set of computers the (20% trimmed) mean values of elapsed times for two machines were between 4 and 20 seconds, and scaled as multiples of `doForeach` are

```
R> toLatex(ftab(t(round(all.F[, c("ada-6", "lynne")], 1)),
+    dnms = c("mach", "f")), do.table = FALSE)
```

mach | *f*	doForeach	doNestForeach	doMclapply	doLapply	doRmpi	doClusterApply
ada-6	1.0	1.5	2.2	3.7	3.0	3.2
lynne	1.0	4.2	2.9	2.5	3.5	4.3

where `ada-6` has been an old compute server (Quad-Core (8 threads) AMD Opteron 2380, 32 GB RAM) and `lynne` a (quite fast, summer 2014) 8 core Intel i7-4765T desktop with 16 GB RAM. Note that `doNestForeach` performed quite bad on lynne, contrary to all other platforms considered, where it typically was close to `doForeach`. Further note that indeed, on lynne, `doLapply()` was faster than all but `doForeach`, clearly showing that this demo running example is *not* representative for realistic larger scale simulations.

Affiliation:

Marius Hofert
Department of Statistics and Actuarial Science
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada N2L 3G1
E-mail: `mhofert@uw.edu`
URL: `http://www.math.ethz.ch/~hofertj/`

Martin Mächler
Seminar für Statistik, HG G 16
ETH Zurich
8092 Zurich, Switzerland
E-mail: `maechler@stat.math.ethz.ch`
URL: `http://stat.ethz.ch/people/maechler`

Journal of Statistical Software `http://www.jstatsoft.org/`
published by the Foundation for Open Access Statistics `http://www.foastat.org/`
February 2016, Volume 69, Issue 4 *Submitted:* 2013-09-18
`doi:10.18637/jss.v069.i04` *Accepted:* 2015-01-30