Hofert, Marius

Article

Implementing the rearrangement algorithm: An example from computational risk management

Risks

Provided in Cooperation with:
MDPI – Multidisciplinary Digital Publishing Institute, Basel

Suggested Citation: Hofert, Marius (2020) : Implementing the rearrangement algorithm: An example from computational risk management, Risks, ISSN 2227-9091, MDPI, Basel, Vol. 8, Iss. 2, pp. 1-28, https://doi.org/10.3390/risks8020047

This Version is available at:
https://hdl.handle.net/10419/258001

risks

MDPI

Article

Implementing the Rearrangement Algorithm: An Example from Computational Risk Management

Marius Hofert

Department of Statistics and Actuarial Science, University of Waterloo, 200 University Avenue West, Waterloo, ON N2L 3G1, Canada; marius.hofert@uwaterloo.ca

Received: 20 April 2020; Accepted: 8 May 2020; Published: 14 May 2020

Abstract: After a brief overview of aspects of computational risk management, the implementation of the rearrangement algorithm in R is considered as an example from computational risk management practice. This algorithm is used to compute the largest quantile (worst value-at-risk) of the sum of the components of a random vector with specified marginal distributions. It is demonstrated how a basic implementation of the rearrangement algorithm can gradually be improved to provide a fast and reliable computational solution to the problem of computing worst value-at-risk. Besides a running example, an example based on real-life data is considered. Bootstrap confidence intervals for the worst value-at-risk as well as a basic worst value-at-risk allocation principle are introduced. The paper concludes with selected lessons learned from this experience.

Keywords: computational risk management; rearrangement algorithm; implementation; R; bootstrap; worst value-at-risk allocation

1. Introduction

Computational risk management is a comparably new and exciting field of research at the intersection of statistics, computer science and data science. It is concerned with computational problems of *quantitative risk management*, such as algorithms, their implementation, computability, numerical robustness, parallel computing, pitfalls in simulations, run-time optimization, software development and maintenance. At the end of the (business) day, solutions in quantitative risk management are to be run on a computer of some sort and thus concern computational risk management.

There are various factors that can play a role when developing and implementing a solution to a problem from computational risk management in practice, for example:

(1) Theoretical hurdles. Some solutions cannot be treated analytically anymore. Suppose $\boldsymbol{X} = (X_1, \dots, X_d)$ is a d-dimensional random vector of risk factor changes, for example negative log-returns, with marginal distribution functions $F_1, \dots, F_d$ and we are interested in computing the probability $\mathbb{P}(X_j \in (F_j^{\leftarrow}(p_j), F_j^{\leftarrow}(q_j)],$ for all $j)$ of the jth risk factor change to take on values between its p_j- and its q_j-quantile for all j. Even if we knew the joint distribution function H of $\boldsymbol{X}$, computing such a probability analytically involves evaluating H at 2^d different values. For $d \approx 32$, this can already be time-consuming (not to speak of numerical inaccuracies appearing, which render the formula as good as useless) and for $d \approx 260$ the number of points at which to evaluate H is roughly equal to the estimated number of atoms in the universe.

(2) Model risk. The risk of using the wrong model or a model in which assumptions are not fulfilled; every solution we implement is affected by model risk to some degree. In the aforementioned example, not knowing the exact H leads to model risk when computing the given probability by Monte Carlo integration.

(3) The choice of software. The risk of using the wrong software; in the above example, a software not suitable for simulating from joint distribution functions H. Another example is to use a

programming language too low-level for the problem at hand, requiring to implement standard tasks (sampling, optimization, etc.) manually and thus bearing the risk of obtaining unreliable results because of many possible pitfalls one might encounter when developing the tools needed to implement the full-blown solution. These aspects become more crucial nowadays since efficient solutions to different problems are often only available in one software each but companies need to combine these solutions and thus the corresponding software in order to solve these different problems at hand.

(4) Syntax errors. These are compilation errors or execution errors by an interpreter because of a violation of the syntax of the programming language under consideration. Syntax errors are typically easy to detect as the program simply stops to run. Also, many programming languages provide tools for debugging to find the culprits.

(5) Run-time errors. These are errors that appear while a program runs. Quite often, run-time errors are numerical errors, errors that appear, for example, because of the floating-point representation of numbers in double precision. Run-time errors can sometimes be challenging to find, for example, when they only happen occasionally in a large-scale simulation study and introduce a non-suspicious bias in the results. Run-time errors can sometimes also be hard to reproduce.

(6) Semantic errors. These are errors where the code is syntactically correct and the program runs, but it does not compute what is intended (for example, due to a flipped logical statement). If results do not look suspicious or pass all tests conducted, sometimes measuring run time can be the only way to find such problems (for example, if a program finishes much earlier than expected).

(7) Scaling errors. These are errors of an otherwise perfectly fine running code that fails when run at large scale. This could be due to a lack of numerical robustness, the sheer run time of the code, an exploding number of parameters involved, etc.

(8) User errors. These are errors caused by users of the software solution; for example, when calling functions with wrong arguments because of a misinterpretation of their meaning. The associated risk of wrongly using software can, by definition, often be viewed as part of operational risk.

(9) Warnings. Warnings are important signs of what to watch out for; for example, a warning could indicate that a numerical optimization routine has not converged after a predetermined number of steps and only the current best value is returned, which might be far off a local or the global optimum. Unfortunately, especially in large-scale simulation studies, users often suppress warnings instead of collecting and considering them.

(10) Run time. The risk of using or propagating method A over B because the run time of A beats the one of B by a split second or second, not realizing that run time depends on factors such as the hardware used, the current workload, the algorithm implemented, the programming language used, the implementation style, compiler flags, whether garbage collection was used, etc. There is not even a unique concept of time (system vs. user vs. elapsed time).

(11) Development and maintenance. It is significantly more challenging to provide a robust, well developed, maintained and documented class of solutions as part of a bigger, coherent software package rather than a single script with hard-coded values to solve a special case of the same problem. Although stand-alone scripts get specific tasks done by 4 p.m., having a software package available is typically beneficial mid- to long-term. It can significantly reduce the risk of the aforementioned errors by reusing code well tested and applied by the users of the package or it can avoid the temptation of introducing errors long after the code was developed just because it suddenly looks suspicious although it is actually correct.

At the core of all solutions to problems from computational risk management lies an *algorithm*, a well-defined (unambiguous) finite set of instructions (steps) for solving a problem. An implementation of an algorithm in a programming language allows one to see how a problem is actually solved, unlike a formulation in pseudo-code, which is often vague and thus opens the door

for misinterpretation. A classical example is if pseudo-code says "Minimize the function…" without mentioning how initial values or intervals can be obtained. Another example is "Choose a tolerance $\varepsilon > 0$…" but one is left in the dark concerning what suitable tolerances ε are for the problem at hand; they often depend on the unknown output one is interested to compute in the first place, see Step (1) of Algorithm 1 later. Oftentimes, these are the hardest parts to solve of the whole problem and it is important for research and scientific journals to accept corresponding contributions as important instead of brushing them off as "not innovative" or representing "no new contribution".

A fundamental principle when developing an implementation is that of a *minimal working example*. A minimal working example is source code that is a *working example* in the sense that it allows someone else to reproduce a problem (sufficiency) and it is *minimal* in the sense that it is as small and as simple as possible, without non-relevant code, data or dependencies (necessity). Minimal working examples are often constructed from existing larger chunks of code by *divide and conquer*, that is by breaking down the problem into sub-problems and commenting out non-relevant parts until the code becomes simple enough to show the problem (which is then easier to grasp and solve or can be sent to an expert in the field to ask for help).

In this paper, we consider the problem of implementing the *rearrangement algorithm (RA)* of Embrechts et al. (2013) as an example from computational risk management. The RA allows one to compute, for example, the worst value-at-risk of the sum of d risks of which the marginal distributions are known but the dependence is unknown; it can also be used to compute the best value-at-risk or expected shortfall, but we focus on the worst value-at-risk. Section 2 contains the necessary details about the RA. Section 3 addresses how the major workhorse underlying this algorithm can be implemented in a straightforward way in R. This turns out to be inefficient and Section 4 presents ways to improve the implementation. Section 5 utilizes the implementations in our R package `qrmtools` for tracing and to motivate the chosen default tolerances, and Section 6 considers a real data example, presents a bootstrap confidence interval for worst value-at-risk and introduces a basic capital allocation principle based on worst value-at-risk. The lessons learned throughout the paper are summarized in Section 7.

2. The Rearrangement Algorithm in a Nutshell

Let $L_1 \sim F_1, \dots, L_d \sim F_d$ be continuously distributed loss random variables over a predetermined period and let $L^+ = \sum_{j=1}^d L_j$ be the total loss over that time period. From a risk management perspective we are interested in computing *value-at-risk* ($\mathrm{VaR}_\alpha(L^+)$, VaR_α *or* VaR) at confidence level $\alpha \in (0,1)$, that is the α-quantile $F^{\leftarrow}_{L^+}(\alpha) = \inf\{x \in \mathbb{R} : F_{L^+}(x) \geq \alpha\}$ of the distribution function F_{L^+} of L^+; in typical applications, $\alpha \in [0.99, 1)$. If we had enough joint observations, so realizations of the random vector $\boldsymbol{L} = (L_1, \dots, L_d)$, we could estimate $\mathrm{VaR}_\alpha(L^+)$ empirically. A major problem is that one often only has realizations of each of L_j individually, non-synchronized. This typically allows to pin down $F_1, \dots, F_d$ but not the joint distribution function H of $\boldsymbol{L}$. By Sklar's Theorem, such H can be written as

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

for a unique copula C. In other words, although we often know or can estimate $F_1, \dots, F_d$, we typically neither know the copula C nor have enough joint realizations to be able to estimate it. However, the copula C determines the dependence between $L_1, \dots, L_d$ and thus the distribution of L^+ as the following example shows.

Example 1 (Motivation for rearrangements).
Consider $d = 2$ and $L_j \sim F_j(x) = 1 - x^{-1/2}$, $x \geq 1$, $j = 1, 2$, that is both losses are Pareto Type I distributed with parameter $1/2$. The left-hand side of Figure 1 shows $n = 1000$ realizations of $\boldsymbol{L} = (L_1, L_2) = (F_1^{\leftarrow}(U_1), F_2^{\leftarrow}(U_2))$, once under independence, so for $(U_1, U_2) \sim \mathrm{U}(0,1)^2$, and once under comonotonicity, so for $U_1 = U_2 \sim \mathrm{U}(0,1)$. We see that the different dependence structures directly influence the shape of the realizations of $\boldsymbol{L}$.

```
> qF <- function(y) (1-y)^(-2) # quantile function (the same for both margins here)
> n <- 1000 # sample size
> set.seed(271) # for reproducibility
> L1 <- qF(runif(n)) # losses 1
> L2 <- qF(runif(n)) # losses 2
> L.indep <- cbind(L1, L2)
> L.comon <- cbind(L1, sort(L2)[rank(L1)]) # sort L2 comonotone to L1
> plot(L.indep, log = "xy", xlab = expression(L[1]), ylab = expression(L[2]))
> points(L.comon, pch = 4)
> legend("top", bty = "n", pch = c(1, 4),
+        legend = c("Independence", "Comonotonicity"))
> mtext(substitute(n==n.~"samples of two Pareto Type 1 losses with parameter 0.5",
+       list(n. = n)), side = 4, line = 0.5, adj = 0)
```

As the middle of Figure 1 shows, the dependence also affects $\mathrm{VaR}_\alpha(L^+)$*. This example is chosen to be rather extreme, (perhaps) in (stark) contrast to intuition,* $\mathrm{VaR}_\alpha(L^+)$ *under independence is even larger than under comonotonicity, for all* α*; this can also be shown analytically in this case in a rather tedious calculation; see (Hofert et al. 2020, Exercise 2.28).*

```
> alpha <- seq(0.5, 0.9999, length.out = 301)
> VaR.L.indep.Par <- 2 * (1+sqrt(1-(1-alpha)^2)) / (1-alpha)^2
> VaR.L.comon.Par <- 2 * qF(alpha)
> plot(alpha, VaR.L.indep.Par, type = "l", log = "y", lty = 2, xlim = range(alpha),
+      ylim = range(VaR.L.indep.Par, VaR.L.comon.Par),
+      xlab = expression(alpha), ylab = expression(VaR[alpha](L^{'+'})))
> lines(alpha, VaR.L.comon.Par)
> legend("topleft", bty = "n", lty = c(2, 1),
+        legend = c("Independence", "Comonotonicity"))
> mtext("Two Pareto Type 1 losses with parameter 0.5", side = 4, line = 0.5, adj = 0)
```

The right-hand side of Figure 1 shows the same plot with the marginal distributions now being gamma. In this case, comonotonicity leads to larger values of $\mathrm{VaR}_\alpha(L^+)$ *than independence, but only for sufficiently large* α*; the corresponding turning point seems to get larger the smaller the shape parameter of the second margin, for example.*

```
> shape <- c(1, 1/2)
> VaR.L.indep.Ga <- qgamma(alpha, shape = sum(shape))
> VaR.L.comon.Ga <- qgamma(alpha, shape = shape[1]) + qgamma(alpha, shape = shape[2])
> plot(alpha, VaR.L.indep.Ga, type = "l", log = "y", lty = 2, xlim = range(alpha),
+      ylim = range(VaR.L.indep.Ga, VaR.L.comon.Ga),
+      xlab = expression(alpha), ylab = expression(VaR[alpha](L^{'+'})))
> lines(alpha, VaR.L.comon.Ga)
> legend("topleft", bty = "n", lty = c(2, 1),
+        legend = c("Independence", "Comonotonicity"))
> mtext(substitute("Two Gamma losses with shapes"~list(s1, s2),
+       list(s1 = shape[1], s2 = shape[2])), side = 4, line = 0.5, adj = 0)
```

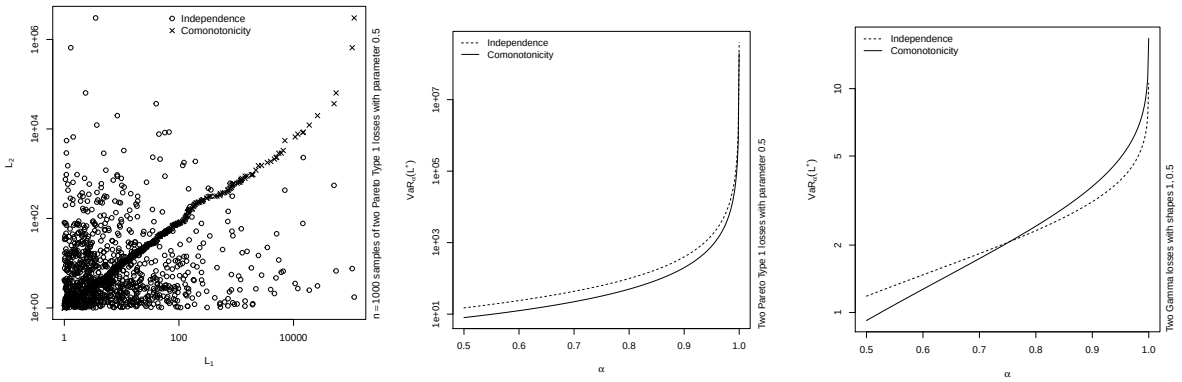

Figure 1. One thousand realizations of independent L_1, L_2 from a Pareto Type I distribution with parameter $1/2$ (**left**), $\alpha \mapsto \mathrm{VaR}_\alpha(L^+)$ for these L_1, L_2 (**middle**) and the same for two Gamma losses with parameters 1 and $1/2$ (**right**).

Note that we did not utilize the stochastic representation $(L_1, L_2) = (F_1^{\leftarrow}(U), F_2^{\leftarrow}(U))$, $U \sim \mathrm{U}(0,1)$, *for sampling* $\boldsymbol{L} = (L_1, L_2)$. *Instead, we sorted the realizations of* L_2 *so that their ranks are aligned with those of* L_1, *in other words, so that the ith-smallest realization of* L_2 *lies next to the ith-smallest realizations of* L_1. *The rows of this sample thus consist of* $(L_{(i)1}, L_{(i)2})$, *where* $L_{(i)j}$ *denotes the ith order statistic of the n realizations* $L_{1j}, \dots, L_{nj}$ *of* L_j. *Such a* rearrangement *mimics comonotonicity between the realizations of* L_1 *and* L_2. *Note that this did not change the realizations of* L_1 *or* L_2 *individually, so it did not change the marginal realizations, only the joint ones.*

As we learned from Example 1, by rearranging the marginal loss realizations, we can mimic different dependence structures between the losses and thus influence $\mathrm{VaR}_\alpha(L^+)$. In practice, the *worst* $\mathrm{VaR}_\alpha(L^+)$, denoted by $\overline{VaR}_\alpha(L^+)$, is of interest, that is the largest $\mathrm{VaR}_\alpha(L^+)$ for given margins $L_1 \sim F_1, \dots, L_d \sim F_d$. The following remark motivates an objective when rearranging marginal loss realizations in order to increase $\mathrm{VaR}_\alpha(L^+)$ and thus approximate $\overline{VaR}_\alpha(L^+)$.

Remark 1 (**Objective of rearrangements**).

(1) By Example 1, the question which copula C maximizes $\mathrm{VaR}_\alpha(L^+)$ *is thus, at least approximately, the question of which reordering of realizations of each of* $L_1, \dots, L_d$ *leads to* $\overline{VaR}_\alpha(L^+)$. *For any C, the probability of exceeding* $\mathrm{VaR}_\alpha(L^+)$ *is (at most, but for simplicity let us assume exactly)* $1-\alpha$. *How that probability mass is distributed beyond* $\mathrm{VaR}_\alpha(L^+)$ *depends on C. If we find a C such that* L^+ *has a rather small variance* $\mathrm{var}(L^+)$, *more probability mass will be concentrated around a single point which can help to pull* $\mathrm{VaR}_\alpha(L^+)$ *further into the right tail and thus increase* $\mathrm{VaR}_\alpha(L^+)$; *Figure 2 provides a sketch in terms of a skewed* t_3 *density. If that single point exists and if* $\mathrm{var}(L^+) = 0$, *then it must be* expected shortfall $\mathrm{ES}_\alpha(L^+) = \frac{1}{1-\alpha}\int_\alpha^1 \mathrm{VaR}_u(L^+)\,\mathrm{d}u$, *which provides an upper bound to* $\mathrm{VaR}_\alpha(L^+)$.

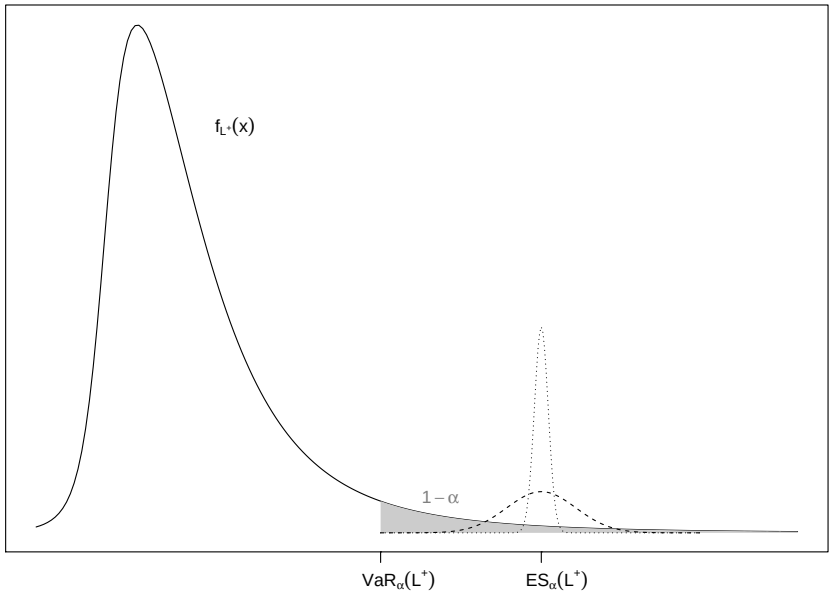

Figure 2. Skewed t_3 density of L^+ with $\mathrm{VaR}_\alpha(L^+)$ and the probability mass of (at most) $1-\alpha$ exceeding it.

(2) *It becomes apparent from Figure 2 that the distribution of L^+ below its α-quantile is irrelevant for the location of* $\mathrm{VaR}_\alpha(L^+)$. *It thus suffices to consider losses $L_1,\dots,L_d$ beyond their marginal α-quantiles $F_1^{\leftarrow}(\alpha),\dots,F_d^{\leftarrow}(\alpha)$, so the conditional losses $L_j \mid L_j > F_j^{\leftarrow}(\alpha)$, $j=1,\dots,d$, and their copula C^α, called* worst VaR copula. *Note that since α is typically rather close to 1, the distribution of $L_j \mid L_j > F_j^{\leftarrow}(\alpha)$ (the tail of L_j) is typically modeled by a continuous distribution; this can be justified theoretically by the Pickands–Balkema–de-Haan Theorem, see (McNeil et al. 2015, Section 5.2.1).*

(3) *In general, if $\mathrm{var}(L^+) = 0$ cannot be attained, the smallest point of the support of L^+ given that $L_j > F_j^{\leftarrow}(\alpha)$, $j=1,\dots,d$, is taken as an approximation to $\overline{VaR}_\alpha(L^+)$.*

The RA aims to compute $\overline{VaR}_\alpha(L^+)$ by rearranging realizations of $L_j \mid L_j > F_j^{\leftarrow}(\alpha)$, $j=1,\dots,d$, such that their sum becomes more and more constant, in order to approximately obtain the smallest possible variance and thus the largest $\mathrm{VaR}_\alpha(L^+)$. There are still fundamental open mathematical questions concerning the convergence of this algorithm, but this intuition is sufficient for us to understand the RA and to study its implementation. To this end, we now address the remaining underlying concepts we need.

For each margin $j=1,\dots,d$, the RA uses two discretizations of $F_j^{\leftarrow}$ beyond the probability α as realizations of $L_j \mid L_j > F_j^{\leftarrow}(\alpha)$ to be rearranged. In practice, one would typically utilize the available data from L_j to estimate its distribution function F_j and then use the implied quantile function $F_j^{\leftarrow}$ of the fitted F_j to obtain such discretizations; see Section 6. Alternatively, $F_j^{\leftarrow}$ could be specified by expert opinion. The RA uses two discretizations to obtain upper and lower bounds to $\overline{VaR}_\alpha(L^+)$, which, when sufficiently close, can be used to obtain an estimate of $\overline{VaR}_\alpha(L^+)$, for example, by taking the midpoint. The two discretizations are stored in the matrices

$$\underline{X}^\alpha = \left(F_j^-\left(\alpha + (1-\alpha)\frac{i-1}{N}\right)\right)_{i=1,\dots,N}^{j=1,\dots,d} \quad \text{and} \quad \overline{X}^\alpha = \left(F_j^-\left(\alpha + (1-\alpha)\frac{i}{N}\right)\right)_{i=1,\dots,N}^{j=1,\dots,d}. \tag{1}$$

These matrices are the central objects the RA works with. For simplicity of the argument, we write $X^\alpha = (X_{ij}^\alpha)$ if the argument applies to $\underline{X}^\alpha$ or $\overline{X}^\alpha$. The RA iterates over the columns of X^α and successively rearranges each column in a way such that the row sums $(\sum_{j=1}^d X_j^\alpha)_{i=1,\dots,N}$ of the rearranged matrix have a smaller variance; compare with Remark 1.

In each rearrangement step, the RA rearranges the jth column $\boldsymbol{X}_j^\alpha$ of X^α *oppositely* to the vector of row sums $\boldsymbol{X}_{-j}^\alpha = \sum_{k=1,k\neq j}^d \boldsymbol{X}_k^\alpha$ of all other columns, where $\boldsymbol{X}_k^\alpha = (X_{1k}^\alpha, \dots, X_{Nk}^\alpha)$. Two vectors $\boldsymbol{a} = (a_1, \dots, a_N)$ and $\boldsymbol{b} = (b_1, \dots, b_N)$ are called *oppositely ordered* if $(a_{i_1} - a_{i_2})(b_{i_1} - b_{i_2}) \leq 0$ for all $i_1, i_2 = 1, \dots, N$. After oppositely ordering, the (second-, third-, etc.) smallest component of $\boldsymbol{a}$ lies next to the (second-, third-, etc.) largest of $\boldsymbol{b}$, which helps decrease the variance of their componentwise sum. To illustrate this, consider the following example.

Example 2 (Motivation for oppositely ordering columns).

We start by writing an auxiliary function to compute a matrix X^α with the modification that for $\overline{X}^\alpha$, 1-quantiles $F_j^-(1)$ appearing in the last row are replaced by $F_j^-\left(\alpha + (1-\alpha)\frac{N-1/2}{N}\right) = F_j^-\left(1 - \frac{1-\alpha}{2N}\right)$ to avoid possible infinities; we come back to this point later. It is useful to start writing functions at this point, as we can reuse them later and rely on them for computing partial results, inputs needed, etc. In R, the function `stopifnot()` *can be used to check inputs of functions. If any of the provided conditions fails, this would produce an error. For more elaborate error messages, one can work with* `stop()`*. Input checks are extremely important when functions are* exported *in a package (so visible to a user of the package) or even made available to users by simply sharing R scripts. Code is often run by users who are not experts in the underlying theory or mechanisms and good input checks can prevent them from wrongly using the shared software; see Section 1 Point (8). Also, documenting the function (here: Roxygen-style documentation) is important, providing easy to understand information about what the function computes, meaning of input parameters, return value and who wrote the function.*

```
> ##' @title Matrix Containing Marginal Quantile Functions Evaluated Beyond
> ##'        the Confidence Level
> ##' @param N integer > 0 giving the number of discretization points used
> ##' @param alpha confidence level in (0,1)
> ##' @param qF list of marginal quantile functions
> ##' @param upper logical indicating whether the matrix for the upper (the default)
> ##'        or the lower discretization matrix is computed
> ##' @return (N, length(qF))-matrix containing in the jth column the jth marginal
> ##'         quantile function evaluated on an equidistant grid in [alpha, 1];
> ##'         if upper is TRUE, the last point is treated separately to avoid
> ##'         1-quantiles which may be Inf.
> ##' @author Marius Hofert
> quantile_matrix <- function(N, alpha, qF, upper = TRUE)
+ {
+     stopifnot(N > 0, N %% 1 == 0, 0 < alpha, alpha < 1, is.list(qF),
+               sapply(qF, is.function), is.logical(upper))
+     p <- alpha + (1-alpha) * (1:(N-1)) / N # grid of probabilities in (alpha, 1)
+     p <- if(upper) { # for upper bound
+              c(p, (p[length(p)] + 1) / 2) # attach last point (midpoint)
+          } else { # for lower bound
+              c(alpha, p) # attach first point = alpha
+          }
+     sapply(qF, function(qF.) qF.(p)) # (N, d)-matrix X^alpha
+ }
```

Consider $d = 2$ and $L_j \sim F_j(x) = 1 - (1+x)^{-\theta_j}$, $x \geq 0$, $j = 1, 2$, that is each of the two losses is Pareto distributed; we choose $\theta_1 = 3/2$ and $\theta_2 = 2$ here. We start by building the list of marginal quantile functions and then compute the corresponding (here: upper) quantile matrix $X^\alpha (= \overline{X}^\alpha)$ for $\alpha = 0.9$ and $N = 10\,000$.

```
> library(qrmtools)
> qF <- lapply(c(1.5, 2), function(th) function(p) qPar(p, shape = th))
> X <- quantile_matrix(1e4, alpha = 0.9, qF = qF)
```

The columns in X^α are sorted in increasing order, so mimic comonotonicity in the joint tail, which leads to the following variance estimate of the row sums.

```
> var(rowSums(X))
```

```
[1] 3519.706
```

If we randomly shuffle the first column, so mimicking independence in the joint tail, we obtain the following variance estimate.

```
> X. <- X
> set.seed(271) # for reproducibility
> X.[,1] <- sample(X.[,1]) # randomly shuffling the first column
> var(rowSums(X.)) # estimate the variance of the row sums
```

```
[1] 2664.61
```

Now if we oppositely order the first column with respect to the sum of all others, here the second column, we can see that the variance of the resulting row sums will decrease; note that we mimic countermonotonicity in the joint tail in this case.

```
> X. <- X
> set.seed(271) # for reproducibility
> X.[,1] <- sort(X.[,1], decreasing = TRUE)[rank(X.[,2])] # opposite reordering
> var(rowSums(X.)) # estimate the variance of the row sums
```

```
[1] 2614.33
```

The two marginal distributions largely differ in their heavy-tailedness, which is why the variance of their sum, even when oppositely reordered is still rather large. For $\theta_1 = \theta_2 = 2$, one obtains 138.6592*, for two standard normal margins* 0.0808 *and for two standard uniform distributions* 0*.*

As a last underlying concept, the RA utilizes the *minimal row sum operator*

$$s(X) = \min_{1 \le i \le N} \sum_{1 \le j \le d} X_{ij} \quad \text{for } X \in \mathbb{R}^{N \times d},$$

which is motivated in Remark 1 Part (3). We are now ready to provide the RA.

The RA as introduced in Embrechts et al. (2013) states that $\underline{s}_N \le \overline{s}_N$ and in practice $\underline{s}_N \approx \overline{s}_N \approx \overline{VaR}_\alpha(L^+)$. Furthermore, the initial randomizations in Steps (2.2) and (3.2) are introduced to avoid convergence problems of $\overline{s}_N - \underline{s}_N \to 0$.

3. A First Implementation of the Basic Rearrangement Step

When implementing an algorithm such as the RA, it is typically a good idea to divide and conquer, which means breaking down the problem into sub-problems and addressing those first. Having implemented solutions to the sub-problems (write functions!), one can then use them as black boxes to implement the whole algorithm. In our case, we can already rely on the function `quantile_matrix()` for Steps (2.1) and (3.1) of the RA and we have already seen how to oppositely order two columns in Example 2. We also see that apart from which matrix is rearranged, Steps (2) and (3) are equal. We thus focus on this step in what follows.

Example 3 (Basic rearrangements).
We start with a basic implementation for computing one of the bounds $\underline{s}_N, \overline{s}_N$ of the RA.

```
> ##' @title Basic Rearrangements
> ##' @param X (N, d)-matrix of marginal quantiles beyond the confidence level
> ##' @param tol convergence tolerance determining when to stop rearranging columns;
> ##'        can also be NULL in which case the algorithm runs until there was no change
> ##'        in the matrix of rearranged columns after one iteration over all columns.
> ##' @return list with the minimal row sums after the algorithm terminates and the
> ##'         corresponding rearranged matrix X
> ##' @author Marius Hofert
> basic_rearrange <- function(X, tol)
+ {
+     ## Random column permutations and basic setup
+     X <- apply(X, 2, function(x) x[sample(seq_along(x))]) # randomly permute each column
+     d <- ncol(X)
+     Y <- X
+     Y.rs <- rowSums(Y) # Y row sums
+     m.rs.old <- min(Y.rs) # initial minimal row sums (to compare against later)
+
+     ## Main
+     while (TRUE) {
+         ## Loop over all columns and oppositely reorder the jth with respect to
+         ## the sum of all others
+         for(j in 1:d) {
+             Y.rs.mj <- rowSums(Y[,-j, drop = FALSE]) # row sums of all but the jth column
+             ## Oppositely order the jth column with respect to the sum of all others
+             Y[,j] <- sort(Y[,j], decreasing = TRUE)[rank(Y.rs.mj, ties.method = "first")]
+         }
+         Y.rs <- rowSums(Y) # update row sums after reordering the jth column
+         m.rs.new <- min(Y.rs) # update minimal row sum
+
+         ## Check stopping criterion
+         tol. <- abs((m.rs.new - m.rs.old) / m.rs.old) # relative change of minimal row sum
+         tol.reached <- if(is.null(tol)) { # if NULL
+             identical(Y, X) # stop only if matrix did not change after d iterations
+         } else { tol. <= tol } # reached tolerance if tol. <= tol
+
+         ## If fulfilled, stop, otherwise update and continue
+         if(tol.reached) { # if tolerance was reached
+             break # break while()
+         } else { # update (and continue)
+             m.rs.old <- m.rs.new
+             X <- Y
+         }
+     }
+
+     ## Return
+     list(worst.VaR = min(rowSums(Y)), X.rearranged = Y)
+ }
```

In comparison to Algorithm 1, our basic implementation already uses a relative rather than an absolute convergence tolerance ε (more intuitively named `tol` *in our implementation), the former is more suitable here as we do not know $\overline{VaR}_\alpha(L^+)$ and therefore cannot judge whether an absolute tolerance of, say,* 0.001 *is reasonable. Also, we terminate if the attained relative tolerance is less than or equal to* `tol` *since including equality allows us to choose* `tol = 0`*; see Steps (2.4) and (3.4) of Algorithm 1. Furthermore, we allow the tolerance to be* `NULL` *in which case the columns are rearranged until all of them are oppositely ordered with respect to the sum of all*

other. The choice `tol = NULL` *is good for testing, but typically results in much larger run times and rarely has an advantage over* `tol = 0`.

Algorithm 1: Rearrangement algorithm for computing bounds on $\overline{VaR}_\alpha(L^+)$

(1) *Fix a confidence level* $\alpha \in (0,1)$, *marginal quantile functions* $F_1^-, \dots, F_d^-$, *a number of discretization points* $N \in \mathbb{N}$ *and an absolute convergence tolerance* $\varepsilon \geq 0$.

(2) *Compute the lower bound:*

  (2.1) *Define the matrix* $\underline{X}^\alpha$ *as in* (1).

  (2.2) *Permute randomly the elements in each column of* $\underline{X}^\alpha$.

  (2.3) *Set* $\underline{Y}^\alpha = \underline{X}^\alpha$. *For* $1 \leq j \leq d$, *rearrange the jth column of the matrix* $\underline{Y}^\alpha$ *so that it becomes oppositely ordered to the sum of all other columns.*

  (2.4) *While* $s(\underline{Y}^\alpha) - s(\underline{X}^\alpha) \geq \varepsilon$, *set* $\underline{X}^\alpha$ *to* $\underline{Y}^\alpha$ *and repeat Step (2.3).*

  (2.5) *Set* $\underline{s}_N = s(\underline{Y}^\alpha)$.

(3) *Compute the upper bound:*

  (3.1) *Define the matrix* $\overline{X}^\alpha$ *as in* (1).

  (3.2) *Permute randomly the elements in each column of* $\overline{X}^\alpha$.

  (3.3) *Set* $\overline{Y}^\alpha = \overline{X}^\alpha$. *For* $1 \leq j \leq d$, *rearrange the jth column of the matrix* $\overline{Y}^\alpha$ *so that it becomes oppositely ordered to the sum of all other columns.*

  (3.4) *While* $s(\overline{Y}^\alpha) - s(\overline{X}^\alpha) \geq \varepsilon$, *set* $\overline{X}^\alpha$ *to* $\overline{Y}^\alpha$ *and repeat Step (3.3).*

  (3.5) *Set* $\overline{s}_N = s(\overline{Y}^\alpha)$.

(4) *Return* $(\underline{s}_N, \overline{s}_N)$.

The opposite ordering step in our basic implementation contains the argument `ties.method = "first"` *of* `rank()`, *which specifies how ties are handled, namely those ties with smaller index get assigned the smaller rank. Although this was not a problem in Example 2, when* N *is large and* $d > 2$, *ties can quickly arise numerically. It can be important to use a sorting algorithm that has deterministic behavior in this case, as otherwise, the RA might not terminate or not as quickly as it could; see the vignette* `VaR_bounds` *of the R package* `qrmtools` *for more details.*

Finally, we can call `basic_rearrange()` *on both matrices in Equation* (1) *to obtain the lower and upper bounds* $\underline{s}_N, \overline{s}_N$ *and thus implement the RA. We would thus reuse the same code for Steps (2) and (3) of the RA, which means less code to check, improve or maintain.*

Example 4 (Using the basic implementation).

To call `basic_rearrange()` *in a running example, we use the following parameters and build the input matrix* $\overline{X}^\alpha$.

```
> alpha <- 0.99 # confidence level
> d <- 128 # number of margins considered
> theta <- 1 + 2 * (1:d)/(d) # Pareto parameter vector (from heavy- to more light-tailed)
> qF <- lapply(theta, function(th) # list of marginal quantile functions
+     function(p) qPar(p, shape = th))
> N <- 1e4 # number of discretization points (= number of rows)
> eps <- 0 # tolerance to determine numerical convergence
> X <- quantile_matrix(N, alpha = alpha, qF = qF) # build input matrix
```

We can now call `basic_rearrange()` *and also measure the elapsed time in seconds of this call.*

```
> set.seed(271) # for reproducibility
> time.basic <- system.time(res.basic <- basic_rearrange(X, tol = eps))[["elapsed"]]
> res.basic[["worst.VaR"]] # worst VaR (upper) approximation for the given setup
```

```
[1] 7635.74
```

```
> time.basic # elapsed time in seconds
```

```
[1] 10.719
```

4. Improvements

It is always good to have a first working version of an algorithm, for example, to compare against in case one tries to improve the code and it becomes harder to read or check. However, as we saw in Example 4, our basic implementation of Example 3 is rather slow even if we chose a rather small N in this running example. Our goal is thus to present ideas how to improve `basic_rearrange()` as a building block of the RA.

4.1. Profiling

As a first step, we profile the last call to see where most of the run time is spent.

```
> Rprof(profiling <- tempfile()) # enable profiling
> res.basic.prof <- basic_rearrange(X, tol = eps) # call
> Rprof(NULL) # disable profiling
> prof <- summaryRprof(profiling)[["by.self"]] # get a summary
> prof[order(prof[,"total.pct"], decreasing = TRUE),] # nicer print (here)
```

```
self.time self.pct total.time total.pct
"basic_rearrange"      0.14     1.75       8.02    100.00
"rowSums"              0.42     5.24       4.10     51.12
"is.data.frame"        3.68    45.89       3.68     45.89
"sort"                 1.74    21.70       2.42     30.17
"rank"                 0.38     4.74       1.28     15.96
"order"                1.08    13.47       1.08     13.47
"sort.int"             0.46     5.74       0.68      8.48
"apply"                0.02     0.25       0.08      1.00
"is.na"                0.04     0.50       0.04      0.50
"sample"               0.02     0.25       0.04      0.50
"aperm.default"        0.02     0.25       0.02      0.25
"sample.int"           0.02     0.25       0.02      0.25
```

Profiling writes out the call stack every so-many split seconds, so checks where the execution currently is. This allows one to measure the time spent in each function call. Note that some functions do not create stack frame records and thus do not show up. Nevertheless, profiling the code is often helpful. It is typically easiest to consider the measurement in percentage (see second and fourth column in the above output) and often in total, so the run time spent in that particular function or the functions it calls (see the fourth column). As this column reveals, two large contributors to run time are the computation of row sums and the sorting.

4.2. Avoiding Summations

First consider the computation of row sums. In each rearrangement step, `basic_rearrange()` computes the row sums of all but the current column. As we explained, this rearrangement step is mathematically intuitive but it is computationally expensive. An important observation is that the row

sums of all but the current column is simply the row sums of all columns minus the current column; in short, $\boldsymbol{X}^{\alpha}_{-j} = \left(\sum_{k=1}^{N} \boldsymbol{X}^{\alpha}_{k}\right) - \boldsymbol{X}^{\alpha}_{j}$. Therefore, if we, after randomly permuting each column, compute the total row sum once, we can subtract the jth column to obtain the row sums of all but the jth column. After having rearranged the jth column, we then add the rearranged jth column to the row sums of all other columns in order to obtain the updated total row sums.

Example 5 (Improved rearrangements).

The following improved version of `basic_rearrange()` *incorporates said idea; to save space, we omit the function header.*

```
> improved_rearrange <- function(X, tol)
+ {
+     ## Random column permutations and basic setup
+     X <- apply(X, 2, function(x) x[sample(seq_along(x))]) # randomly permute each column
+     d <- ncol(X)
+     Y <- X
+     Y.rs <- rowSums(Y) # Y row sums (only computed once)
+     m.rs.old <- min(Y.rs) # initial minimal row sums (to compare against)
+
+     ## Main
+     while (TRUE) {
+         ## Loop over all columns and oppositely reorder the jth with respect to
+         ## the sum of all others
+         for(j in 1:d) {
+             Y.j <- Y[,j] # jth column
+             Y.rs.mj <- Y.rs - Y.j # row sums without jth = total row sums - jth column
+             ## Oppositely order the jth column with respect to the sum of all others
+             Y[,j] <- sort(Y.j, decreasing = TRUE)[rank(Y.rs.mj, ties.method = "first")]
+             Y.rs <- Y.rs.mj + Y[,j] # update total row sum
+         }
+         m.rs.new <- min(Y.rs) # update minimal row sum
+
+         ## Check stopping criterion
+         tol. <- abs((m.rs.new - m.rs.old) / m.rs.old) # relative change of minimal row sum
+         tol.reached <- if(is.null(tol)) { # if NULL
+             identical(Y, X) # stop only if matrix did not change after d iterations
+         } else { tol. <= tol } # reached tolerance if tol. <= tol
+
+         ## If fulfilled, stop, otherwise update and continue
+         if(tol.reached) { # if tolerance was reached
+             break # break while()
+         } else { # update (and continue)
+             m.rs.old <- m.rs.new
+             X <- Y
+         }
+     }
+
+     ## Return
+     list(worst.VaR = min(Y.rs), # we can also reuse the last updated total row sums here
+          X.rearranged = Y)
+ }
```

Next, we compare `improved_rearrange()` *with* `basic_rearrange()`*. We check if we obtain the same result and also measure run time as a comparison.*

```
> set.seed(271) # for reproducibility
> time.imp <- system.time(res.imp <- improved_rearrange(X, tol = eps))[["elapsed"]]
> stopifnot(all.equal(res.imp[["X.rearranged"]], res.basic[["X.rearranged"]])) # comparison
> time.imp # elapsed time in seconds
```

```
[1] 1.652
```

Run time is substantially improved by about 85%. This should be especially helpful for large d, so let us investigate the percentage improvement; here only done for rather small d to show the effect. Figure 3 shows the relative speed-up in percentage of `improved_rearrange()` *over* `basic_rearrange()` *as a function of d for the running example introduced in Example 4.*

```
> d <- round(2^seq(3, 7, by = 0.5)) # dimensions d considered here
> num.d <- length(d) # number of dimensions
> time <- matrix(, nrow = num.d, ncol = 2) # (num.d, 2)-matrix (initialized with NAs)
> colnames(time) <- c("basic", "improved")
> ## Loop over the dimensions and measure elapsed time
> for(j in seq_along(d)) { # loop over dimensions
+     set.seed(271) # for reproducibility
+     time[j, "basic"] <-
+         system.time(basic_rearrange(X[,1:d[j]], tol = eps))[["elapsed"]]
+     set.seed(271) # for reproducibility and a fair comparison
+     time[j, "improved"] <-
+         system.time(improved_rearrange(X[,1:d[j]], tol = eps))[["elapsed"]]
+ }
> ## Plot of percentage improvement in elapsed time
> plot(d, y = ((time[,"basic"] - time[,"improved"]) / time[,"basic"]) * 100,
+      type = "b", log = "x", ylim = c(0, 100),
+      xlab = "Dimension d", ylab = "Relative speed-up (in %)")
```

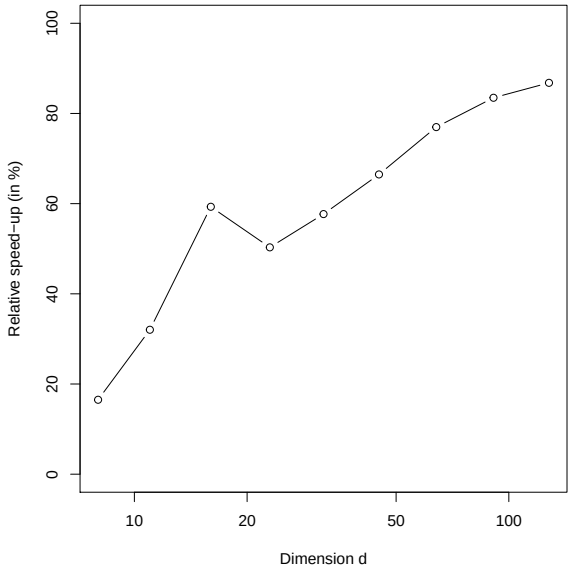

Figure 3. Relative speed-up in percentage of `improved_rearrange()` over `basic_rearrange()` as a function of the dimension for the running example introduces in Example 4.

We can infer that the percentage improvement of `improved_rearrange()` *in comparison to* `basic_rearrange()` *converges to 100% for large d. This is not a surprise since determining the row sums of all but a fixed column requires* `basic_rearrange()` *to compute* $N(d-1)$*-many sums whereas*

`improved_rearrange()` requires only N-many. This is an improvement by $O(d)$, which especially starts to matter for large d.

A quick note on run-time measurement is in order here. In a more serious setup, one would take repeated measurements of run time (at least for smaller d), determine the average run time for each d and perhaps empirical confidence intervals for the true run time based on the repeated measurements. Also, we use the same seed for each method, which guarantees the same shuffling of columns and thus allows for a fair comparison. Not doing so can result in quite different run times and would thus indeed require repeated measurements to get more reliable results.

Finally, the trick of subtracting a single column from the total row sums in order to obtain the row sums of all other columns comes at a price. Although the RA as in Algorithm 1 works well if some of the 1-quantiles appearing in the last row of $\overline{X}^{\alpha}$ are infinity, this does not apply to `improved_rearrange()` anymore since if the last entry in the jth column is infinity, the last entry in the vector of total row sums is infinity and thus the last entry in the vector of row sums of all but the jth column would not be defined (`NaN` in R). This is why `quantile_matrix()` avoids computing 1-quantiles; see Example 2.

4.3. Avoiding Accessing Matrix Columns and Improved Sorting

There are two further improvements we can consider.

The first concerns the opposite ordering of columns. It so far involved a call to `sort()` and one to `rank()`. It turns out that these two calls can be replaced by a nested `order()` call, which is slightly faster than `rank()` alone. We start by demonstrating that we can replicate `rank(, ties.method = "first")` by a nested `order()` call based on standard uniform data.

```
> set.seed(271) # for reproducibility
> x <- runif(1e5) # generate U(0,1) data
> stopifnot(order(order(x)) == rank(x, ties.method = "first")) # check for equality
```

Providing `rank()` with a method to deal with ties is important here as the default assigns average ranks on ties and thus can produce non-integer numbers. One would not guess to see ties in such a small set of standard uniform random numbers, but that is not true; see Hofert (2020) for details.

To see how much faster the nested `order()` call is than `rank()`, we consider a small simulation study. For each of the two methods, we measure elapsed time in seconds when applied 20 times to samples of random numbers of different sizes.

```
> B <- 20 # number of replications
> i <- 5:7 # powers of 10
> n <- 10^i # sample sizes considered
> res <- array(, dim = c(2, length(n), B), # result object
+              dimnames = list(Method = c("rank", "order") , "n" = n, "Replication" = 1:B))
> set.seed(271) # for reproducibility
> for(b in 1:B) { # loop over the number of replications
+     x <- runif(max(n)) # generate U(0,1) data
+     res[1,,b] <- sapply(n, function(n.) # measure run time when using rank()
+         system.time(rank(x[1:n.], ties.method = "first"))[["elapsed"]])
+     res[2,,b] <- sapply(n, function(n.) # measure run time when using order()
+         system.time(order(order((x[1:n.]))))[["elapsed"]])
+ }
> stopifnot(res > 0) # sanity check for logarithmic y-axis later
```

Figure 4 provides a summary in terms of a box plot. We see that the nested `order()` call is faster than `rank()`; percentages by how much the latter is worse are displayed in the labels.

```
> res.ave <- apply(res, 1:2, mean) # average elapsed times
> rank.worse <- round(100 * (res.ave["rank",] - res.ave["order",]) / res.ave["order",])
> tab <- as.data.frame.table(res, responseName = "Elapsed") # all results as a table
> center <- seq_along(n) # x ticks centers
> offset <- c(-0.2, 0.2) # deviations from the center
> x.ticks <- as.vector(outer(offset, center, function(e, c) e+c)) # where to put x axis ticks
> x.tick.labs <- rep(c("rank()", "order()"), length(n)) # x-axis labels
> boxplot(Elapsed ~ Method + n, data = tab, log = "y",
+         xlab = "", ylab = "Elapsed time in seconds",
+         at = x.ticks, names = x.tick.labs, boxwex = rep(0.35, 2 * length(n)))
> x.labs <- as.expression(lapply(seq_along(n), function(k) # labels for secondary first axis
+     substitute(atop(italic(n)==10^i., "rank() worse by "*p*"%"),
+                list(i. = i[k], p = rank.worse[k]))))
> axis(1, at = center, tick = FALSE, line = 3, labels = x.labs) # secondary first axis
```

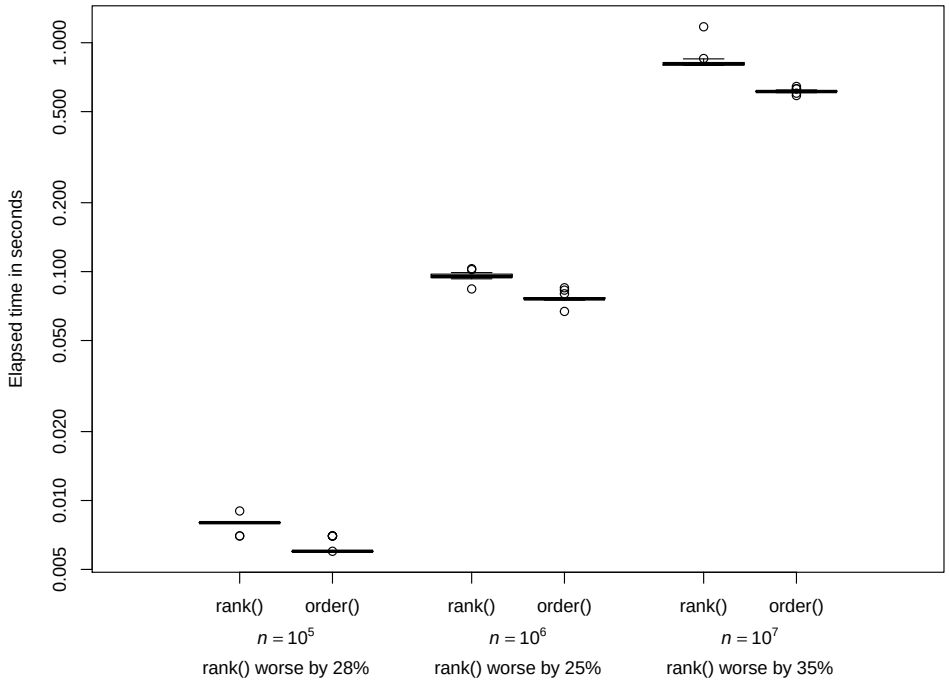

Figure 4. Box plot showing elapsed times in seconds when calling `rank(, ties.method = "first")` and `order(order())` on samples of random numbers of size n. The labels indicate percentages of how much worse the former is in comparison to the latter.

As a second improvement, recall that the RA naturally works on the columns of matrices. Matrices or arrays are internally stored as long vectors with attributes indicating when to "wrap around". Accessing a matrix column thus requires to determine the beginning and the end of that column in the flattened vector. We can thus speed-up `improved_rearrange()` by working with lists of columns instead of matrices.

The following example incorporates these two further improvements and some more.

Example 6 (**Advanced rearrangements**).

The following implementation saves column-access time by working with lists rather than matrices. We also use the slightly faster `.rowSums()` *(as we know the dimensions of the input matrix) instead of* `rowSums()` *for computing the initial row sums once in the beginning, and we incorporate the idea of a nested* `order()` *call*

instead of `rank()`*. Furthermore, before shuffling the columns, we also store their original sorted versions, which allows us to omit the call of* `sort()` *in* `improved_rearrange()`*.*

```
> advanced_rearrange <- function(X, tol)
+ {
+     ## Setup
+     X.lst <- split(X, col(X)) # split 'matrix' X in columns
+     X.lst.sorted <- X.lst # X is assumed to be sorted
+     X.lst <- lapply(X.lst, sample) # randomly permute each 'column' (element of X.lst)
+     N <- nrow(X)
+     d <- ncol(X)
+     X.rs <- .rowSums(do.call(cbind, X.lst), N, d) # compute initial row sums
+     m.rs.old <- min(X.rs) # initial minimal row sums (to compare against)
+
+     ## Main
+     while (TRUE) {
+         ## Update
+         Y.lst <- X.lst # define list representing the 'matrix' Y (former 'matrix' X)
+         Y.rs <- X.rs # row sums of Y (= row sums of X)
+
+         ## Loop over all columns and oppositely reorder the jth with respect to
+         ## the sum of all others
+         for(j in 1:d) {
+             Y.j <- Y.lst[[j]] # jth column
+             Y.rs.mj <- Y.rs - Y.j # row sums without jth = total row sums - jth column
+             ## Oppositely order the jth column with respect to the sum of all others
+             Y.j. <- X.lst.sorted[[j]][order(order(Y.rs.mj, decreasing = TRUE))]
+             ## Update the working 'matrix' and vector of row sums
+             Y.lst[[j]] <- Y.j. # update with rearranged jth column
+             Y.rs <- Y.rs.mj + Y.j. # update total row sum
+         }
+         m.rs.new <- min(Y.rs) # update minimal row sum
+
+         ## Check stopping criterion
+         tol. <- abs((m.rs.new - m.rs.old) / m.rs.old) # relative change of minimal row sum
+         tol.reached <- if(is.null(tol)) { # if NULL
+             identical(Y.lst, X.lst) # stop only if no change after d iterations
+         } else { tol. <= tol } # reached tolerance if tol. <= tol
+
+         ## If fulfilled, stop, otherwise update and continue
+         if(tol.reached) { # if tolerance was reached
+             break # break while()
+         } else { # update (and continue)
+             X.rs <- Y.rs # update the row sums
+             m.rs.old <- m.rs.new # update minimal row sum
+             X.lst <- Y.lst # update 'matrix' X
+         }
+     }
+
+     ## Return
+     list(worst.VaR = min(Y.rs), X.rearranged = do.call(cbind, Y.lst))
+ }
```

We now check for correctness of `advanced_rearrange()` *and measure its run time in our running example. Concerning correctness, for the more involved* `advanced_rearrange()` *it is especially useful to have*

simpler versions available to check against; we use the result of `improved_rearrange()` *here as we already checked it, but could have equally well used the one of* `basic_rearrange()`*.*

```
> set.seed(271) # for reproducibility
> time.adv <- system.time(res.adv <- advanced_rearrange(X, tol = eps))[["elapsed"]]
> stopifnot(all.equal(res.adv[["X.rearranged"]], res.imp[["X.rearranged"]], # comparison
+           check.attributes = FALSE))
> time.adv # elapsed time in seconds
```

```
[1] 0.632
```

Although the gain in run time is not as dramatic as before, we still see an improvement of about 62% in comparison to `improved_rearrange()`*.*

4.4. Comparison with the Actual Implementation

The computational improvements so far have already lead to a percentage improvement of 94% in comparison to `basic_rearrange()` in our running example. The following example compares our fastest function `advanced_rearrange()` so far with our implementation `rearrange()` in the R package `qrmtools` whose development was motivated by a risk management team at a Swiss bank.

Example 7 (Comparison with `rearrange()`).

To save even more run time, the function `rearrange()` *from the R package* `qrmtools` *splits the input matrix into its columns at C level. It also uses an early stopping criterion in the sense that after all columns have been rearranged once, stopping is checked after every column rearrangement. Because of the latter, the result is not quite comparable to our previous versions.*

```
> library(qrmtools)
> set.seed(271) # for reproducibility
> res.rearr <- rearrange(X, tol = eps)
> all.equal(res.rearr[["X.rearranged"]], res.adv[["X.rearranged"]], # comparison
+           check.attributes = FALSE)
```

```
[1] "Mean relative difference: 5.611011e-05"
```

We did not measure run time here since, similar to Heisenberg's uncertainty principle, the difference in run time becomes more and more random; this is in line with the point we made about run-time measurement in the introduction. In other words, the two implementations become closer in run time. This can be demonstrated by repeated measurements.

```
> B <- 20 # number of replications
> res <- matrix(, nrow = B, ncol = 2,
+               dimnames = list(Replication = 1:B, Method = c("advanced", "rearrange")))
> set.seed(271) # for reproducibility
> res[,"advanced"] <-
+     replicate(B, expr = system.time(advanced_rearrange(X, tol = eps))[["elapsed"]])
> set.seed(271) # for reproducibility
> res[,"rearrange"] <-
+     replicate(B, expr = system.time(rearrange(X, tol = eps))[["elapsed"]])
> tab <- as.data.frame.table(res, responseName = "Elapsed") # all results as a table
> boxplot(Elapsed ~ Method, data = tab, xlab = "", ylab = "Elapsed time in seconds",
+         names = c("advanced_rearrange()", "rearrange()"), boxwex = rep(0.35, 2))
```

As we see from the box plot in Figure 5, on average, `rearrange()` *is slightly faster than* `advanced_rearrange()`*. Although* `rearrange()` *could even be made faster, it also computes and returns*

more information about the rearrangements than `advanced_rearrange()`*; for example, the computed minimal row sums after each column rearrangement and the row of the final rearranged matrix corresponding to the minimal row sum in our case here. Given that this is a function applied by users who are not necessarily familiar with the rearrangement algorithm, it is more important to provide such information. Nevertheless,* `rearrange()` *is still faster than* `advanced_rearrange()` *on average.*

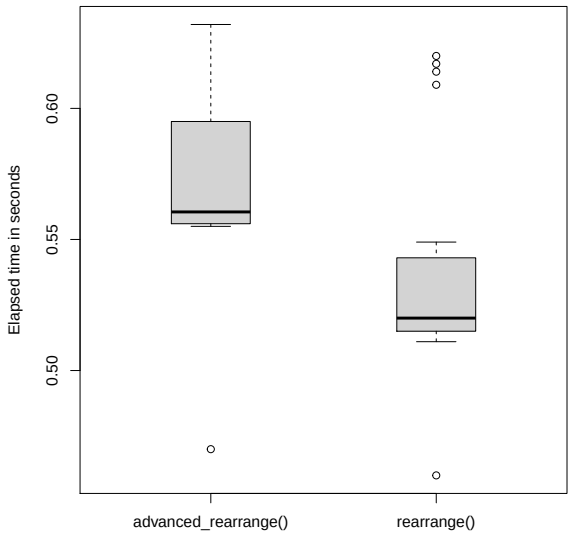

Figure 5. Box plot showing elapsed times in seconds when calling `advanced_rearrange()` and `rearrange()` of the R package `qrmtools` in our running example.

The function `rearrange()` is the workhorse underlying the function `RA()` that implements the rearrangement algorithm in the R package `qrmtools`. The adaptive rearrangement algorithm (ARA) introduced in Hofert et al. (2017) improves the RA in that it introduces the *joint tolerance*, that is the tolerance between $\underline{s}_N$ and $\overline{s}_N$; the tolerance used so far is called *individual tolerance*. The ARA applies column rearrangements until the individual tolerance is met for each of the two bounds and until the joint tolerance is met or a maximal number of column rearrangements has been reached. If the latter is the case, N is increased and the procedure repeated. The ARA implementation `ARA()` in `qrmtools` also returns useful information about how the algorithm proceeded. Overall, `ARA()` neither requires to choose a tolerance nor the number of discretization points, which makes this algorithm straightforward to apply in practice where the choice of such tuning parameters is often unclear. Finding good tuning parameters (adaptively or not) is often a significant problem in newly suggested procedures and not rarely a question open for future research by itself.

5. The Functions `rearrange()` and `ARA()`

In this section we utilize the implementations `rearrange()` and `ARA()` of the rearrangement step and the adaptive rearrangement algorithm.

Example 8 (**Tracing** `rearrange()`)**.**

The implementation `rearrange()` *has a tracing feature. It can produce a lot of output but we here consider the rather minimal example of rearranging the matrix*

$$X = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}.$$

The following code enables tracing (`trace = TRUE`) to see how the column rearrangements proceed; to this end we rearrange until all columns are oppositely ordered to the sum of all other columns (`tol = NULL`), disable random permutation of the columns (`sample = FALSE`) and provide the information that the columns are already sorted here (`is.sorted = TRUE`).

```
> trace <- rearrange(matrix(rep(1:3, times = 3), ncol = 3), tol = NULL,
+                    sample = FALSE, is.sorted = TRUE, trace = TRUE)
```

```
[1,] 1 1 1
[2,] 2 2 2
[3,] 3 3 3
|     -col sum
[1,] 3 1 1    2   5
[2,] 2 2 2    4   6
[3,] 1 3 3    6   7
=   -col sum
[1,] 3 1 1    4   5
[2,] 2 2 2    4   6
[3,] 1 3 3    4   7
= -col sum
[1,] 3 1 1    4   5
[2,] 2 2 2    4   6
[3,] 1 3 3    4   7
=     -col sum
[1,] 3 1 1    2   5
[2,] 2 2 2    4   6
[3,] 1 3 3    6   7
```

A "|" or "=" symbol over the just worked on column indicates whether this column was changed ("|") or not ("=") in this rearrangement step. The last two printed columns provide the row sums of all other columns as well as the new updated total row sums after the rearrangement. As we can see from this example, the rearranged matrix has minimal row sum 5 *and is given by*

$$\begin{pmatrix} 3 & 1 & 1 \\ 2 & 2 & 2 \\ 1 & 3 & 3 \end{pmatrix}, \quad \textit{whereas the matrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

would have led to the larger minimal row sum 6*; this rather constructed example for when the greedy column rearrangements of the RA can fail to provide the maximal minimal row sum was provided by Haus (2015). As more interesting example to trace is to rearrange the matrix*

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 2 \\ 3 & 5 & 4 \\ 4 & 7 & 8 \end{pmatrix};$$

see the vignette `VaR_bounds` of the R package `qrmtools`.

Example 9 (Calling `ARA()`).

We now call `ARA()` in the case of our running example in this paper.

```
> ARA. <- ARA(alpha, qF = qF)
> str(ARA.)
```

```
List of 10
$ bounds              : Named num [1:2] 7596 7638
..- attr(*, "names")= chr [1:2] "low" "up"
$ rel.ra.gap          : num 0.00558
$ tol                 : Named num [1:3] 0 0 0.00558
..- attr(*, "names")= chr [1:3] "low" "up" "joint"
$ converged           : Named logi [1:3] TRUE TRUE TRUE
..- attr(*, "names")= chr [1:3] "low" "up" "joint"
$ N.used              : num 8192
$ num.ra              : Named int [1:2] 767 879
..- attr(*, "names")= chr [1:2] "low" "up"
$ opt.row.sums        :List of 2
..$ low: num [1:767] 5547 5924 6193 6396 6562 ...
..$ up : num [1:879] 5553 5939 6205 6408 6570 ...
$ X                   :List of 2
..$ low: num [1:8192, 1:128] 92.2 92.2 92.2 92.2 92.2 ...
..$ up : num [1:8192, 1:128] 92.2 92.2 92.2 92.2 92.2 ...
$ X.rearranged        :List of 2
..$ low: num [1:8192, 1:128] 797 457 138 202 170 ...
..$ up : num [1:8192, 1:128] 135 107 191 113 963 ...
$ X.rearranged.opt.row:List of 2
..$ low: num [1:128] 107.2 99.5 92.6 86.5 80.9 ...
..$ up : num [1:128] 129 116 106 98 823 ...
```

As we can see, the implementation returns a list with the computed bounds $\underline{s}_N, \overline{s}_N$*, the relative rearrangement gap* $(\overline{s}_N - \underline{s}_N)/\overline{s}_N$*, the two individual and one joint tolerance reached on stopping, a corresponding vector of logicals indicating whether the required tolerances have been met, the number* N *of discretization points used in the final step, the number of considered column rearrangements for each bound, vectors of computed optimal (for worst VaR this means "minimal") row sums after each column rearrangement considered, a list with the two input matrices* $\underline{X}^\alpha, \overline{X}^\alpha$*, a list with the corresponding final rearranged matrices* $\underline{Y}^\alpha, \overline{Y}^\alpha$ *and the rows corresponding to their optimal row sums. An estimate of* $\overline{VaR}_{0.99}(L^+)$ *in our running example can then be computed as follows.*

```
> worst.VaR <- mean(ARA.[["bounds"]]) # estimate of worst VaR
> stopifnot(all.equal(worst.VaR, res.adv[["worst.VaR"]], tol = 5e-3)) # comparison
> worst.VaR
```

```
[1] 7617.149
```

Figure 6 shows the minimal row sum as a function of the number of rearranged columns for each of the two bounds $\underline{s}_N$ *and* $\overline{s}_N$ *on* $\overline{VaR}_{0.99}$ *in our running example. We can see that already after all columns were rearranged once, the actual value of the two bounds does not change much anymore in this case, yet the algorithm still proceeds until the required tolerances are met.*

```
> opt <- ARA.[["opt.row.sums"]] # 2-list with minimal row sums for each bound
> ylim <- range(opt) # range of all minimal row sums
> len <- sapply(opt, length) # length of the components of the 2-list
> ii <- which.min(len) # index of the shorter one
> n <- max(len) # length of the longer one
> opt[[ii]] <- c(opt[[ii]], rep(NA, n - length(opt[[ii]]))) # fill with NA
> min.row.sums <- matrix(unlist(opt), ncol = 2, dimnames = list(NULL, c("low", "up")))
> iter <- seq_len(n) # sequence from 1 to the maximal length
> plot(iter,  min.row.sums[,"up"], type = "l", log = "x", # upper bound
+      xlab = "Number of column rearrangements", ylab = "Minimal row sum")
> lines(iter, min.row.sums[,"low"],  type = "l", lty = 2) # lower bound
> mtext(substitute("Individual and joint tolerances = ("*tol.*"), N used ="~N.,
+                  list(tol. = paste(round(ARA.[["tol"]], 4), collapse = ", "),
+                       N. = ARA.[["N.used"]])),
+       side = 4, line = 0.5, adj = 0)
> legend("bottomright", bty = "n", lty = 1:2,
+        legend = c(expression("Upper bound"~bar(s)[N]),
+                   expression("Lower bound"~underline(s)[N])))
```

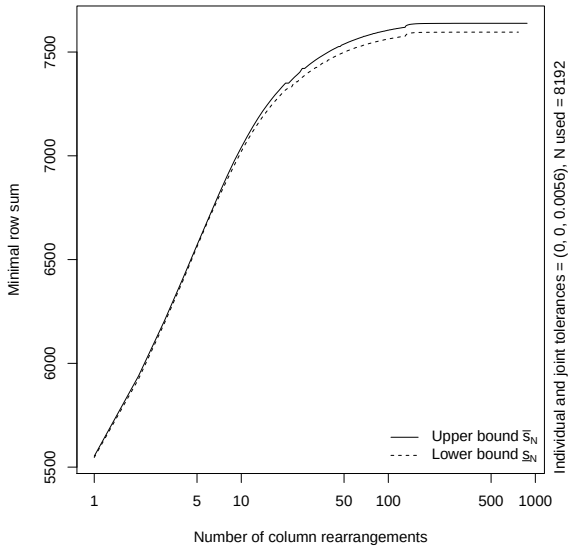

Figure 6. Minimal row sums after each column rearrangement for the lower and upper bounds $\underline{s}_N$ and $\bar{s}_N$ on $\overline{VaR}_{0.99}$ in our running example.

6. Applications

We now consider a real data example and introduce bootstrap confidence intervals for $\overline{VaR}_{0.99}$ as well as a basic worst VaR allocation principle.

Example 10 (**A real-life example**).

We consider negative log-returns of Google, Apple and Microsoft stocks from 2006-01-03 to 2010-12-31.

```
> library(qrmdata)
> data("SP500_const") # load the constituents of the S&P 500
> stocks <- c("GOOGL", "AAPL", "MSFT") # stocks we consider (Google, Apple, Microsoft)
> d <- length(stocks) # dimension
> time <- c("2006-01-03", "2010-12-31") # time period considered
> S <- SP500_const[paste0(time, collapse = "/"), stocks] # pick out data
> stopifnot(all(!is.na(S))) # check that there is no missing data
> X <- -returns(S) # -log-returns
```

We treat the negative log-returns as (roughly) stationary here and consider mean excess plots in Figure 7 to determine suitable thresholds for applying the peaks-over-threshold method.

```
> mean_excess_plot(X[X[,"GOOGL"] > 0, "GOOGL"])
> abline(v = 0.012) # threshold choice
> mean_excess_plot(X[X[,"AAPL"]  > 0, "AAPL"])
> abline(v = 0.021) # threshold choice
> mean_excess_plot(X[X[,"MSFT"]  > 0, "MSFT"])
> abline(v = 0.017) # threshold choice
> u <- c(0.012, 0.021, 0.017)
```

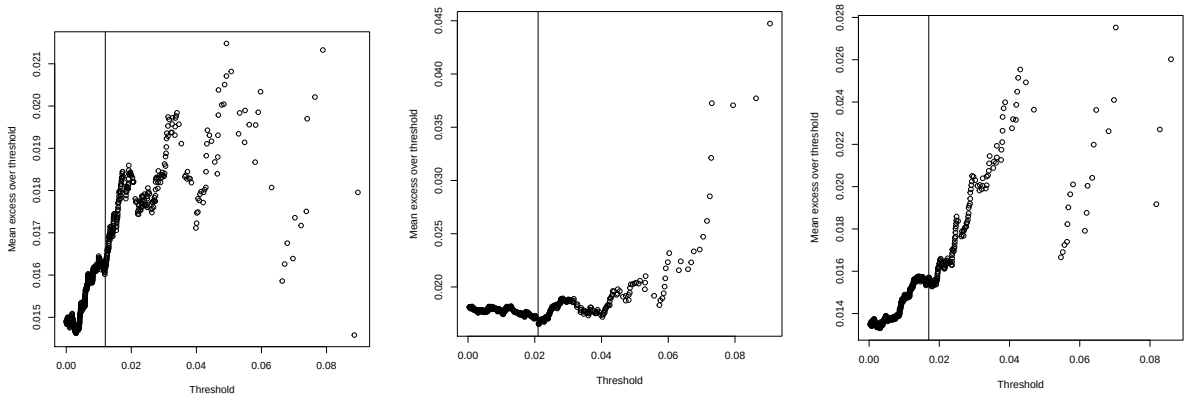

Figure 7. Mean excess plots for the negative log-returns of GOOGL (**left**), AAPL (**middle**) and MSFT (**right**) with chosen thresholds indicated by vertical lines.

We then fit generalized Pareto distributions (GPDs) to the excess losses over these thresholds.

```
> fit <- lapply(1:d, function(j) fit_GPD_MLE(X[X[,j] > u[j],j] - u[j]))
> params <- sapply(fit, function(x) x$par)
> colnames(params) <- names(X)
```

We consider the corresponding quantile functions as approximate quantile functions to our empirical losses; see (McNeil et al. 2015, Section 5.2.3).

```
> p.exceed <- sapply(1:d, function(j) mean(X[,j] > u[j])) #probability of exceeding threshold
> stopifnot(alpha >= max(1 - p.exceed)) # check validity condition of tail approximation
> qF <- lapply(1:d, function(j) function(p) # list of quantile functions
+     qGPDtail(p, threshold = u[j], p.exceed = p.exceed[j],
+              shape = params["shape",j], scale = params["scale",j]))
```

Based on these quantile functions, we can then compute $\overline{VaR}_{0.99}(L^+)$ *with* `ARA()`.

```
> alpha <- 0.99 # confidence level
> ARA. <- ARA(alpha, qF = qF) # compute worst VaR
> mean(ARA.[["bounds"]]) # worst VaR estimate
```

```
[1] 0.2541438
```

Given the additional information obtained from `ARA()`*, we can also visualize a (pseudo-)sample from the worst VaR copula* C^α*, see Figure 8. Such a sample is obtained by computing the pseudo-observations, that is componentwise ranks scaled to* $(0,1)$*, of the rearranged matrix* $\overline{Y}^\alpha$ *corresponding to the upper bound* $\overline{s}_N$*; we could have also considered* $\underline{Y}^\alpha$ *here.*

```
> library(copula)
> U <- pobs(ARA.[["X.rearranged"]]\$up) # pseudo-observations of Y for the upper bound
> colnames(U) <- paste0("U[", 1:3, "]")
> pairs2(U) # worst VaR dependence as a pairs plot
> cloud2(U, screen = list(z = -30, x = -60)) # worst VaR dependence as a 3d plot
```

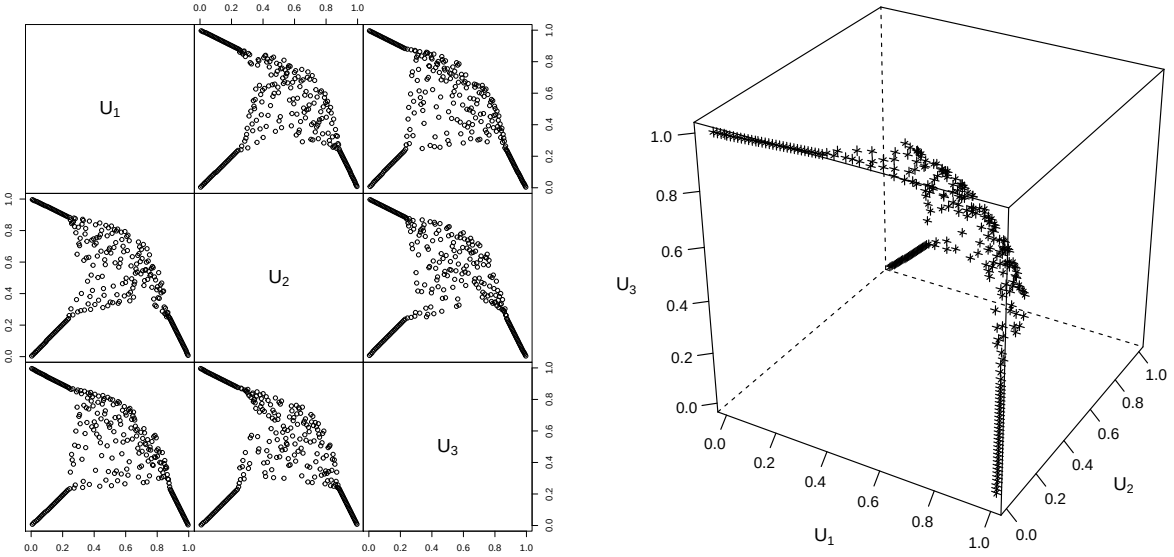

Figure 8. Pairs plot (**left**) and 3D scatter plot (**right**) of the pseudo-observations of the rearranged $\overline{Y}^\alpha$, so a sample from the worst VaR copula C^α.

Also, we can investigate how much the $\overline{VaR}_{0.99}(L^+)$ *bounds* $\underline{s}_N$ *and* $\overline{s}_N$ *change with changing individual relative tolerance. Figure 9 motivates the use of* 0 *as individual relative tolerance; this is often quite a bit faster than* `NULL` *and provides similar* $\overline{VaR}_{0.99}(L^+)$ *bounds.*

```
> itol <- c(NA, seq(0, 0.5, length.out = 21)) #individual tolerances used; NA encodes 'NULL'
> res <- t(sapply(itol, function(t) # worst VaR bounds for all individual tolerances
+     ARA(alpha, qF = qF, reltol = c(if(is.na(t)) NULL else t, 0.01))[["bounds"]]))
> colnames(res) <- c("low", "up")
> plot(itol, res[,"up"], type = "b", log = "y", ylim = range(res),
+      xlab = "Individual relative tolerance",
+      ylab = substitute("Bounds on"~bar(VaR)[a](L^{"+"}), list(a = alpha)))
> lines(itol, res[,"low"], type = "b", lty = 2)
> points(c(0, 0), res[1,],  pch = 3) # add 'NULL' case at 0
> legend("topright", bty = "n", lty = c(1:2, NA), pch = c(NA, NA, 3),
+        legend = c(expression("Upper bound"~bar(s)[N]),
+                   expression("Lower bound"~underline(s)[N]), "case 'NULL'"))
```

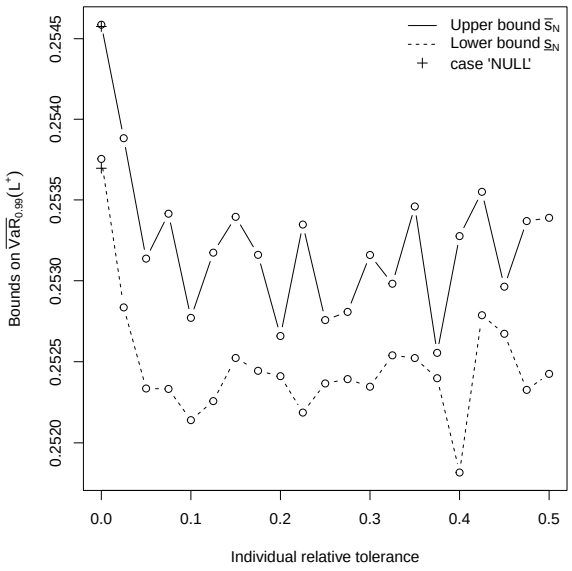

Figure 9. Bounds $\underline{s}_N$ and $\overline{s}_N$ on $\overline{VaR}_{0.99}(L^+)$ as functions of the chosen individual relative tolerance.

Example 11 (Bootstrap confidence interval for worst VaR).

As in Example 10 and mentioned in Section 2, the marginal quantile functions used as input for the rearrangement algorithm may need to be estimated from data. The corresponding estimation error translates to a confidence interval for $\overline{VaR}_{0.99}(L^+)$, which we can obtain from a bootstrap. To this end, we start by building B bootstrap samples from the negative log-returns; we choose a rather small $B = 20$ here to demonstrate the idea.

```
> B <- 20 # bootstrap sample size
> set.seed(271) # for reproducibility
> X.boot <- lapply(1:B, function(b) # B-list of (nrow(X), d)-matrices
+     apply(X, 2, function(x.) sample(x., replace = TRUE)))
```

Next, we build excess losses over 90% thresholds (a typical broad-brush choice in practice), fit GPDs to the d component samples for each of the B bootstrap samples and construct the corresponding quantile functions.

```
> u.prob <- 0.9 # threshold as probability
> stopifnot(alpha >= u.prob) # check validity condition
> qF.boot <- lapply(X.boot, function(x) { # x is a d-list of resampled -log-returns
+     lapply(1:d, function(j) { # going over all margins
+         u <- quantile(x[,j], probs = u.prob, names = FALSE) # threshold
+         excesses <- x[x[,j] > u, j] - u # excess losses
+         params <- fit_GPD_MLE(excesses)\$par # fitted GPD (shape, scale) vector
+         function(p) qGPDtail(p, threshold = u, p.exceed = 1 - u.prob,
+                              shape = params[["shape"]], scale = params[["scale"]])
+     })
+ }) # B-list of d-lists (one for each stock) of quantile functions
```

Now we can call `ARA()` with each of the B sets of marginal quantile functions.

```
> worst.VaR.boot <- lapply(qF.boot, function(qFs) ARA(alpha, qF = qFs)) #B-list of ARA() obj.
```

And then extract the computed estimates of $\overline{VaR}_{0.99}(L^+)$.

```
> worst.VaRs <- sapply(worst.VaR.boot, function(ara) mean(ara[["bounds"]]))
> summary(worst.VaRs) # a summary
```

```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.2215  0.2355  0.2417  0.2444  0.2522  0.2841
```

A bootstrap 95%-confidence interval for $\overline{VaR}_{0.99}(L^+)$ *is thus given as follows.*

```
> quantile(worst.VaRs, probs = c(0.025, 0.975))
```

```
2.5%     97.5%
0.2226960 0.2761207
```

Note that our run-time improvements of Section 4 are especially useful here as a bootstrap would otherwise be relatively time-consuming.

Example 12 (Basic worst VaR allocation).

Capital allocation concerns the allocation of a total capital K *to* d *business lines so that* $K = \mathrm{AC}_1 + \cdots + \mathrm{AC}_d$ *(the full allocation property), where* AC_j *denotes the capital allocated to business line* j*. As the RA uses the minimal row sum as* $\overline{VaR}_\alpha(L^+)$ *estimate, the rows in the final rearranged matrices* $\underline{Y}^\alpha, \overline{Y}^\alpha$ *provide an allocation of* $K = \overline{VaR}_\alpha(L^+)$ *one can call* worst VaR allocation*. There could be multiple rows in* $\underline{Y}^\alpha, \overline{Y}^\alpha$ *leading to the minimal row sum in which case the componentwise average of these rows is considered. The resulting two rows are returned by* `(rearrange()` *and)* `ARA()` *in* `qrmtools` *version* `0.0.13`*. By componentwise averaging, the latter two rows we obtain an approximate worst VaR allocation. Given that we already have a bootstrap sample, we can average the B computed allocations componentwise to obtain the bootstrap mean of the worst VaR allocation.*

```
> stopifnot(packageVersion("qrmtools") >= "0.0.13") # package version check
> worst.VaR.rows <- t(sapply(worst.VaR.boot, function(ara) { # (B, d)-matrix
+     opt.rows <- ara[["X.rearranged.opt.row"]] # 2-list (lower/upper bound) of d-vectors
+     rowMeans(matrix(unlist(opt.rows), ncol = 2)) # average over both bounds
+ }))
> colnames(worst.VaR.rows) <- stocks # name columns according to the stocks
> (worst.VaR.alloc <- colMeans(worst.VaR.rows)) # estimated worst VaR allocation
```

```
GOOGL       AAPL       MSFT
0.08067026 0.08612456 0.07756922
```

We can also compare the corresponding estimate of the total allocation $\mathrm{AC}_1 + \cdots + \mathrm{AC}_d$ *with the average of the B optimal row sums computed by the ARA to check the full allocation property; to this end, we first compute the B averages of the optimal row sums over the lower and upper bounds for each replication.*

```
> worst.VaR.row.sums <- sapply(worst.VaR.boot, function(ara) # B-vector
+     mean(sapply(ara[["opt.row.sums"]], function(v) tail(v, n = 1))))
> ave.row.sum <- mean(worst.VaR.row.sums) # average optimal row sum over B replications
> stopifnot(all.equal(sum(worst.VaR.alloc), ave.row.sum))
```

Bootstrap 95%-confidence intervals for each of $\mathrm{AC}_1, \ldots, \mathrm{AC}_d$ *can be obtained as follows.*

```
> apply(worst.VaR.rows, 2, quantile, probs = c(0.025, 0.975))
```

```
GOOGL       AAPL       MSFT
2.5%  0.06659040 0.07376124 0.06471199
97.5% 0.09593365 0.09654867 0.09201542
```

For other applications of the RA, see, for example, Embrechts and Jakobsons (2016), Bernard et al. (2017), Bernard et al. (2018) and, most notably, Ramsey and Goodwin (2019) where the RA is applied in the context of crop insurance.

7. Selected Lessons Learned

After using R to motivate column rearrangements and opposite ordering for finding the worst VaR for given marginal distributions in Section 2, we considered a basic implementation of such rearrangement in Section 3. We profiled the code in Section 4 and improved the implementation step-by-step by avoiding unnecessary summations, avoiding accessing matrix columns and improving the sorting step. In Section 5 we then used the implementations `rearrange()` and `ARA()` in our R package `qrmtools` to trace the rearrangement steps and to motivate the use of the default tolerances used by `ARA()`. A real data example was considered throughout Section 6 where we computed the worst VaR with `ARA()`, visualized a sample from the corresponding worst VaR copula and assessed the sensitivity of the computed worst VaR with respect to the individual convergence tolerance. To incorporate the uncertainty due to the estimation error of the marginal distributions, we used a bootstrap idea to obtain a confidence interval for the true worst VaR. We also introduced worst VaR allocation as a capital allocation principle and computed bootstrap confidence intervals for the allocated capitals.

Our experiments reveal lessons to learn that frequently appear in problems from the realm of computational risk management:

- We can and should use software to learn about new problems and communicate their solutions. Implementing an algorithm often helps to fully comprehend a problem, experiment with it and get new ideas for practical solutions. It also allows one to communicate a solution since an implementation is unambiguous even if a theoretical presentation or pseudo-code of an algorithm leaves questions unanswered (such as how to choose tuning parameters); the latter is often the case when algorithms are provided in academic research papers.
- Both for understanding a problem and for communicating its solution, it is paramount to use visualizations. Creating meaningful graphs is unfortunately still an exception rather than the rule in academic publications where tables are frequently used. Tables mainly allow one to see single numbers at a time, whereas graphs allow one to see the "bigger picture", so how results are connected and behave as functions of the investigated inputs.
- Given an algorithm, start with a basic, straightforward implementation and make sure it is valid, ideally by testing against known solutions in special cases; we omitted that part here, but note that there are semi-analytical solutions available for computing worst VaR in the case of homogeneous margins.
- Learn from the basic implementation, experiment with it and improve it, for example, by improving numerical stability, run time, maintenance, etc. If you run into problems along the way, use minimal working examples to solve them; they are typically constructed by divide and conquer. As implementations get more involved, compare them against the basic implementation or previously checked improved versions.
- When improving code, be aware of the issues mentioned in the introduction and exploit the fact that mathematically unique solutions often allow for different computational solutions. Also, computational problems (ties, parts of domains close to limits, etc.) can arise when there are none mathematically. These are often the hardest problems to solve, which is why it is important to be aware of computational risk management problems and their solutions. Implementing a computationally tractable solution is often one "dimension" more complicated than finding a mathematical solution—also literally, since a random variable (random vector) is often replaced by a vector (matrix) of realizations in a computational solution.

- Results in academic research papers are often based on an implementation of a newly presented algorithm that works only under specific conditions with parameters found by experimentation (often hard-coded) under such conditions in the examples considered. A publicly available implementation in a software package needs to go well beyond this point; for example, as users typically do not have expert knowledge of problems the implementation intends to solve and often come from different backgrounds altogether in the hope to find solutions to challenging problems in their field of application. On the one hand, the amount of work spent on development and maintenance of publicly available software solutions is largely underestimated. On the other hand, one can sometimes benefit from getting to know different areas of application or valuable input for further theoretical or computational investigation through user feedback. Also, ones own implementation often helps one to explore potential further improvements or new solutions altogether.
- Optimizing run time to be able to apply solutions in practical situations can be important but is not all there is. Providing a readable, comprehensible and numerically stable solution is equally important, for example. Code optimized solely for run time typically becomes less transparent and harder to maintain, is thus harder to adapt to future conceptual improvements and more prone to semantic errors. Also, a good solution typically not only provides a final answer but useful by-products and intermediate results computed along the way, just like a good solution in a mathematical exam requires intermediate steps to be presented in order to obtain full marks (which also simplifies to find the culprit if the final answer is erroneous).

Funding: This research was funded by NSERC under Discovery Grant RGPIN-5010-2015.

Acknowledgments: The author would like to thank Kurt Hornik (Wirtschaftsuniversität Wien) and Takaaki Koike (University of Waterloo) for valuable feedback and inspiring discussions.

Conflicts of Interest: The author declares no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ARA	adaptive rearrangement algorithm
RA	rearrangement algorithm
VaR	value-at-risk

References

Bernard, Carole, Michel Denuit, and Steven Vanduffel. 2018. Measuring portfolio risk under partial dependence information. *The Journal of Risk and Insurance* 85: 843–63. [CrossRef]

Bernard, Carole, Ludger Rüschendorf, Steven Vanduffel, and Jing Yao. 2017. How robust is the value-at-risk of credit risk portfolios? *The European Journal of Finance* 23: 507–34. [CrossRef]

Embrechts, Paul, and Edgars Jakobsons. 2016. Dependence uncertainty for aggregate risk: Examples and simple bounds. In *The Fascination of Probability, Statistics and their Applications*. Edited by Mark Podolskij, Robert Stelzer, Steen Thorbjørnsen and Almut E. D. Veraart. Cham: Springer.

Embrechts, Paul, Giovanni Puccetti, and Ludger Rüschendorf. 2013. Model uncertainty and var aggregation. *Journal of Banking and Finance 37*: 2750–64. [CrossRef]

Haus, Utz-Uwe. 2015. Bounding stochastic dependence, joint mixability of matrices, and multidimensional bottleneck assignment problems. *Operations Research Letters* 43: 74–79. [CrossRef]

Hofert, Marius. 2020. Random Number Generators Produce Ties: Why and How Many. Available online: https://arxiv.org/abs/2003.08009 (accessed on 15 April 2020).

Hofert, Marius, Rüdiger Frey, and Alexander J. McNeil. 2020. *The Quantitative Risk Management Exercise Book*. Princeton: Princeton University Press, ISBN 9780691206707. Available online: https://assets.press.princeton.edu/releases/McNeil_Quantitative_Risk_Management_Exercise_Book.pdf (accessed on 31 March 2020).

Hofert, Marius, Amir Memartoluie, David Saunders, and Tony Wirjanto. 2017. Improved algorithms for computing worst Value-at-Risk. *Statistics & Risk Modeling* 34: 13–31. [CrossRef]

McNeil, Alexander J., Rüdiger Frey, and Paul Embrechts. 2015. *Quantitative Risk Management: Concepts, Techniques, Tools*, 2nd ed. Princeton: Princeton University Press.

Ramsey, A. Ford, and Barry K. Goodwin. 2019. Value-at-risk and models of dependence in the u.s. federal crop insurance program. *Journal of Risk and Financial Management* 12: 65. [CrossRef]