Taking R to its limits: 70+ tips

Michail Tsagris[1] **and Manos Papadakis**[2]

[1]**University of Crete, Herakleion, Greece**
[2]**University of Crete, Herakleion, Greece**

Corresponding author:
Michail Tsagris[1]

Email address: mtsagris@yahoo.gr

ABSTRACT

R has many capabilities most of which are not known by many users, yet waiting to be discovered. For this reason we provide more tips on how to write really efficient code without having to program in C++, programming advices, and tips to avoid errors and numerical overflows.

INTRODUCTION

Efficient code is really important in large scale datasets, computations, or simulation studies. R may not be a scripting language or computationally efficient when compared to other open source programs, yet it has strong capabilities not known however to most of its users.

We will show a few tips for faster computations. The speed-ups one will in small sample and or low dimensions will be small, but in bigger datasets the differences are observable. One might observe a time difference of only 5 seconds in the whole process. Differences from 40 seconds to 12 seconds for example, or to 22 seconds are still worthy. But not always this kinds of differences will be experienced. Some times, one tip gives 1 second and then another tip 1 second and so on until you save 5 seconds. If a researcher has to run 1000 simulations, then they can save 5000 seconds. Our moto is that every decrease in time or memory matters.

Some times the speed-ups will appear with large samples only. To put it simply, for someone who needs a simple car, a very expensive car or a jeep type might not be of such use, especially if they do not go off-road. But for the user who needs a jeep, every computational power, every second they can gain matters.

We have listed 70+ tips that can help the user write more efficient code, use less memory, become more familiar with R and its commands. The list is not exhaustive and we are sure there are tips we are not aware of. However, this is the first time such a long list is available.

1 DURATION OF A PROCESSES

In order to measure the time your process or computation or simulation needs, one can do the following in R

```
runtime <- proc.time()
## put your function here
runtime <- proc.time() - runtime
## runtime gives you 3 numbers (all in seconds) like the ones below
user  system elapsed
0.18    0.07    3.35
```

The elapsed is the desired metric. Alternatively the package microbenchmark (Mersmann, 2015) which allows for comparison between two or more functions measuring the execution time even in the nanoseconds scale.

2 GENERAL ADVICE

We begin with a list of of programming advices.

1. Avoid unnecessary calculations. In a discriminant analysis setting for example there is no need to calculate constant parts, such as $\log(2\pi)$, every time for each group and every iteration. This only adds time and takes memory and does not affect the algorithm or the result. The same for the Euclidean or the Hellinger distance. The square root and the multiplication by the same factor everywhere makes no difference and if can be avoided, then it should be avoided.

2. Try to make the mathematics and hence the operations as simple as possible. The partial correlation coefficient can be computed with many ways. Two of them are via two regression models, or via the simple correlation matrix. The second one is much faster.

3. If you have a function which uses many regression models, not necessarily from different downloadable R packages, but even from R's built-in models you should not to create a function for each model. This seems like the obvious thing to do, simply because different regression models have different signatures and hypothesis test with each of them is different, *lm* uses F test, *glm* use F or χ^2 test and so on.

   The optimal solution is to write one function which includes all regression models and their arguments. This way, you have one inclusive function for all models, instead of many. Every time you want to add a new regression model you add it in this function and do not have to create a new function. This way, you have less files and require less memory.

4. When writing code, leave space between operations, parentheses, indent the lines inside the *for* loops. Put comments at the end of every loop so as to know to which loop each bracket refers to. This is really helpful if your code is long. Make your code more readable and *clean*. The goal is not only to make it readable by someone else, but mainly by yourselves. Clean code is easier to understand and edit. A good programmer makes mistakes, but detects them fast. If your code is long, has no spaces and no comments, spotting any mistakes can become a difficult task.

5. Understand how R works and what operations do, what you can or are not allowed to do. The command *mahalanobis* for example uses this line to calculate the distances.

   ```
   rowSums(x %*% cov * x)
   ```

   The first key message is that the function performs one and not two (as one would expect) matrix multiplications. Secondly, the user is given the chance to see that he can apply these types of operations. Thirdly, he is given the chance to make this function faster.

6. Do not use object oriented programming (S3,S4 methods) . It is not necessary and makes the hole program slow but if it is necessary then try make you own object using environments.

7. If you have a vector x with integer numbers, it is preferable to have it *as.integer(x)* than *as.numeric(x)*. The former requires less memory than the latter.

3 SIMPLE FUNCTIONS

We continue with a list of simple functions which will help you make your code more efficient and faster.

1. Do not expect that all packages or R itself have fast implementations. The function *mean* is slower than *sum(x)/length(x)*. If you type *sum* you will see it is a *.Primitive* function whereas *crossprod*for example is an *.Internal* function. Another example of a more advanced function is to perform many univariate logistic regressions and each time calculate the deviance of the model. Create your own functions, you will be surprised to see that you may do faster than R's built-in functions (it doesn't always work that way).

2. The commands *cor* and *cov* are two very fast functions which can be used to calculate the correlations or covariances between a vector and a matrix very efficiently. Many simple regression models can be calculated using these two commands.

3. Search for functions that take less time. For example, the command *lm.fit(x,y)* is a wrapper for *lm(y x)*, which means that the former is used by the latter. If you need only the coefficients, for example, then use the first one. The syntax is a bit different, *x* must be the design matrix, and the speed is also very different, especially in the big cases. The command *.lm.fit(x,y)* is even faster.

4. It's your algorithm. In a recent paper, Tsagris (2017) showed that conditional G^2 tests of independence can become 3-4 faster if they are run using Poisson log-linear models and not by constructing the appropriate tables using *for* loops. There are may ways to perform your computations. Choose or implement the most appropriate and most efficient. For example, in order to generate vectors from the multivariate normal distribution you can either use spectral or Cholesky decomposition. The latter is faster. In general, try to find the optimal mathematical method or way for your problem.

5. Suppose you want to center some data. You can try with *apply*

```
cent <- function(x) x - mean(x)
a1 <- apply(data, 2, cent)
```

   or use one of these

```
a2 <- scale(data, center = TRUE, scale = FALSE)
m <- colMeans(data)
a3 <- sweep(data, 2L, m)
a4 <- t( t(data) - m ) ## this is from the previous tip
dm <- dim(data)
n <- dm[1]
p <- dm[2]
a5 <- data - rep( m, rep(n, p) ) ## looks faster
```

6. If you want to extract the mean vector of each group you can use a loop (*for* function) or

```
a1 <- aggregate(x, by = list(ina), mean)
```

   where *ina* is a numerical variable indicating the group. A faster alternative is the built-in command *rowsum*

```
a2 <- rowsum(x, ina) / as.vector( table(ina) )
## a bit faster option is a3 below
a3 <- rowsum(x, ina, reorder = FALSE) / as.vector( table(ina) )
```

7. Use *which.min(x)* and *which.max(x)* instead of *which(x == min(x)) which(x == max(x))* to find the position of the minimum and maximum number respectively.

```
x <- array( dim = c(1000,10,10) )
for (i in 1:10) x[, , i] = matrix( rnorm(1000* 10), ncol = 10 )
a1 <- apply(x, 1:2, mean)
a2 <- t( colMeans( aperm(x) ) )
```

8. Incremental calculations. An example of this is the first order partial correlations, which can be computed from the simple correlations with fewer calculations. In general, if your function can perform calculations incrementally, the boost in the speed can be very high.

9. If you have a vector x and want to put it in a matrix with say 10 columns, do not write *as.matrix(x, ncol = 10)*, but *matrix(x, ncol = 10)*. The first method creates a matrix and puts the vector in. The second method, simply changes the dimension of *x*, instead of 1 column, it will now have 10. Again, about 2 times faster.

10. When it comes to calculating probabilities or p-values more specifically, do not do $1 - pchisq(stat, dof)$, but do $pchisq(stat, dof, lower.tail = FALSE)$ as is a bit faster. In the tens of thousands of repetitions (simulation studies for example or an algorithm that requires p-values repeatedly), the differences become seconds.

11. When calculating operations such as *sum(a * x)*, where **x** is a vector or a matrix and *a* is a scalar (number) do *a * sum(x)*. In the first case, the scalar is multiplied with all elements of the vector (many multiplications), whereas in the second case, the sum is calculated first and then a multiplication between two numbers take place.

12. Suppose you want to calculate the factorial of some integers and most (or all) of those integers appear more than once (Poisson, beta binomial, beta geometric, negative binomial distribution for example). Instead of doing the operation for each element, do it for the unique ones and simply calculate its result by its frequency. See the example below. **Note however**, that this trick does not always work. It will work in the case where you have many integers and a *for* or a *while* loop and hence you have to calculate factorials all the time.

    ```
    x <- rpois(10000, 5)
    sum( lgamma(x + 1) )
    y <- sort( unique(x) )
    ny <- as.vector( table(x) )
    sum( lgamma(y + 1) * ny )
    ```

13. If you use the *glm* or *lm* commands multiple times, then you should do

    ```
    glm(y ~ x, family = ..., y = FALSE, model = FALSE)
    lm(y ~ x, y = FALSE, model = FALSE)
    ```

    The two extra arguments *y = FALSE ,model = FALSE* reduce the memory requirements of the glm object.

14. When calculating $\log(1+x)$ use *log1p(x)* and not *log(1+x)* as the first one is faster.

15. A very useful command is *tabulate*.

    ```
    table(iris[, 5])
    tabulate(iris[, 5])
    ```

    Two differences between these two are that table gives you a name with the values, but tabulate gives your only the frequencies. Hence, *tabulate(x) = as.vector(table(x))*. In addition, if you use *tabulate*, you can do so with factor variables as well. But, if you have numbers, a numerical vector, make sure the numbers are consecutive, and strictly positive, i.e. no zero is included.

    ```
    x <- rep(0:5, each = 4)
    table(x)
    tabulate(x)  ## 0 is missing
    x <- rep(c(1, 3, 4), each = 5)
    table(x)
    tabulate(x) ## there is a 0 appearing indicating the absence of 2
    ```

    The command *tabulate* is many times faster than *table*. For discriminant analysis algorithms, *tabulate* might be more useful, because of speed, when counting frequencies, it could be more useful as well, as it will return a 0 value if a number has a zero frequency. The drawback arises when you have negative numerical data, data with a zero or positive numbers but not consecutive. If you want speed, formulate your data to match the requirements of *tabulate*.

16. Vectorization can save tremendous amount of time even in the small datasets. Try to avoid *for* loops by using matrix multiplications. For example, instead of

    ```
    for (i in 1:n) y[i] <- x[i]^2
    ```

    you can use

    ```
    y <- x^2
    ```

    Of course, this is a very easy example, but our point is made. This one requires a lot of thinking and is not always applicable. But, if it can be done, things can be substantially faster.

17. Make use of the command *outer*. An example is where you have two vectors x and y and you want to add each element of y in x. The final item would be a matrix.

    ```
    z <- matrix(0, nrow = length(x), ncol = length(y) )
    for ( i in 1:dim(z)[1] )  z[i, ] <- x[i] + y
    ```

```
## The above task can take place a lot faster by typing
outer(x, y, "+")
```

18. The command *sort* has an extra feature, by specifying *index.return = TRUE*, the outcome is a list with the sorted numbers, along with their ordered indexes. That is, you call *sort(x, index.return = TRUE)* and the command sorts the numbers and returns their order as if you used *order* separately.

19. You can partially sort a vector. In the k-NN algorithm for example you want the k smallest distances. There is no need to sort all the distances, only the k smallest. Hence, *sort(x, partial=1:k)*.

20. When using *sort* select type *sort(x, method=quick)* in order to make it even faster, in most cases.

21. Many functions call internal function inside. *sort* for example calls *sort.int*, *sample* calls *sample.int* etc.

22. You are given many matrices in a list, ”A”, and wish to unlist all of them and create a new matrix, where the matrixc of each element is added one under the other. The command to do this efficiently is

```
do.call(rbind, A)
```

4 CALCULATIONS INVOLVING MATRICES

R is not designed to handle large scale datasets, yet there are many ways to efficiently handle matrices and we present to you some tricks below.

1. Use *colMeans* and *rowMeans* instead of *apply(x, 1, mean)* and *apply(x, 2, mean)* as they are extremely fast. In addition, many really fast functions can be produced using these two commands.

2. Avoid using *apply* or *aggregate* whenever possible. For example, use *colMeans* or *colSums* instead of *apply(x, 2, mean)* to get the mean vector of a sample because it’s faster. For the median though, you have to use *apply(x, 2, median)* instead of a *for* going to every column of the matrix. The *for* loop is not slower, but the *apply* is knitter.

3. If you have to use a *for* loop going through the rows of a matrix, consider transposing the matrix and then go through its columns.

4. If you are given a matrix and by using a *for* loop you would like to extract some specific columns/rows of the matrix each time. Instead of that, you can store the indices of the columns/rows inside the *for* loop and outside simply extract the columns/rows and perform any operations you want.

5. If you want to extract the number of rows or columns of a matrix **X**, do not use *nrow(X)* or *ncol(X)*, but *dim(X)[1]* or *dim(X)[2]* as they are almost 2 times faster.

6. Suppose you have a matrix **X** and you want the position of the maximum for each row. The obvious solution is *apply(X, 1, which.max)*. The efficient solution is *max.col(X)*.

7. If you want to subtract a vector from a matrix in a row-wise fashion, you should be aware of the fact that R does it column-wise. This is because R reads, writes and stores data in a column-wise fashion. For example, **X** is a matrix and **y** is a vector whose length is equal to the number of columns of **X**. You should type

```
t(X) - y
```

8. If you take your input matrix and transpose it and never use the initial matrix in the subsequent steps it is best to delete the initial matrix, or even better store its transpose in the same object. That is, if you have a matrix **X**, you should do the following

```
Y <- t(X)  ## not suggested
X <- t(X)  ## suggested
```

We repeat that this in the case when x is not used again in latter steps. The reason for this is memory saving. If x is a big and you have a second object as big as the first one, you request your computer to use extra memory with no reason.

9. Use the command *prcomp* instead of *princomp*. The first one should be used for principal component analysis when you have matrices with more than 100 variables. The more variables the bigger the difference (40 times for example) from using *eigen(cov(x))*.

10. Create the vectors or matrices from the start. Instead of making a vector longer each time, using *c()* create an empty vector with the required size. The commands *rbind* and *cbind* are useful, but come with a heavy price. They are very expensive when called a lot of times.

11. For the covariance matrices the command *by* could be used. The matrices are stored in a list and then you need the command *simplify2array* to convert the list to an array in order to calculate for example the determinant of each matrix. The *for* loop is faster, at least that's what we have seen in our trials.

12. What if you have an array with matrices and want to calculate the sum or the mean of all the matrices? The obvious answer is to use *apply(x, 1:2, mean)*. R works in a column-wise fashion and not in a row-wise fashion. Instead of the *apply* you can try *t(colSums(aperm(x)))* and *t(colMeans(aperm(x)))* for the *sum* and *mean* operations respectively.

13. If you want to calculate the logarithm of the determinant of a matrix $\mathbf{X}$, instead of *log(det(X))*, you can type *determinant(X, logarithm = TRUE)* as it is slightly faster for small matrices. In the big matrices, say of dimensions 100×100 or more, the differences become negligible though.

14. If you want the matrix of distances, with the zeros in the diagonal and the upper triangular do not use the command *as.matrix(dist(x))* but use *dist(x, diag = TRUE, upper = TRUE)*.

15. Suppose you want the Euclidean distance of a single vector from many others (say thousands for example). The inefficient way is to calculate the distance matrix of all points and take the row which corresponds to your vector. The efficient way is to use the Mahalanobis distance with the identity matrix and the covariance matrix.

```
x <- rnorm(50)
y <- matrix( rnorm(1000 * 50), ncol = 50) )
a1 <- dist( rbind(x, y) )  ## inefficient way
Ip <- diag(50)
## a2 is a better way
a2 <- mahalanobis( y, center = x, cov = Ip, inverted = TRUE )
```

Another way is the following

```
z <- y - x
a <- sqrt( colSums(z^2) )
```

16. Calculating $\mathbf{X}^T\mathbf{Y}$ in R as $t(X)\% * \%Y$ instead of *crossprod(X, Y)* causes X to be transposed twice; once in the calculation of $t(X)$ and a second time in the inner loop of the matrix product. The *crossprod* function does not do any transposition of matrices.

17. If you want to calculate the product of an $n \times p$ matrix $\mathbf{X}^T\mathbf{X}$ for example. The command *crossprod(X)* will do the job faster than the matrix multiplication.

```
t(X) %*% Y      ## classical
crossprod(X, Y)  ## more efficient
X %*% t(Y)    ## classical
tcrossprod(X, Y)    ## more efficient
```

```
t(X) %*% X     ## classical
crossprod(X)    ## more efficient
```

18. Let **X** and **m** be a matrix and a vector and want to multiply them. There are two ways to do it.

```
sum(m * x)
sum(x %*% m)   ## a bit faster
```

19. When working with arrays it is more efficient to have them transposed. For example, if you have K covariance matrices of dimension $p \times p$, you would create an array of dimensions $c(p, p, K)$. Make its dimensions $c(K, p, p)$. If you want for example to divide each matrix with a different scalar (number) in the first case you will have to use a for loop, whereas in the transposed case you just divide the array by the vector of the numbers you have.

20. If you want to only invert a positive definite matrix (e.g. covariance matrix) then you should use *chol2inv(chol(X))* as it is faster.

21. To invert a matrix (not necessarily positive definite) and multiply the result with a vector or another matrix, there are two ways to do that

```
solve(X) %*% Y  ## classical
## a much more efficient way is not
## to invert the matrix X
solve(X, Y)
```

22. The trace of the square of a matrix tr $\left(\mathbf{X}^2\right)$ can be evaluated either via

```
sum( diag( crossprod(X) ) )
```

or faster via

```
sum(X * X) ## or
sum(X^2)
```

23. If you want to calculate the following trace involving a matrix multiplication tr $\left(\mathbf{X}^T\mathbf{Y}\right)$ you can do either

```
sum( diag( crossprod(X, Y) ) )  ## just like before
```

or faster

```
sum(X * Y)  ## faster, like before
```

24. Moving in the same spirit, suppose you want the diagonal of the crossproduct of two matrices, then do

```
diag( tcrossprod(X, Y) )  ## for example
rowSums(X * Y)  ## this is faster
```

25. Suppose you have two matrices **A**, **B** and a vector **x** and want to find **ABx** (the dimensions must match of course).

```
A %*% B %*% x  ## inefficient way
A %*% (B %*% x) ## efficient way
```

The explanation for this one is that in the first case you have a matrix by matrix by vector calculations. In the second case you have a matrix by vector which is a vector and then a matrix by a vector. You do less calculations. The final tip is to avoid unnecessary and/or extra calculations and try to avoid doing calculations more than once.

26. As for the eigen-value decomposition, there are two ways to do the multiplication

```
s <- matrix( rnorm(100 * 100), ncol = 100 )
s <- crossprod(s)
eig <- eigen(s)
vec <- eig$vectors
lam <- eig$values
a1 <- vec %*% diag(lam) %*%t(vec)
a2 <- vec %*% ( t(vec) * lam )  ## faster way
```

27. The exponential term in the multivariate normal can be either calculated using matrices or simply with the command *mahalanobis*. If you have many observations and many dimensions and or many groups, this can save you a **lot** of time.

```
x <- matrix( rnorm(1000 * 20), ncol = 20 )
m <- colMeans(x)
n <- nrow(x)
p <- ncol(x)
s <- cov(x)
a1 <- diag( (x - rep(m, rep(n, p)) ) %*% solve(s)
%*% t(x - rep(m, rep(n, p)) ) )
a2 <- diag( t( t(x)- m ) %*% solve(s) %*% t(x)- m )
a3 <- mahalanobis(x, m, s)  ## much faster
```

5 NUMERICAL OPTIMIZATION

1. The command *nlm* is much faster than *optim* for optimization purposes but *optim* is more reliable and robust. Try in your examples or cases, if they give the same results and choose. Or, use *nlm* followed by *optim*.

2. If you have a function for which some parameters have to be positive, do not use constrained optimization, but instead put an exponential inside the function. The parameter can take any values in the whole of *R* but inside the function its exponentiated form is used. In the end, simply take the exponential of the returned value. As for its variance use the δ-method (Casella and Berger, 2002). The trick is to use a link function, similarly to generalised linear models.

3. There are two ways to estimate the parameters of a distribution, Dirichlet for example. Either with the use *nlm* or via the Newton-Raphson algorithm. We performed some simulations and saw that the Newton-Raphson can be at least 10 times faster. The same is true for the circular regression (Presnell et al., 1998) when comparing *nlm* with the E-M algorithm as described by Presnell et al. (1998). Switching to E-M or the Newton-Raphson and not relying on *nlm* can save you a lot of time. If you want to write code and you have the description of the E-M or the Newton-Raphson algorithm available, then do it. Among these two, Newton-Raphson is faster.

4. If you have an iterative algorithm, such as Newton-Raphson, E-M or fixed points and you stop when the vector of parameters does not change any further, do not use *rbind*, *cbind* or *c()*. Store only two values, *vec.old* and *vec.new*. What we mean is, do not do for example

```
u[i, ] <- u[i - 1, ] + W%*%B  ## not efficient
u.new <- u.old + W%*%B  ## efficient
```

   So, every time keep two vectors only, not the whole sequence of vectors. The same is true for the log-likelihood or the criterion of interest. Unless you want to keep track of how things change our advice is to keep two values only, the current and the previous one. Otherwise, apart from being faster, it also helps the computer run faster since less memory is used.

6 NUMERICAL OVERFLOWS

Numerical instabilities occur frequently when using real, not simulated, data. The reason why we mention these tricks is because one should not sacrifice numerical issues for the shake of speed. When trying to speed-up code, we experienced numerical instabilities and overflows and we would like to share some of the issues we faced.

1. The fitted values in a logistic regression model make use of the inverse of the logit function $\frac{e^x}{1+e^x}$. If x is a really high number, the numerator becomes infinity (*Inf*). If however you write this formula in its equivalent form $\frac{1}{1+e^{-x}}$, then the result is 1 as it should be.

2. The same is true for the Poisson regression or other regression which use the log as their link function. In those cases one uses e^x. Try using e^{-x} instead.

3. Kernel functions are of the form $e^{\frac{f(\mathbf{X})}{h}}$, where h is a scalar and $\mathbf{X}$ is a matrix. In order to speed up the calculations one could pre-calculate $e^{f(\mathbf{X})}$ and then raise the result to the power of the different values of h. This can easily result in overflow and *Inf* values. Speed-up can lead to overflow, hence caution must be taken.

4. By Taylor series we know that $\log(1+e^x) \simeq x$, but when $x \geq 13$ the two terms are equal $\log(1+e^x) = x$. Even the command *log1p* will not avoid the *Inf* result.

5. The product $\prod_{i=1}^n x_i$, where $0 < x_i \leq 1$, becomes 0 very fast. In order to avoid this, you should use $e^{\sum_{i=1}^n \log x_i}$.

6. Use the logarithm of the p-values and not the p-values. This can be very beneficial when your algorithm calculates and compares p-values. When the p-value is smaller than the 2.220446e-16 (*.Machine$double.eps*) are rounded to zero. R cannot sort, correctly, p-values less than that number because they are all considered equal to zero. If you have requested the logarithm of the p-values though, those negative numbers can be sorted correctly.

7 PARALLEL COMPUTING IN R

If you have a machine that has more than 1 cores, then you can put them all to work simultaneously and speed up the process a lot. If you have tricks to speed up your code that is also beneficiary. We have started taking into account tricks to speed up my code as we have mentioned before.

The idea behind is to use a library that allows parallel computing. We make use of the doParallel package (which uses the *foreach* package). Below are some instructions on how to use the package in order to perform parallel computing. In addition, we have included the parallel computing as an option in some functions and in some others we have created another function for this purpose. So, if you do not understand the notes below, you can always see the functions in R packages that use this package.

```
## requires(doParallel)
Create a set of copies of R running in parallel and communicating
## over sockets.
cl <- makePSOCKcluster(nc) ## nc is the number of cluster you
## want to use
registerDoParallel(cl) ## register the parallel backend with the
## foreach package.
## Now suppose you want to run R simulations, could be
## R <- 1000 for example
## Divide the number of simulations to smaller equally
## divided chunks.
## Each chunk for a core.
ba <- round( rep(R/nc, nc) )
## Then each core will receive a chunk of simulations
ww <- foreach(j = 1:nc,.combine = rbind) %dopar% {
## see the .combine = rbind. This will put the results in a matrix.
## Every results will be saved in a row.
## So if you have matrices, make them vectors. If you have lists
## you want to return,
## you have to think about it.
```

```
  a <- test(arguments, R = ba[j], arguments)$results
  ## Instead of running your function "test" with R simulations
  ## you run it with R/nc simulations.
  ## So a stores the result of every chunk of simulations.
  return(a)
   }
 stopCluster(cl) ## stop the cluster of the connections.
```

To see your outcome all you have to press is *ww* and you will see something like this

```
result.1 .....
result.2 .....
result.3 .....
result.4 .....
```

The object *ww* contains the results you want to see in a matrix form. If every time you want a number, the *ww* will be a matrix with 1 column. We will see more cases later on. Note that f you choose to use parallel computing for something simple, multicore analysis might take the same or a bit more time than single core analysis only because it requires a couple of seconds to set up the cluster of the cores. In addition, you might use 4 cores, yet the time is half than when 1 core is used. This could be because not all 4 cores work at 100%.

In this example we have given each chunk of simulations to a core. Alternatively, one can have a *for* loop going through all columns of a matrix, for example. The latter is slower, but the former is not always possible.

8 EFFICIENTLY WRITTEN FUNCTIONS IN R PACKAGES

The multinomial regression is offered in the package VGAM (Yee, 2010), but it also offered in the package nnet (Venables and Ripley, 2002). The implementation in the second package is much faster. The same is true for the implementation of the ordinal logistic regression in the VGAM and in the ordinal (Christensen, 2015). The latter package does it much faster. Also, the package *fields* (Nychka et al., 2015) has a function called *rdist* which is faster than the built-in *dist* in R. These are just some examples where functions are available in more than one packages, but do not share the same computational cost.

Regression models, commands calculate distance matrices matrix, statistical tests, utility functions and many more can also be found in the package Rfast (Papadakis et al., 2017). This package contains many fast or really fast functions, either written in C++ or simply using R functions exploiting fast built-in functions. *colMedians* for example is much faster than *apply(x, 2, median)*. The same is true for the *colVars*. Functions for matrices, distribution fitting, utility functions and many more are there and we keep adding functions. We have also implemented regression functions as well, which can handle large sample sizes (50,000 or more, for example) efficiently. All codes are accessible in the .R or .cpp source files of the package.

9 MORE ADVANCED PROGRAMMING TIPS

There are programming languages that offer you extravagant features. We will mention some of those who support R, as well as their strengths and benefits.

1. **Operator overloading.**

   General: The overloading of operators is something incredibly handy and at the same time somehow dangerous. It allows you to manage your variables in the specific way that you declare. C++ supports overloading operators for a large number of operators, while others forbid it. R supports overload for a fairly large total (less than C ++ since it contains fewer operators) of operators:

   (a) brackets: [ ] (see in the special operators)

   (b) double brackets: [[ ]] (see in the special operators)

(c) binary and unary operators:

```
+, -, *, /, |, &, ^, %%, ==  (always 2 arguments)
```

(d) unary operators: (always 1 argument)

Be careful: Overloading is not supported for

```
&&, ||, =, <-, ->, <<-, ->>.
```

Although there is a way of overloading some of these operators that can be used together in a command, e.g. [ ] and <- , [[ ]] and <-.

The only drawback is that each operator is handled by a different set of arguments. However, there are no other drawbacks to R since all the above-mentioned operators (overloaded operators) are essential functions in relation to C ++ that if you use overloading then the operators converted into functions (where you pay a penny as long as functions are called) while no surplus treatments are being handled in a completely different way. The advantage (for both R and C++) are: a) new ways of implementation, b) control of variables for error management (as does R in some cases) and c) perhaps even higher speed (depends on the algorithm).

Implementation of overloading operators: The algorithm for the implementation of the overloading is the same for all overloaded operators (with some exceptions).

(a) Declare a variable (e.g. ”x”).

(b) Declare a variable with character value which represents anything you want.e.g. new_class = ”anything”. This is the 1st important step for oveloading operators.

(c) Change class for the variable from step (a): class(x) <- new_class This is the 2nd important step for oveloading operators.

(d) Declare a function for the operator you want to overload. A general way is, let the operator be the ”oper” which get values from the overloaded operators that R supports: ”t.new_class” <- function(whatever arguments the operator want) a) Set class to null for its arguments. b) Execute the operator. c) Set class to new_class_name. *Be careful:* the function name is very important.

(e) Finished with the creation.

(f) Press: x + x

Special operators [ ] and [[ ]]: The operators [ ], [[ ]] are used for access in some elements of a variable. There are 2 ways to use them, to extract an element and to import.

Extract:

```
[]:   "[.new_class" <- function(x,i,...) {}
[[]]: "[[.new_class" <- function(x,i,...) {}
```

Import:

```
[]<- :   "[<-.new_class" <- function(x, i, ..., value) { }
[[]]<- : "[[<-.new_class" <- function(x, i, ..., value) { return(x) }
```

Be careful: You don’t have to write different functions for operators <-, =, ->. You need only to use one of the three and R will understand the rest. Also you have to add one more argument for the import function because R uses it for the value to be stored. Finally, always return the argument ”x”.
E.g.

```
## OK
[]<- :  "[<-.new_class" <- function(x, i, ..., v) { return(x) }

## will produce error
[]<- :  "[<-.new_class" <- function(x, i) { return(x) }

## still OK but a bit risky
[]<- :  "[<-.new_class" <- function(x, i, ...) { return(x) }

## for us, preferred way if you want to access more than one cells
[]<- :  "[<-.new_class" <- function(x, ..., v) { return(x) }
```

Examples: Let's say we want to overload operator "+" for our own class.

(a) Declare a variable: x ¡- rnorm(10).

(b) Change class: class(x) ¡- "test_add".

(c) Create function.

```
"+.test_add" <- function(x, y) {
  class(x) <- class(y) <- NULL
  tmp <- x + y
  class(x) <- class(y) <- "test_add"
  tmp
}
```

(d) Create function for extract.

```
"[.test_add" <- function(x, ...){
  indices <- c(...)
  if (any (indices>length(x) ) ) {
    stop("Out of bounds in function '[]'")
  }
  class(x) <- NULL
  tmp <- x[indices]
  class(x) <- "test_add"
  tmp
}
```

(e) Create functions for import.

```
"[<-.test_add"<-function(x,...,v){
  indices <- c(...)
  if ( any(indices>length(x) ) ) {
    stop("Out of bounds in function '[]<-'")
  }
  class(x) <- NULL
  x[indices] <- v
  class(x) <- "test_add"
  x  ## neccessary step for R itself
}
```

2. Auto-printing your own class with your own style. In R you can print a variable using the functions "print","cat" or just type the variables names and press the enter key. But what happened if you don't like the way that R treats the printing? that is the point of auto-printing. R support a general way to print your own class using its default function "print". The method is very simple but with one exception, you must change the class as we do in the 1st programming advise.

```
print.new_class_name(x, other_arguments_that_rs_print_passes) {
  cat("auto-printing my variable with class 'new_class_name'
  and value: ", x)
}
```

Example:

(a) Create a variable: x¡-0

(b) Change its class: class(x) ¡- ”anything”

(c) create the function to print the variable:

```
print.anything(x, other_arguments_that_rs_print_passes) {
  cat("Auto-printing variable of class anything: ", x)
}
```

(d) Press: print(x)

(e) That’s it.

3. Using if-else in one line. Another feature in C++ is the ternary operator (which is not overloaded). General is anif-else statement with different syntax,sometimes more fast than if-else and also with one more capability, it returns the last command. That means the ternary operator is an if-else assign syntax. R support it also in the default if-else syntax. This can reduce the length of the code. Example:

```
if (is true) {
        x <- 1
} else {
      x <- 2
}
```

The above example is very simple but it needs 5 lines to be written. Lets reduce it:

```
x <- if (is true)  1  else  2
```

In a single line we have the same code. *Be careful* with the syntax and the last returned statement to be the one you want to initialize the variable ”x”. For more than one commands use curly brackets. This works fine because of the if-else. The *if-else* in R is like a function which always return the last command if you write it with the way that functions uses to return a variable. Not using the ”return” keyword, but just write the variable. *Note:* you can do the same exactly thing with *if-else-if* but do not forget to careful with the syntax.

4. Iterate vector. If you want to iterate a vector and use the variables for something you can do:

```
x < -rnorm(10)
for (i in 1:length(x) ) {
  cat(x[i])
}
```

This is classic. Another way to iterate through vector is:

```
x <- rnorm(10)
for ( i in x) {
  cat(i)
}
```

The second way is the same with the first one but instead of using the indices for the vector, then you use the vector itself. With the second way you eliminate one more action for each element of ”x” in *for* loop. This means that the total eliminations is ”length(x)”. So, you can decrease your speed, not much but, satisfactorily.

5. Efficient implementation of R's factor using any build-in type. *Factor* is a clever implementation of an integer vector. It has for values integers that are indices to a character vector with values the initial that user gives. It uses low memory because a character value needs memory equal to its length but loses in speed because you have to convert from the initial type to character. This means that if someone wants to extract a value and convert to its real type then it is slow. Fortunately the build in ¡as¿ functions are quite fast with one exception, the function ”as.character” needs a lot of execution time for convert each number. It is reasonable but imagine a large data set. Also factor is a read-only vector (by default in R). Lets see the code:

   (a) We are going to use an environment which is a reference struct so R will never copy it.

```
uf <- new.env()
## with this step we can apply all the above tricks
class(uf) <- "ufactor"
```

   (b) Create local variables inside ”uf” and a wrapper function:

```
ufactor <- function(x) {
  un <- sort( unique(x), method = "quick" )
  uf$values <- match(x, un)
  uf$levels <- un
  lockBinding("values", uf)
  lockBinding("levels", uf)
  lockEnvironment(uf)
  x
}
```

   (c) Create function extract element extraction:

```
"[.ufactor" <- function(uf, i) {
  uf$levels[ uf$values[i] ]
}
```

   Be careful: we don't need a function for import element because R doens't support it. We want our variable to behave exactly like R's built-in.

   (d) Create function for auto-printing the variable:

```
print.ufactor <- function(x) {
  cat("Levels:\n ")
  options(digits=15)
  cat( uf$levels[uf$values] )
  cat("Values:\n ")
  cat(uf$values)
}
```

And that's it. with this 5 steps we have a more general factor supporting the 4 built-in types (character, integer, numeric, logical) just like R. R might support more but we care for these mostly. So if you want to get the values and levels you don't need to use as.integers/levels but instead use the ”$” to access the local variables. We also use the lock functions to lock environment and local variables of our ufactor just to remind the user to do not change them at all. R will produce an error. In the end, our variable ”uf” is an environment with class ”ufactor” (for untyped factor) and you can use it for anything without losing speed with the copies that R might do or not. Example:

```
x <- sample( rnorm(10), 1000, replace = TRUE)
r_factor <- factor(x)
u_factor <- ufactor(x)
all.equal( as.integer(r_factor), u_factor$values)  ## TRUE
all.equal( levels(r_factor), as.character(u_factor$levels) ) ## TRUE
```

```
print(r_factor)
print(uf_factor)
r_factor[1] == u_factor[1]   ## TRUE
```

10 CONCLUSION

We have provided a long list of efficient coding tips to help the user make their programs faster. In some cases, the user can benefit from having functions that use less memory, which is also another important feature, apart from speed. We also provided tips to avoid numerical instabilities and numerical overflows. These tips should not go unattended as one can easily face such errors when trying to optimize the efficiency of their code.

ACKNOWLEDGEMENTS

We would like to acknowledge Stefanos Fafalios, Christina Chatzipantsiou, Marco Maier, Panagiotis Tzirakis, Giorgos Athineou, Kleio Lakiotaki, Kurt Hornik, Uwe Ligges and the R Core team.

REFERENCES

Casella, G. and Berger, R. L. (2002). *Statistical inference*. Duxbury Pacific Grove, CA.
Christensen, R. H. B. (2015). *ordinal—Regression Models for Ordinal Data*. R package version 2015.6-28.
Mersmann, O. (2015). *microbenchmark: Accurate Timing Functions*. R package version 1.4-2.1.
Nychka, D., Furrer, R., and Sain, S. (2015). *fields: Tools for Spatial Data*. R package version 8.2-1.
Papadakis, M., Tsagris, M., Dimitriadis, M., Fafalios, S., Tsamardinos, I., Fasiolo, M., Borboudakis, G., Burkardt, J., Zou, C., and Lakiotaki, K. (2017). *Rfast: Fast R Functions*. R package version 1.8.6.
Presnell, B., Morrison, S. P., and Littell, R. C. (1998). Projected multivariate linear models for directional data. *Journal of the American Statistical Association*, 93(443):1068–1077.
Tsagris, M. (2017). Conditional independence test for categorical data using poisson log-linear model. *Journal of Data Science*, 15(2):345–354.
Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S*. Springer, New York, fourth edition. ISBN 0-387-95457-0.
Yee, T. W. (2010). The VGAM package for categorical data analysis. *Journal of Statistical Software*, 32(10):1–34.