1 Introduction

Speeding up our R code with the "compiler" package[1].

It is widely agreed that R is not a fast language. Notably, because it is an interpreted language. To overcome this issue, some solutions exists which allow to compile functions written in R. The gains in computation time can be considerable. But it depends on our ability to write code that can benefit from these tools.

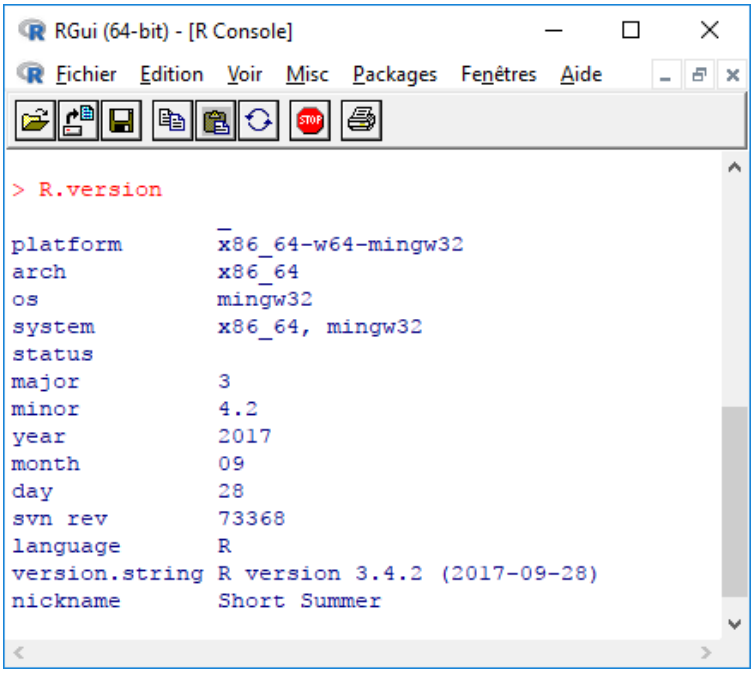
In this tutorial, we study the efficiency of the Luke Tierney's "compiler" package which is provided in the base distribution of R. We program two standard data analysis treatments, (1) with and (2) without using loops: the scaling of variables in a data frame; the calculation of a correlation matrix by matrix product. We compare the efficiency of non-compiled and compiled versions of these functions.

We observe that the gain for the compiled version is dramatic for the version with loops, but negligible for the second variant. We note also that, in the R 3.4.2 version used, it is not needed to compile explicitly the functions containing loops because it exists a JIT (just in time compilation) mechanism which ensure to our code the maximal performance.

This tutorial is initially based on the French version written in July 2012 (version R 2.15.1). When we observe a difference in behavior with the current version (R 3.4.2, October 2017), we will take a different direction.

2 Functions to compile

2.1 R Version



[1] The R version used when writing the French version (July 2012) of this tutorial was R 2.15.1. For this English version (October 2017), we use version R 3.4.2. The screenshots have been updated.

In order for everyone to be able to reproduce the process, here is (above) information about the version of R used during our experiment.

2.2 Scaling of variables

The function must center and scale the variables of a data frame. A new vector is generated for each variable processed. The output is a new data frame with the same number of variables.

The first variant of our function is the following one:

```
#function for scaling with loop
my.scale <- function(X){
 #scaling one variable
 one.scale <- function(x){
  n <- length(x)
  moy <- mean(x)
  et <- sqrt((n-1)/n*var(x))
  y <- numeric(length(x))
  for (i in 1:length(x)){
   y[i] <- (x[i] - moy)/et
  }
  return(y)
 }
 #calling one.scale for all the variables of a data frame
 Y <- as.data.frame(lapply(X,one.scale))
 return(Y)
}
```

« my.scale » applies the function “one.scale” to each column of the data.frame “X”. During the processing, one.scale read each value of the variable using a loop, then it performs a centering and reduction. This writing goes completely against the R philosophy where the basic object is the vector. Hence the second variant exploiting the specificities of R this time.

```
# function for scaling without a loop
R.scale <- function(X){
 #scaling one variable
 one.scale <- function(x){
  n <- length(x)
  moy <- mean(x)
  et <- sqrt((n-1)/n*var(x))
  y <- (x-moy)/et
  return(y)
 }
 #calling one.scale for each variable
 Y <- as.data.frame(lapply(X,one.scale))
 return(Y)
}
```

2.3 Correlation matrix

To calculate the correlation matrix of a data frame, we do the product of the transposed matrix by itself. We divide each value by the number of instances "n". The variables must be centered and reduced beforehand.

Here is the version with nested loops[2].

```
#my function for correlation matrix with nested loops
#the columns are already scaled
my.correlation <- function(X){
 Y <- as.matrix(X)
 TY <- t(Y)
 n <- nrow(X)
 p <- ncol(X)
 M <- matrix(0,p,p)
 for (i in 1:p){
  for (j in 1:p){
   for (k in 1:n){
    M[i,j] <- M[i,j] + TY[i,k]*Y[k,j]
   }
  }
 }
 M <- M/n
 return(M)
}
```

And here is the version based on the R matrix operations capabilities.

```
#R function for correlation matrix using R matrix capabilities
#the columns are already scaled
R.correlation <- function(X){
 Y <- as.matrix(X)
 TY <- t(Y)
 M <- TY%*%Y
 M <- M/nrow(X)
 return(M)
}
```

2.4 Artificial dataset

We use an artificial dataset to evaluate the various functions. We set n = 10,000 instances and p = 5 variables.

```
#generate a dataset
set.seed(1)
n <- 10000 #number of observations
p <- 5 #number of variables
```

[2] The aim is to evaluate the "compiler" package, and not to write the most performing code.

```
dataset <- as.data.frame(matrix(runif(n*p),n,p))
```

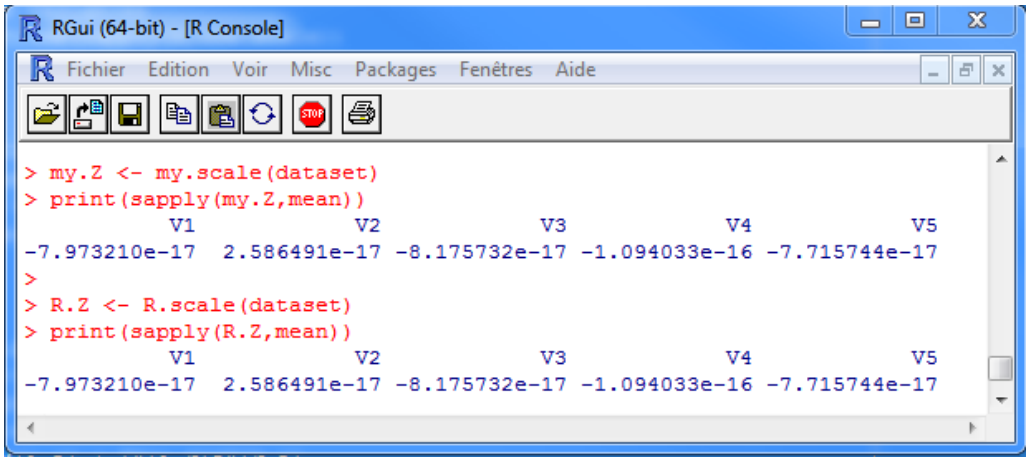
2.5 Comparison of the results

To check the consistency of our functions, we calculate the mean of the resulting variables after the centering and reduction processing.

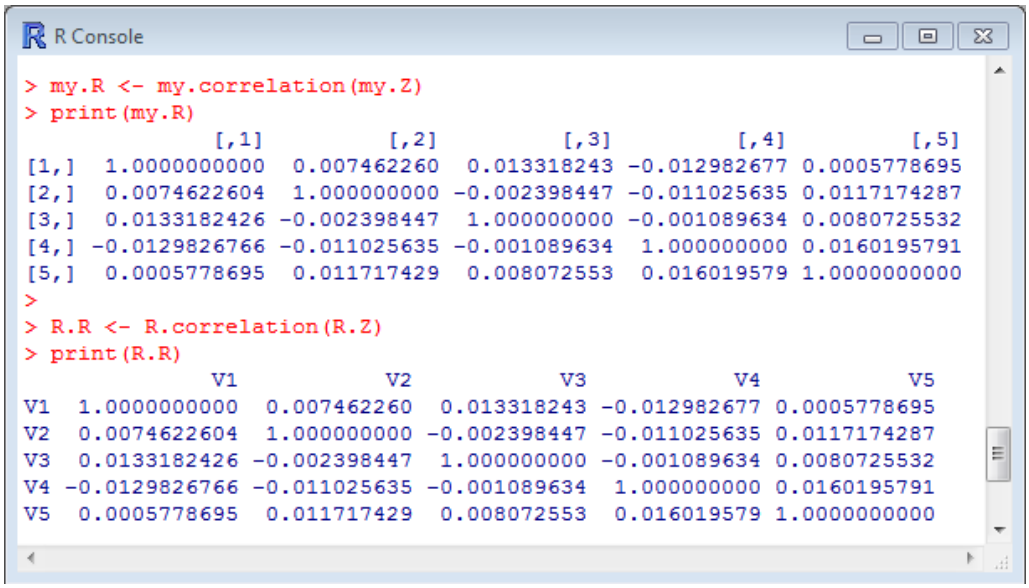
```
#checking the two scaling functions
my.Z <- my.scale(dataset)
print(sapply(my.Z,mean))

R.Z <- R.scale(dataset)
print(sapply(R.Z,mean))
```

We get exactly the same averages per variable for both functions, and the result is the expected one i.e. the mean of each transformed variable is equal to zero.



We then calculated the correlation matrices. The results are consistent and they are identical to those of the **cor()** function of R.



3 Experiment (1)

We can proceed to the study of computing times. We use the **benchmark()** procedure provided by the "rbenchmark" package. It launches several times (parameter 'replications = 10') the functions to be analyzed by measuring the execution time by successive calls to the **system.time()** function.

To make the differences stronger, we have modified the dimensions of the data matrix. We now use n = 50,000 rows and p = 50 columns.

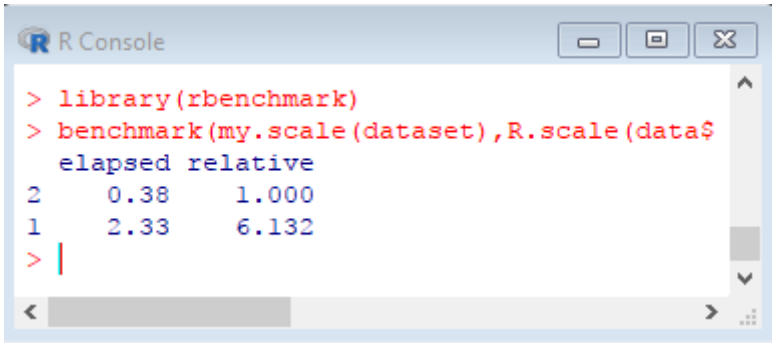
```
#generate the dataset
set.seed(1)
n <- 50000
p <- 50
dataset <- as.data.frame(matrix(runif(n*p),n,p))
```

To compare the two approaches of the scaling functions, we use the following commands:

```
#loading the rbenchmark package
library(rbenchmark)

#comparison of execution time - scaling
benchmark(my.scale(dataset),
    R.scale(dataset),
    columns=c("elapsed","relative"),
    order="relative",
    replications=10)
```

We obtain:



benchmark() displays the total duration of the 10 replicates. It indicates the ratio of duration between the fastest function (the 2nd, centering reduction without loops) and the others.

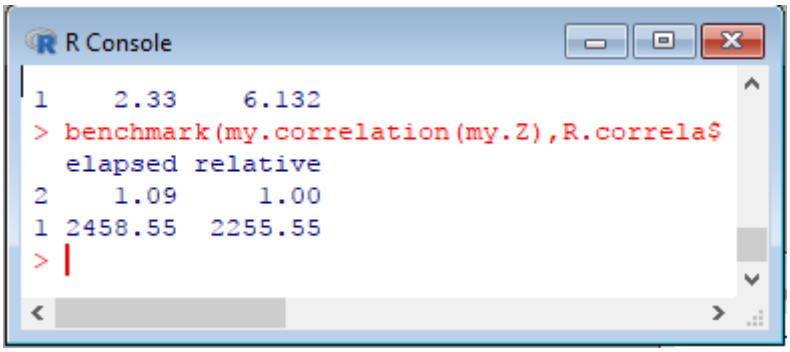
If there were still doubts about the ineffectiveness of the loops in R, they are dispelled here. The second variant took a total of 0.38 seconds for the 10 trials i.e. 0.038 seconds for each call; the first variant, using loops to read the values, took 2.33 seconds i.e. 0.233 seconds per call. The function is 6 times slower.

Note: But nevertheless, I am a little surprised. The ratio between the two approaches was much higher when I used version R 2.15.1 for the French version of this tutorial (**47 times slower**). We will understand why below (section 4).

We perform the same experiment for the calculation of the correlation matrix.

```
#comparison of execution time - correlation matrix
benchmark(my.correlation(my.Z),
    R.correlation(my.Z),
    columns=c("elapsed","relative"),
    order="relative",
    replications=10)
```

The behavior of the variant based on nested loops is totally disastrous.



The ratio is dramatic. The approach based on nested loops is **2255** times slower than the one based on matrix operations (1.09 sec vs. 2458.55 sec.).

Note: The calculation time is 0.93 seconds for 10 replicates of the native cor() function of R.

Note: Here also, I am surprised. In R 2.15.1, the ratio was 3834.

4 Compilation of functions

The package "compiler" enables to compile functions. The use of the **cmpfun()** tool is easy. We compile the various functions above with the following commands:

```
#chargement du package
library(compiler)

#my scale compiled
comp.my.scale <- cmpfun(my.scale)
#R scale compiled
comp.R.scale <- cmpfun(R.scale)

#my correlation compiled
comp.my.correlation <- cmpfun(my.correlation)
#R correlation compiled
comp.R.correlation <- cmpfun(R.correlation)
```

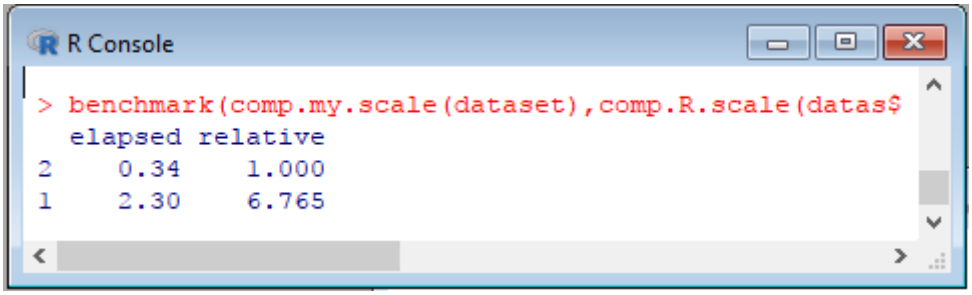
So, we measure again the computation times.

```
#benchmarking scale compiled functions
benchmark(comp.my.scale(dataset),
      comp.R.scale(dataset),
      columns=c("elapsed","relative"),
      order="relative",
```

```
      replications=10)

#benchmarking correlation compiled functions
benchmark(comp.my.correlation(my.Z),
      comp.R.correlation(my.Z),
      columns=c("elapsed","relative"),
      order="relative",
      replications=10)
```

For **my.scale()**, the compiled version **cmp.my.scale()** seems not decrease the calculation time (2.33 sec. vs. 2.30 sec.).

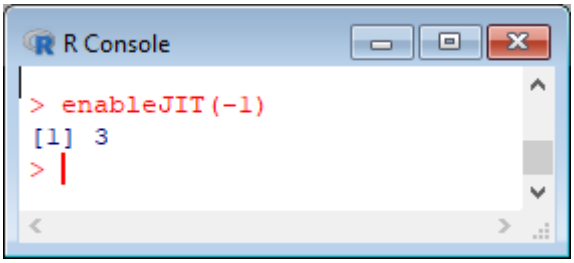


This is very surprising. When I write the French version of this tutorial (2012/07/19), I use **R 2.15.1**. The compiled version was 7 times faster than the non-compiled one. What is different in this version of **R 3.4.2** that we use today?

From here, this English version of the tutorial is different from the French version.

4.1 Checking the activating of the JIT mechanism

By reading the documentation of the package "compiler", I realized that a compilation on the fly mechanism (JIT – just in time compiler) exists, with different levels. We check if it is enabled by default in R 3.4.2 with the **enableJIT()** command.



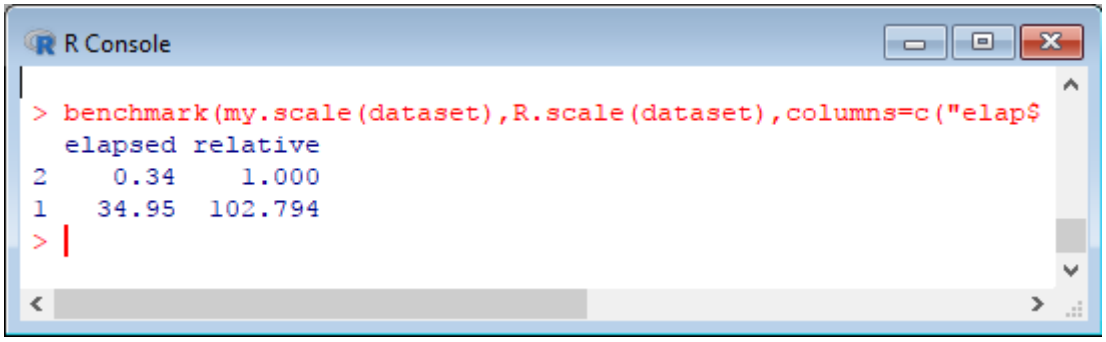
The default level **(LEVEL = 3)** means "*…(large and small) closures are compiled, in addition all top level loops are compiled before they are executed*"**. Actually, the so-called non-compiled functions studied above were compiled with the JIT mechanism. This is the reason for which we have no improvement when we explicitly compile them.**

4.2 Deactivate the JIT mechanism - Non-compiled functions

In this section, we deactivate the JIT and, again, we measure the execution time of the non-compiled version of the scaling function. We set the following commands **after we load the functions in memory**:

```
#deactivating the JIT mechanism
enableJIT(0)
#measuring the scale functions with and without loop
benchmark(my.scale(dataset),R.scale(dataset),
    columns=c("elapsed","relative"),order="relative", replications=10)
```

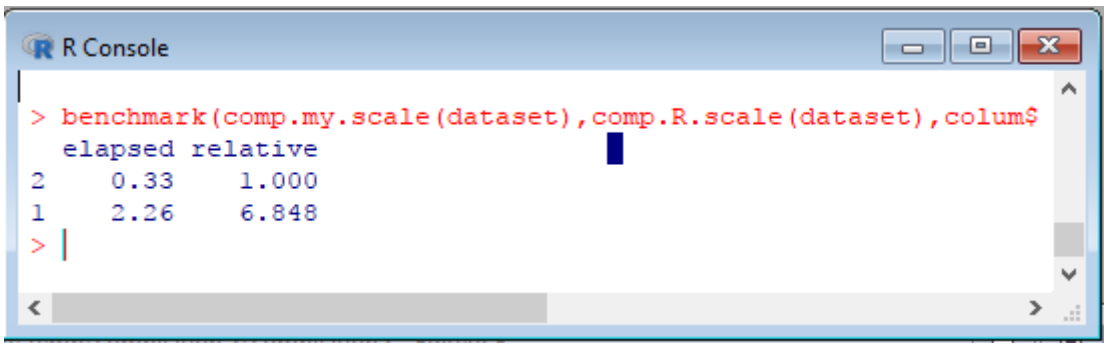
We obtain:



Now, we observe that the use of loops is really disastrous, especially when the function is not compiled under R.

4.3 Deactivate the JIT mechanism – Compiled functions

Even if the JIT is not activated, we observe that the functions explicitly compiled are faster, and the ratio between the versions with and without loops is less important. The computation time of the compiled version is similar to the one observed in the first part of this tutorial (section 3, where the function is compiled on-the-fly with the JIT mechanism).



5 Conclusion

The main conclusion of this tutorial is that loops should not be used under R. When we cannot do otherwise, it is in our best interest to compile the functions. The “compiler” package enables to do it efficiently; the improvement of processing times is dramatic.

We observe in R 3.4.2 that we do not need to compile explicitly the functions. A JIT (just in time compiler) mechanism exists, it is activated in its highest level (LEVEL = 3) by default, it means that the loops are compiled before they are executed. We are assured of maximum performance when programming.