1 Objectif

Présentation du package « compiler » de R (Version de R utilisée : 2.15.1).

De temps en temps, des informaticiens me tombent dessus en m'assénant que R est bien beau, mais que ça reste un langage interprété, donc notoirement lent. Invariablement je réponds que c'est indéniable, mais que nos programmes représentent la plupart du temps un enchaînement d'appels de fonctions qui, elles, sont compilées (depuis la version 2.14 tout du moins). De fait, les temps de traitements sont très peu grevés par les caractéristiques de l'interpréteur. Avec un peu d'expérience, on se rend compte que R est surtout mal à l'aise lorsque nous implémentons explicitement des boucles (ex. la boucle ***for***). Il faut donc les éviter autant que possible.

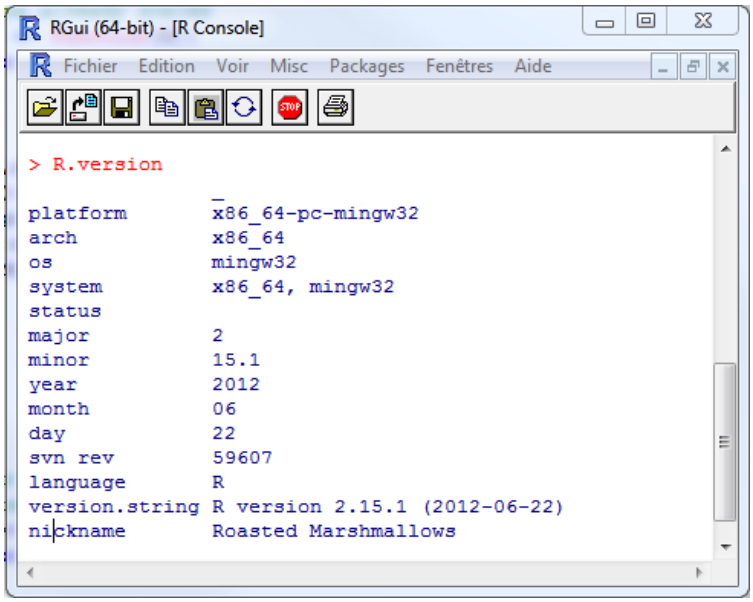
J'en étais resté à cette idée lorsque j'ai découvert le package « compiler » de Luke Tierney, inclus dans la distribution standard de R 2.14[1]. Il serait possible de compiler très simplement un bloc d'instructions intégré dans une fonction. Dans certaines configurations, que l'on s'attachera à déterminer justement, le « byte code » qui en découle se révèlerait nettement plus performant que le code natif passé à la moulinette de l'interpréteur.

Dans ce tutoriel, nous programmons deux traitements classiques de l'analyse de données, (1) avec et (2) sans l'utilisation des boucles : le centrage réduction des variables d'un data frame et le calcul d'une matrice de corrélation par produit matriciel. Dans un premier temps, nous mesurons les temps d'exécution respectifs des versions interprétées. Dans un second temps, nous les compilons avec la fonction « **cmpfun** » du package compiler, puis nous les évaluons de nouveau.

Nous constatons que le gain en vitesse d'exécution de la version compilée est particulièrement spectaculaire pour la version avec boucles, il est négligeable en revanche pour la seconde variante.

2 Configurations à analyser

2.1 Version de R



[1] Tierney, « A Byte Code Compiler for R », http://homepage.stat.uiowa.edu/~luke/R/compiler/compiler.pdf, 03/2012.

Pour que tout un chacun puisse reproduire à l'identique notre expérimentation, voici (ci-dessus) les informations sur la version de R utilisée durant notre expérimentation.

2.2 Centrage – réduction de variables

La fonction doit centrer et réduire les variables d'un ensemble de données (de type data frame dans R) de « n » lignes et « p » colonnes. Une nouvelle variable est générée pour chaque colonne traitée. A la sortie, nous obtenons un nouveau tableau de données.

La première variante de notre fonction s'écrit comme suit :

```
#centrage réduction des variables d'un data frame
my.scale <- function(X){
 #centrage réduction d'une variable – avec utilisation de la boucle for
 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)
 }
 #appel de one.scale pour l'ensemble des variables
 Y <- as.data.frame(lapply(X,one.scale))
 return(Y)
}
```

« my.scale » applique la fonction « one.scale » sur chaque colonne du data.frame « X ». Lors du traitement, « one.scale » accède à l'aide d'une boucle for à chaque cellule du vecteur « x » pour réaliser la transformation. Cette écriture, naturelle pour un informaticien, va totalement à l'encontre de la philosophie de R où l'objet de base est le vecteur. D'où la seconde variante exploitant les spécificités de R cette fois-ci.

```
#centrage réduction des variables d'un data frame
R.scale <- function(X){
 #centrage réduction d'une variable – sans utilisation de boucle for
 one.scale <- function(x){
  n <- length(x)
  moy <- mean(x)
  et <- sqrt((n-1)/n*var(x))
  y <- (x-moy)/et
  return(y)
 }
 #appel de one.scale pour chaque variable
 Y <- as.data.frame(lapply(X,one.scale))
 return(Y)
}
```

2.3 Matrice de corrélation

Pour calculer la matrice de corrélation d'un data frame, nous effectuons le produit de la transposée de la matrice par elle-même. Nous divisons chaque valeur par l'effectif « n ». Attention, il faut que les variables aient été centrées et réduites au préalable.

Voici la version avec des boucles imbriquées[2].

```
#calcul de la matrice de corrélation avec boucles
#les variables sont censées être centrées et réduites
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)
}
```

Et la version exploitant le produit matriciel natif de R.

```
#calcul de la matrice de corrélation sans boucles
#les variables sont censées être centrées et réduites
R.correlation <- function(X){
 Y <- as.matrix(X)
 TY <- t(Y)
 M <- TY%*%Y
 M <- M/nrow(X)
 return(M)
}
```

2.4 Génération des données

Les valeurs sont générées aléatoirement avec **runif(.)**. Dans une première phase, pour vérifier le bon fonctionnement des fonctions, nous fixons n = 10000 et p = 5. Nous les augmenterons lors des expérimentations.

```
#génération des données
set.seed(1)
n <- 10000 #nombre de lignes
p <- 5 #nombre de colonnes
dataset <- as.data.frame(matrix(runif(n*p),n,p))
```

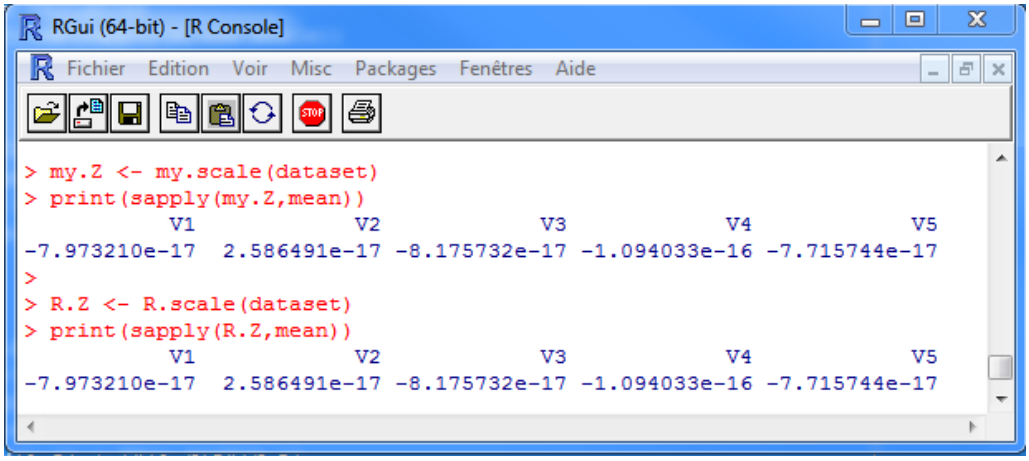
[2] L'écriture du produit matriciel est très naïve. L'objectif est simplement de situer les différentes approches.

2.5 Un exemple d'exécution – Comparaison des fonctions

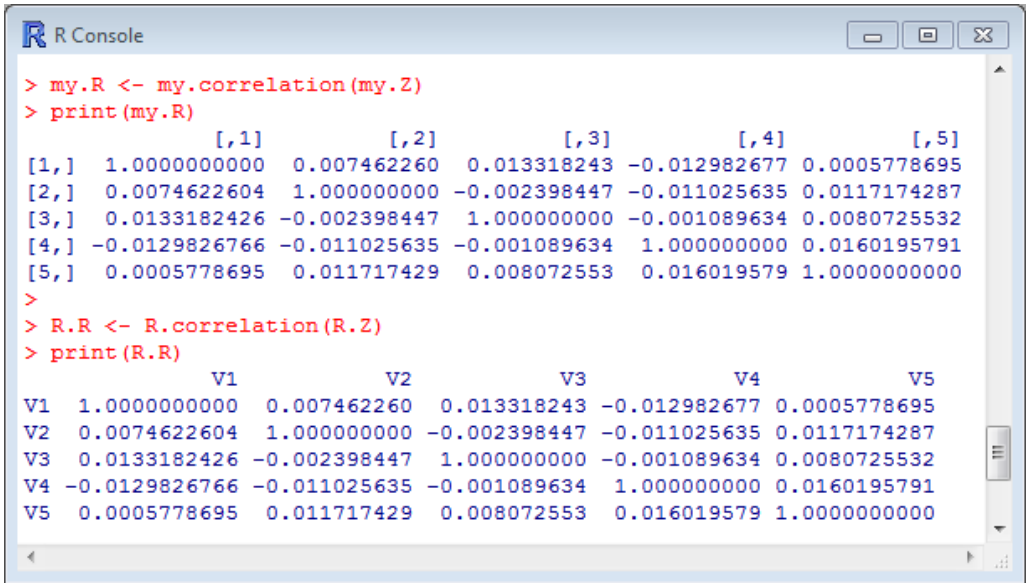
Pour contrôler les fonctions, nous calculons les moyennes des variables lors du centrage réduction.

```
#vérification des fonctions de centrage réduction
my.Z <- my.scale(dataset)
print(sapply(my.Z,mean))

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

Nous obtenons exactement les mêmes moyennes par variable pour les deux fonctions, et le résultat est celui attendu c.-à-d. la moyenne de chaque variable transformée est nulle.



Par la suite, nous avons calculé les matrices de corrélation. Les résultats coïncident, et ils sont identiques à ceux de la fonction **cor()** de R.



3 Expérimentation (1)

L'exactitude des calculs étant assurée, nous pouvons passer à l'étude des temps de traitement. Nous utilisons la procédure **benchmark()** du package « **rbenchmark** ». Elle se charge de lancer plusieurs

fois (paramètre 'replications = 10') les fonctions à analyser en mesurant la durée d'exécution par des appels successifs à la fonction **system.time()**.

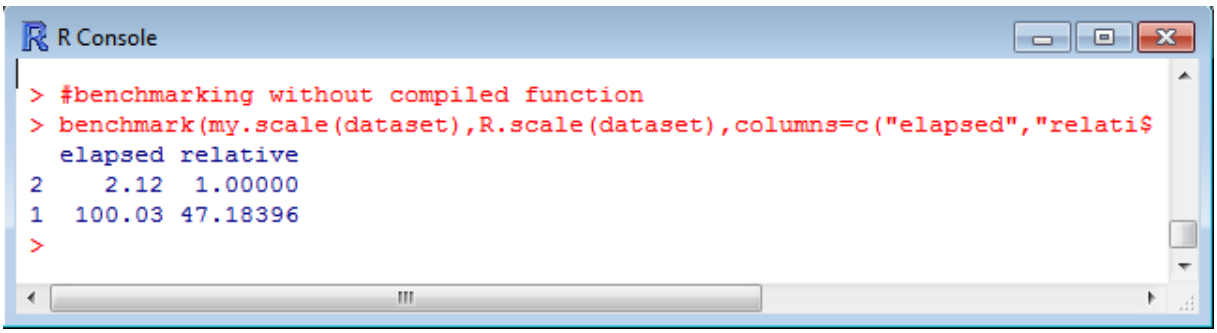
Pour rendre plus marquées les différences, nous avons modifié les dimensions de la matrice de données. Nous utilisons maintenant 'n = 50.000 lignes' et 'p = 50 variables'.

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

Pour comparer les implémentations du centrage réduction, nous utilisons le code suivant :

```
#chargement du package
library(rbenchmark)

#comparaison des durées d'exécution - centrage réduction
benchmark(my.scale(dataset),
    R.scale(dataset),
    columns=c("elapsed","relative"),
    order="relative",
    replications=10)
```

Nous obtenons :



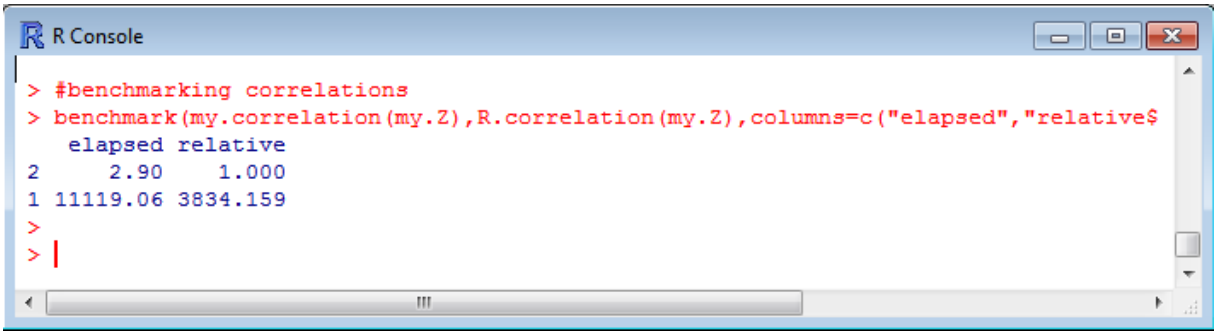
benchmark() affiche la durée totale des 10 réplications. Il indique le rapport de durée entre la fonction la plus rapide (la 2nde, centrage réduction sans les boucles) et les autres.

Si on avait encore des doutes sur l'inefficacité des boucles dans R, elles sont complètement levées ici. La seconde variante a pris 2.12 secondes au total pour les 10 essais, soit 0.212 secondes pour chaque appel ; la première variante, avec le parcours des valeurs à l'aide de la boucle ***for***, en a eu pour 100.03 secondes, soit 10.003 secondes par appel. **La fonction est 47 fois plus lente** !

Nous faisons de même pour les corrélations.

```
#comparaison des durées d'exécution - corrélation
benchmark(my.correlation(my.Z),
    R.correlation(my.Z),
    columns=c("elapsed","relative"),
    order="relative",
    replications=10)
```

Le comportement de la variante basée sur l'imbrication de boucles est carrément catastrophique.



Le ratio est dramatique, le temps de calcul de la première procédure est **3834** fois plus élevé. R n'a visiblement pas aimé les 3 boucles ***for*** imbriquées !

Remarque : A titre de curiosité, le temps de calcul est de 1.67 secondes pour 10 réplications de la fonction native **cor()** de R.

4 Compilation des fonctions

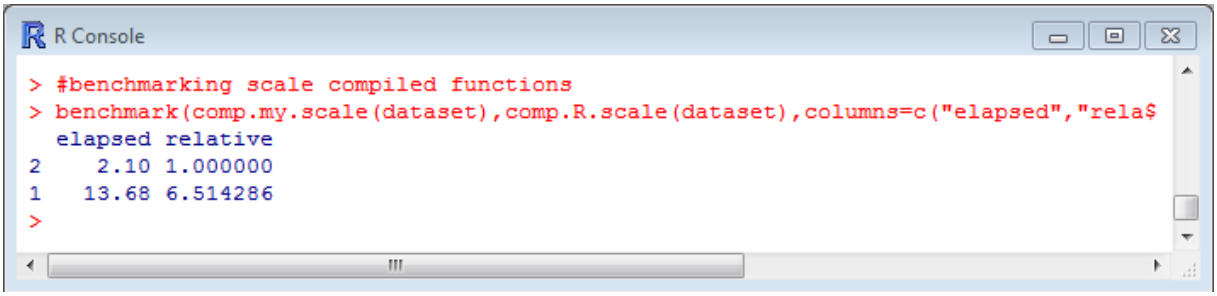
Le package « compiler » introduit la possibilité de compiler des blocs de code intégrés dans des fonctions. L'utilisation de la procédure **cmpfun()** est particulièrement simple. Pour les différentes fonctions ci-dessus, nous produisons les versions compilées à l'aide des instructions suivantes :

```
#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)
```

Il ne nous reste plus qu'à mesurer de nouveau la durée des traitements.

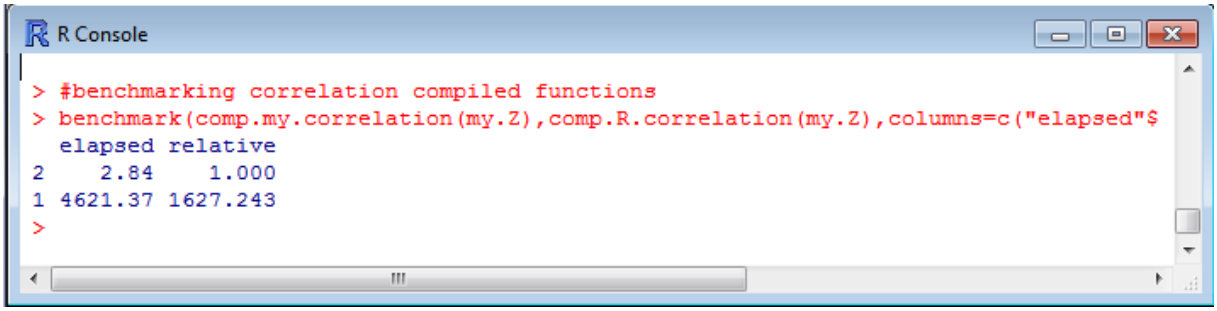
```
#benchmarking scale compiled functions
benchmark(comp.my.scale(dataset),
      comp.R.scale(dataset),
      columns=c("elapsed","relative"),
      order="relative",
      replications=10)

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

Pour **my.scale()**, le passage à la version compilée **cmp.my.scale()** améliore considérablement les performances. Nous sommes passés de 100.03 à 13.68 secondes pour 10 réplications. De fait, le ratio entre les versions sans et avec boucles n'est plus que de 6.51. La compilation emmène indubitablement un plus considérable dans ce contexte.



Pour la version compilée **comp.R.scale()**, le bilan est tout autre. Le gain est nul par rapport à **R.scale()** (de 2.12 sec. à 2.10). De par la structure de notre programme, la compilation n'apporte pas d'améliorations.

Le constat est similaire concernant la corrélation. **R.correlation()** ne tire pas parti de la compilation (2.90 sec. vs. 2.84 sec.) ; la version compilée de **my.correlation()** en revanche divise par 2.5 la durée d'exécution (11119 sec. vs. 4621 sec.).



5 Bilan

Récapitulons les durées d'exécution pour chaque appel de fonction (moyenne des 10 réplications).

Durée (sec.)	**Avec boucles**		**Sans boucles** (appels de fonctions R)	
	Interprété	**Compilé**	**Interprété**	**Compilé**
Centrage – réduction	10.003	**1.368**	0.212	0.210
Corrélation	1111.906	**462.137**	0.290	0.284

Se conformer à la philosophie de R (*pas de boucles !*) permet d'obtenir des programmes véloces. Dans ce cas, l'usage du package 'compiler' n'est pas très utile. Le gain est négligeable.

Si l'on persiste à utiliser des boucles, ou tout simplement parce qu'on ne peut pas faire autrement, le passage à la compilation est une bonne piste d'optimisation. La réduction du temps de calcul peut être spectaculaire.

Il n'en reste pas moins que la procédure « avec boucles » compilée est moins rapide que la procédure « sans boucles » interprétée. C'est aussi un des enseignements forts de notre expérimentation. Au-delà des outils miracles, bien écrire son programme reste la meilleure manière de garantir les meilleures performances.

6 Conclusion

R est un langage de programmation plutôt rapide, sauf lorsque l'on utilise intensivement les boucles. Tout ce que l'on peut mettre en œuvre pour les éviter est donc bon à prendre. A défaut, une manière simple d'accélérer les traitements consiste à compiler les fonctions qui en comportent. Le package « compiler » de R permet de le faire très simplement.

Mais la compilation n'est efficace que pour certains types de code. Depuis R **2.14**, toutes les fonctions standards et les packages sont pré-compilées. Il ne sert donc à rien de tenter d'optimiser via la librairie « compiler » les programmes constitués d'une succession d'appels à ces fonctions.

Enfin, rendons à César ce qui lui appartient, l'idée de ce tutoriel m'est venue à la lecture de l'excellent article de Dirk Eddelbuettel accessible en ligne : « The new R compiler package in R 2.13.0 : Some first experiments ». Sur le fond, il est particulièrement intéressant. Sincèrement, je ne pensais pas qu'il était possible de gagner autant en rapidité en plaçant judicieusement les parenthèses ou les accolades dans un programme R. Manifestement, il me reste encore beaucoup de choses à apprendre concernant ce langage. Je serais en revanche un peu plus réservé concernant la forme. Un non spécialiste désireux d'accélérer ses programmes aura vraisemblablement du mal à y trouver des astuces exploitables pour son propre code source. J'ai donc repris à mon compte la démonstration en la simplifiant pour l'axer sur deux idées importantes : (a) le problème que posent les boucles sous R ; (b) l'opportunité de compiler les fonctions pour les rendre plus rapides.