
Bonne pratique de codage

Librement inspiré du Style Guide, by Hadley Wickham

- C'est important d'adopter des bonnes pratiques de codage :
 - permettre une lecture et une compréhension simple et rapide du code
 - tant pour le(s) développeur(s), que pour les utilisateurs, et favoriser le travail collaboratif
- Il n'y a pas un style parfait, le principal est d'en adopter un et de s'y tenir

```
# dur à lire
aze=data.frame(cole=rnorm(1000),refdzf=LETTERS[1:2]);ff=lapply(split(aze$cole,aze$refdzf),
                                                                function(x){mean(x)});ff

# c'est mieux quand même... ?
data <- data.frame(value = rnorm(1000), group = LETTERS[1:2])
mean_group <- lapply(
  split(data$value, data$group),
  function(x){
    mean(x)
  })
mean_group
```

Fichiers

Les noms doivent être **explicites** et se terminer par **.R**. Si les scripts sont ordonnés, les pré-fixer par un numéro.

# Good	# Bad	0-download.R
modélisation.R	toto.r	1-parse.R

Variables et fonctions

- Noms **courts** et **explicites**, de préférence **en minuscule**, en évitant d'utiliser des noms de fonctions connues...
- Utilisation d'un underscore (**_**) pour séparer les noms. Eviter le point (**.**), il peut amener de mauvaises interactions avec d'autres langages (java, javascript, ...)
- Variable == noms, fonctions == verbes, autant que possible...
- **Pas d'accents !**

# Good	# Bad
day_one	first_day_of_the_month
day_1	DayOne
	mean <- function(x) sum(x)

Espacer son code

- Mettre des espaces **autour** de tous les opérateurs (**=**, **+**, **-**, **<-**, etc.), **surtout** à l'intérieur de l'appel d'une fonction.
- Mettre un espace **après** une virgule, **pas avant**

- Essayer de mettre un espace avant l'ouverture d'une parenthèse, **sauf dans l'appel d'une fonction**

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

- Exception pour `:`, `::` and `:::`

# Good	# Bad
x <- 1:10	x <- 1 : 10
base::get	base :: get
if (debug) do(x)	if(debug)do(x)
plot(x, y)	plot (x, y)

Namespace et appel d'une fonction

- `::` accès aux fonctions exportées d'un package
- `:::` accès aux fonctions *cachées* d'un package

Bonne pratique

- essayer de toujours préfixer l'appel à une fonction par `::`
 - meilleure lisibilité des appels / dépendances
 - évite des conflits potentiels : deux fonctions du même nom dans deux packages différents...
- éviter l'utilisation des fonctions *cachées* `:::`

```
require(FactoMineR)

# Good
FactoMineR::PCA(X, scale.unit = TRUE)
```

Accolades et indentation

- L'ouverture d'une accolade doit **toujours** être suivie d'un passage à la ligne.
- La fermeture d'un accolade doit être suivie d'un passage à la ligne, sauf dans le cas d'un **else**
- Le code à l'intérieur des accolades doit être indenté

# Good	# Bad
if (y == 0) {	if (y == 0) {
log(x)	log(x)
} else {	}
y ^ x	else{ y ^ x}
}	

- Indenter son code, de préférence en utilisant deux espaces. **Raccourci RStudio : Ctrl+A, Ctrl+I**
- Cas particulier dans la définition d'une fonction

```
long_function_name <- function(a = "a long argument",
                                b = "another argument",
                                c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

Assignement

- Utiliser `<-`, et **banir** `=`, lors de l'assignement

```
# Good
x <- 5
# Bad
x = 5
```

Commentaires

- Commenter son code, toujours dans un soucis de lecture et de collaboration. **Raccourci RStudio : Ctrl+Shift+C**
- un commentaire comportant au-moins `----` crée une section pouvant être réduite

```
# Load data -----
# Plot data -----
```

Structures conditionnelles

if / else/ else if

```
if(condition1){
  print("la condition1 est vrai")
}else if(condition2){
  print("la condition1 est fausse, mais la condition2 est vrai")
}else{
  print("les conditions sont fausses... :-(")
}
```

- la condition doit retourner une (**et une seule**) valeur logique (TRUE/FALSE)

ifelse, une variante

```
ifelse(vecteur.condition, vecteur.vrai, vecteur.faux)
```

- Pour chaque élément i , regarde `condition[i]`, et retourne `vrai[i]` ou `faux[i]`

```
x <- 1:2
ifelse(x%%2 == 0, 0, x)      #> [1] 1 0
```

switch

- Suivant les cas, une autre façon de faire un **if / else if**

```
res <- switch(valeur,
  cas1 = resultat1,
  cas2 = resultat2,
  cas3 = resultat3,
  sinon (optionnel))
```

```

fonction <- "mean"
x <- rnorm(100)
res <- switch(fonction,
  mean = mean(x),
  median = median(x),
  sum = sum(x))
res

## [1] 0.08430679

```

Opérateurs logiques

- ==, !=, >, <, >=, <=

```

x <- 1

x == 1 # TRUE
x != 1 # FALSE
x < 1  # FALSE

vx <- c(1, 2)
vx != 1 # FALSE TRUE

```

- any : retourne vrai si au-moins un élément répond à la condition

```

x <- c(1:10)

any(x == 10) # TRUE
any(x > 10)  # FALSE

```

- all : retourne vrai si tous les éléments répondent à la condition

```

x <- c(1:10)

all(x <= 10) # TRUE
all(x > 10)  # FALSE

```

- %in% : vérifie l'appartenance de chaque élément d'un vecteur à un autre ensemble

```

x <- "rennes"
x %in% c("rennes", "brest") # TRUE

x <- c("rennes", "paris")
x %in% c("rennes", "brest") # TRUE FALSE

```

- is.vector, is.data.frame, is.list, ...

```

x <- c(1:10)

is.vector(x) # TRUE

```

- ! : retourne la négation

```
x <- 1
y <- 10

(x == 1 & y == 10)  # TRUE
!(x == 1 & y == 10) # FALSE
```

- `&` : opérateur logique ‘AND’. Vrai si les deux conditions sont vraies, faux sinon

```
(x == 1 & y == 10)  # TRUE
(x == 1 & y == 9)  # FALSE
```

- `|` : opérateur logique ‘OR’. Vrai si au-moins une des deux conditions est vraie, faux sinon

```
(x == 1 | y == 10)  # TRUE
(x == 1 | y == 9)   # TRUE
(x == 2 | y == 9)   # FALSE
```

- `xor` : opérateur logique ‘OR’ exclusif. Vrai si une et une seule condition est vraie, faux sinon

```
xor(TRUE, FALSE) # TRUE
xor(TRUE, TRUE)  # FALSE
```

Les boucles

- Rarement efficaces...
- Donc à utiliser avec précautions dans **R**
- Utiliser de préférence les propriétés offertes par la **vectorisation**, et la “**Apply Family**”

For

On parcourt un ensemble d’éléments

```
for(variable in elements){
  ...
}
```

```
for(lettre in LETTERS[1:2]){
  print(lettre)
}
```

```
## [1] "A"
## [1] "B"
```

While

- Tant que la condition est vrai, on continue
 - Si elle est fausse au départ, rien ne s’exécute
 - Si elle est toujours vraie, ou s’il n’y a pas de sortie **explicite**, elle continue à tourner...!

```
while(condition){
  ...
}
```

```
x <- 1
while(x < 4){
```

```
print(x)
x <- x+1
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

Repeat

- Tant qu'on ne sort pas, on continue
 - L'exécution a donc lieu au-moins une fois
 - Utilisation de **break** pour sortir

```
repeat{
  ...
  if(condition) break
}
```

```
x <- 1
repeat{
  x <- x+1
  if(x == 3){
    print("x vaut 3, on s'arrête.")
    break
  }
}
```

```
## [1] "x vaut 3, on s'arrête."
```

Break et next

- **break** : Sortie immédiate d'une boucle **for**, **while** ou **repeat**
- **next** : Itération suivante d'une boucle **for**, **while** ou **repeat**

```
for(i in 1:3){
  if(i%%2 != 0) {
    next
  }
  print(i)
}
```

```
## [1] 2
```

Un petit mot sur la vectorisation

‘La vectorisation est le processus de conversion d'un programme informatique à partir d'une implémentation scalaire, qui traite une seule paire d'opérandes à la fois, à une implémentation vectorielle qui traite une opération sur plusieurs paires d'opérandes à la fois. Le terme vient de la convention de mettre les opérandes dans des vecteurs ou des matrices.’ (Wikipédia)

- R est un langage **interprété**

- Beaucoup de calculs pouvant être réalisés par une boucle peuvent se faire en utilisant la vectorisation, avec une performance accrue :
 - opérations sur des vecteurs
 - opérations sur des matrices (= un ensemble de vecteurs)
 - opérations sur des data.frame
 - Une performance accrue, pourquoi ?
 - **R**, et ses fonctions “de base” sont codés en **C**, **Fortran**, ...
 - avec l’utilisation efficace et optimisée dans “routines” d’algèbre linéaire (*BLAS*, *LAPACK*, ...)
-

Cas exemple : la somme de deux vecteurs

```
x <- rnorm(1000000)
y <- rnorm(1000000)
res <- integer(1000000)
# calcul de la somme via une boucle
system.time(for(i in 1:length(x)){
  res[i] <- x[i] + y[i]
})
```

```
##      user  system elapsed
##      0.1      0.0      0.1
```

```
# avec la vectorisation
system.time(res2 <- x + y)
```

```
##      user  system elapsed
##       0       0       0
```

```
identical(res, res2)
```

```
## [1] TRUE
```

Y penser donc pour notamment :

- opérations entre vecteurs / matrices

```
x <- matrix(ncol = 2, nrow = 2, 1)
y <- matrix(ncol = 2, nrow = 2, 2)
```

```
z <- x + y
z
```

```
##      [,1] [,2]
## [1,]   3   3
## [2,]   3   3
```

- Création / modification de colonne

```
data <- data.frame(x = 1:10, y = 100:109)
data$z <- data$x + data$y
head(data, n = 2)
```

```
##      x  y  z
## 1 1 100 101
## 2 2 101 103
```

Les fonctions

On définit une nouvelle fonction avec la syntaxe suivante :

```
fun <- function(arguments) expression
```

- **fun** le nom de la fonction
- **arguments** la liste des arguments, séparés par des virgules. *formals(fun)*
- **expression** le corps de la fonction. une seule expression, ou plusieurs entre des accolades. *body(fun)*

```
test <- function(x) x^2
test                # function(x) x^2

formals(test)       # $x
body(test)          # x^2
environment(test)   # <environment: R_GlobalEnv>
```

- Une fonction appartient à un environnement. Le plus souvent un package, ou alors l'environnement global **GlobalEnv**. *environment(fun)*
-

Les arguments

- **Valeur par défaut**
 - via une affectation, avec '=', dans la définition de la fonction
 - optionnel lors de l'appel

```
test <- function(x, y = 2){
  x + y
}
test(x = 2)           # 4
test(x = 2, y = 10)   # 12
```

- **Quelques fonctions utiles de contrôle :**
 - *missing(arg)* : retourne TRUE si l'argument est manquant lors de l'appel
 - *match.arg()* : permet de renseigner le paramètre partiellement (ex : "p" pour point)
 - *typeof(arg)*, *class(arg)*, *is.vector()*, *is.data.frame()*, ...

```
match.arg("mea", c("mean", "sum", "median")) # "mean"
class(10)                                     # "numeric"
```

- **Dépendances entre arguments**

On peut définir un argument en fonction d'autres arguments

```
# avec une expression simple
test <- function(x, y = x + 10){
  x + y
}
test(5) # 20

# un peu plus compliqué
test <- function(x,
  sum = if(class(x) %in% c("numeric", "integer")){
    "sum"
  }else{
    "length"
  })
```



```

    }){

    do.call(fun, list(x = x))
  }

test(1:10)          #55
test(LETTERS[1:10]) #10

```

- Evaluation des arguments

Point Important : les arguments ne sont évalués que lorsqu'ils sont appelés, sinon ils n'existent pas dans la fonction. ... Pour forcer l'évaluation, on peut utiliser la fonction *force()*. Démonstration :

```

f <- function(x) {
  10
}
f(stop("This is an error!"))

# la fonction retourne 10 alors que l'argument est un stop...
# 10

# utilisation de force
f <- function(x) {
  force(x)
  10
}
f(stop("This is an error!"))

# Error: This is an error!

```

Comprendre les '...'

- Signifie que la fonction accepte d'autres arguments que ceux définis explicitement
- Sert généralement à passer ces arguments à une autre fonction
- Se récupère facilement avec : *list(...)*

```

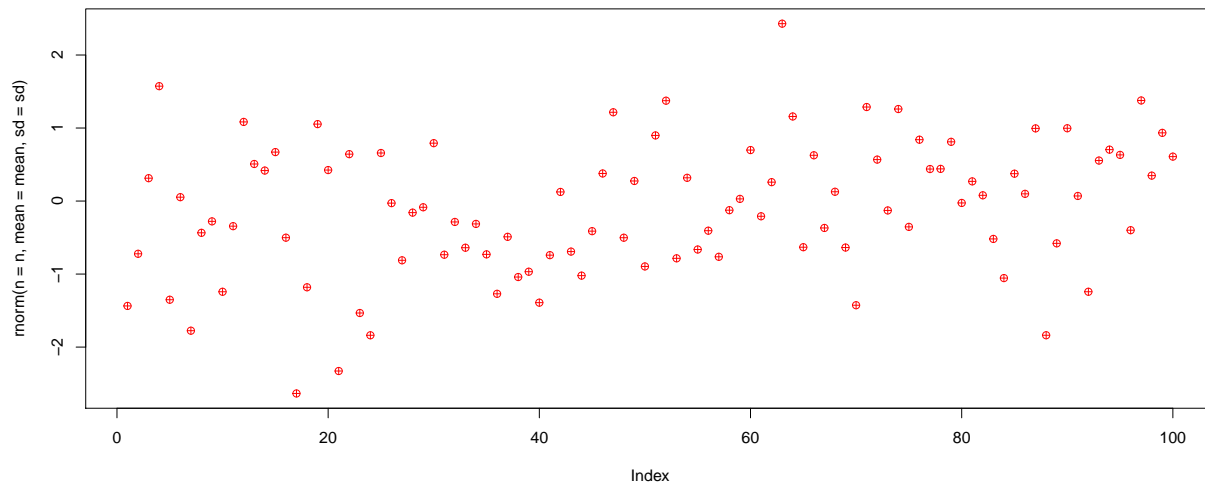
viewdot <- function(arg, ...){
  list(...)
}
viewdot(arg = 1, x = 2, name = "name")

#$x
#[1] 2
#
#$name
#[1] "name"

rnormPlot <- function(n, mean = 0, sd = 1, ...){
  plot(rnorm(n = n, mean = mean, sd = sd), ...)
}
rnormPlot(n = 100, main = "Comprendre les ...", col = "red", pch = 10)

```

Comprendre les ...



Retourner un résultat

Une fonction retourne par défaut le résultat de la dernière expression

```
test <- function(x, y = 2){  
  x + y  
}  
test(2)
```

```
## [1] 4
```

```
somme <- test(x = 2, y = 2)  
somme
```

```
## [1] 4
```

- Renvoi d'un résultat avant la fin de la fonction : fonction *return()*
- Utilisation de *return()* pour la dernière expression ? **Inutile.**
- Retour de plusieurs résultats : liste nommée.
- Aucun résultat ? Possible avec par exemple la fonction *invisible()*

-
- Utilisation de la fonction *return()*

```
test <- function(x, y = 2){  
  if(y == 0){  
    return(x)  
  }  
  x + y  
}  
test(2)
```

```
## [1] 4
```

- Plusieurs résultats

```
test <- function(x, y = 2){
  list(x = x, y = y)
}
test(2)
```

```
## $x
## [1] 2
##
## $y
## [1] 2
```

-
- La fonction *invisible()*

“This function can be useful when it is desired to have functions return values which can be assigned, but which do not print when they are not assigned”

```
test <- function(x, y = 2){
  x + y
  invisible()
}
test(2)  # no print on console
res <- test(2)
res      # and NULL result
```

```
## NULL
```

```
test <- function(x, y = 2){
  invisible(x + y)
}
test(2)  # no print on console
res <- test(2)
res      # but a result !
```

```
## [1] 4
```

Variables locales et globales

- Une variable définie dans une fonction est **locale** :
 - elle ne sera pas présente ensuite dans l'espace de travail
 - elle n'écrasera pas une variable du même nom existante

```
x <- 100
test <- function(x, y){
  x <- x + y
  x
}

# la fonction retourne bien 10
test(5, 5)
```

```
## [1] 10
```

```
# et x vaut bien toujours 100
x
```

```
## [1] 100
```

-
- Via l'opérateur d'affectation <<-, on peut affecter ou modifier une variable **globale**
 - Autant que possible **non-recommandé**...!

```
x <- 100
test <- function(x, y){
  x <- x + y
  y <- y
  x
}

# la fonction retourne ... 5 ?
test(5, 5)

## [1] 5
# et x vaut maintenant 10, et y 5
x ; y

## [1] 10
## [1] 5
```

-
- Et si la fonction utilise une variable non-définie ?

```
test <- function(x){
  x + z
}

# Erreur, z n'existe pas
test(5)

#> Error in test(5) : object 'z' not found

# Si, à tout hasard, une variable 'z' existe dans un autre environnement
# au moment de l'appel, la fonction l'utilise...
z <- 5
test(5)

#> 10
```

- **R** va chercher une variable d'un même nom dans les environnements *parents*. Pratique également à éviter.

Il faut passer tous les arguments en paramètres, et retourner l'ensemble des résultats souhaités en sortie

Fonctions anonymes

Comme son nom l'indique, une fonction qui n'a pas de nom...

- fonction courte, utilisée dans une autre fonction
- qui n'a pas pour but d'être ré-utilisée par la suite

```
f <- function(x){
  x + 1
}
```

```

}

res1 <- sapply(1:10, f)

res2 <- sapply(1:10, function(x) x + 1)

res1
## [1] 2 3 4 5 6 7 8 9 10 11

res2
## [1] 2 3 4 5 6 7 8 9 10 11

```

Communication

Quand on développe, il est important d’anticiper les problèmes potentiels du code :

- mauvais type d’argument
- fichier non-existant
- données manquantes, valeurs infinies, ...

Et communiquer avec l’utilisateur. Trois niveaux sont disponibles :

- fonction `stop()` : erreur “fatale”, l’exécution se termine. A utiliser quand la suite du code ne peut pas être exécutée
- fonction `warning()` : problème “potentielle”, l’exécution continue, mais il y aura peut-être un soucis...
- fonction `message()` : message “informatif”, l’exécution continue.

```

test <- function(x){
  # pour une erreur plus compréhensible
  if(missing(x)){
    stop("x is missing. Please enter a valid argument")
  }
  if(!class(x) %in% c("numeric", "integer")){
    x <- as.numeric(as.character(x))
    warning("x is coerced to numeric")
  }
  message("compute x*2")
  x*2
}

try(test())

## Error in test() : x is missing. Please enter a valid argument
##> Error: x is missing. Please enter a valid argument

test("5")

## Warning in test("5"): x is coerced to numeric
## compute x*2
## [1] 10

```

Et la documentation dans tout ça ?

La documentation est très importante :

- pour que l'utilisateur sache comment utiliser la fonction
- pour vous et d'autres développeurs, lors d'améliorations

Adopter la convention *doxygen*

- simple d'utilisation
- utiliser dans de nombreux langages de programmation
- via le package roxygen2, vous simplifiera ensuite la vie si vous créez des packages !

Utilisation dans R

- en commençant la ligne par #'

<http://r-pkgs.had.co.nz/man.html>

Les balises indispensables

- @param : pour les arguments
- @return : pour le résultat
- @examples : pour les exemples
- @import : packages dépendants utilisés
- @importFrom : packages dépendants utilisés (mais importation uniquement de quelques fonctions)

Penser à préfixer le nom des fonctions utilisées par le package::fonction, et cela même pour les packages de base :

```
# Bad                                # Good
res_pca <- PCA(decathlon)             res_pca <- FactoMineR::PCA(decathlon)
```

```
## le titre de ma fonction
#'
## Une description succincte de ma fonction
## sur plusieurs lignes si on veut
#'
#' @param nom : Character. Nom de la personne
#' @param prenom : Character. Prénom de la personne
#'
#' @return : Character. Identification de la personne
#'
#' @importFrom base paste0
#'
#' @examples
#' # les exemples sont exécutables dans RStudio avec Ctrl+Entrée
#' identify("Thieurmél", "Benoit")
identify <- function(nom, prenom){
  base::paste0("Nom :", nom, ", prénom :", prenom)
}
```

Gestion des erreurs et des messages

Quand **R** rencontre une erreur, il s'arrête net. Dans certains cas, on voudrait pouvoir continuer notre calcul. Trois fonctions sont disponibles dans R :

- `try()` : la plus simple pour contrôler l'apparition d'erreurs
- `tryCatch()` : la plus complète, avec la définition d'action en cas d'erreurs / warnings / messages
- `withCallingHandlers()` : une variante de `tryCatch()`

```
test <- sapply(list(1:5,"a", 6:10), log)
#>Error in FUN(X[[2L]], ...) :
# non-numeric argument to mathematical function
```

```
try(expr, silent = FALSE)
```

- `silent` : affichage ou non d'erreur
- retourne un objet de `class try-error` incluant le message d'erreur

```
test <- sapply(list(1:2,"a"), function(x) try(log(x), silent = TRUE));test

## [[1]]
## [1] 0.0000000 0.6931472
##
## [[2]]
## [1] "Error in log(x) : argument non numérique pour une fonction mathématique\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in log(x): argument non numérique pour une fonction mathématique>
# on récupère un objet de class "try-error", avec le message d'erreur
class(test[[2]])

## [1] "try-error"
test[[2]][1]

## [1] "Error in log(x) : argument non numérique pour une fonction mathématique\n"
```

```
tryCatch(expr, ..., finally)
```

- `error = function(e)` : fonction à exécuter en cas d'erreur, `e` étant le message.
- idem avec `warning = function(e)` et `message = function(e)`

Si ces fonctions sont définies, elles seront donc évaluées le cas échéant **ET le calcul sera arrêté**

```
test <- tryCatch(log("a"), error = function(e){
  print(e)
  return(0)
})

## <simpleError in log("a"): argument non numérique pour une fonction mathématique>
test

## [1] 0
```

```
withCallingHandlers(expr, ..., finally)
```

- `error = function(e)` : fonction à exécuter en cas d'erreur, `e` étant le message
- idem avec `warning = function(e)` et `message = function(e)`

Si ces fonctions sont définies, elles seront donc évaluées le cas échéant **MAIS le calcul continuera**

```
f <- function(){message("message") ; 0}
test <- withCallingHandlers(f(), message = function(e){e})
```

```
## message
```

```
test
```

```
## [1] 0
```

```
# tryCatch
```

```
test <- tryCatch(f(), message = function(e){e})
test
```

```
## <simpleMessage in message("message"): message
## >
```

La “Apply family”

R donc pas au top pour interpréter et exécuter efficacement des boucles **for**

- Une solution **radicale** : NE PAS LES UTILISER !
- Penser à la vectorisation
- Utiliser la “**Apply family**”
 - **apply** : appliquer une fonction sur un data.frame, une matrice, ou un tableau multi-dimensionnel
 - **lapply** : appliquer une fonction sur une liste, ou un vecteur, et retourne une liste
 - **sapply** : identique à **lapply**, mais essaye de structurer un peu mieux les résultats si cela est possible
 - **vapply** : identique à **sapply**, en permettant de définir (un peu) le format des résultats
 - **mapply** : prend en entrée plusieurs vecteurs/listes, et applique la fonction sur les premiers éléments de chaque entrées, puis sur les seconds,
 - **rapply** : exécution récursive de **apply**, avec contrôle préalable des éléments
 - **tapply** : calculs par sous-population

Apply

```
apply(X, MARGIN, FUN, ...)
```

- **X** : une matrice ou un tableau
- **MARGIN** : un vecteur d'entiers contenant la ou les dimensions sur lesquelles on souhaite appliquer la fonction (1 : lignes, 2 : colonnes)
- **FUN** : la fonction à appliquer
- ... : ensemble d'arguments supplémentaires, à passer à la fonction

```
x <- cbind(x1 = 3, x2 = c(NA, 4:1, 2:6))
apply(x, 2, mean)                # moyenne par colonnes
```

```
## x1 x2
```

```
## 3 NA
```



```
apply(x, 2, mean, na.rm = TRUE)    # en passant un argument
```

```
##          x1          x2
## 3.000000 3.333333
```

lapply, sapply, vapply

```
lapply(X, FUN, ...)
```

```
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

- **X** : un vecteur ou une liste
- **FUN** : la fonction à appliquer à tous les éléments de X
- **...** : ensemble d'arguments supplémentaires, à passer à la fonction
- **simplify** : booléen ou caractère, pour simplifier les résultats
- **USE.NAMES** : booléen. Si X est nommé, les utiliser dans les résultats ?
- **FUN.VALUE** : un "template" pour les résultats

Essayons de comprendre ces petites différences...

1. Calcul de la moyenne, soit une valeur par élément

- les données de départ :

```
x <- list(a = 1:3, b = rnorm(5))
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] -0.47313867  0.51551910 -0.05452259 -0.64979368  0.49577062
```

- lapply retourne donc une liste

```
lapply(x, FUN = mean)
```

```
## $a
## [1] 2
##
## $b
## [1] -0.03323304
```

- sapply simplifie les résultats dans un vecteur

```
sapply(x, FUN = mean)
```

```
##          a          b
## 2.00000000 -0.03323304
```

- vapply attend une précision sur le résultat

```
# on s'attend à récupérer une valeur numérique
vapply(x, FUN = mean, FUN.VALUE = 0)
```

```
##           a           b
## 2.00000000 -0.03323304

# et si on s'attend à récupérer une valeur logique ?
vapply(x, FUN = mean, FUN.VALUE = TRUE)

# Error in vapply(x, FUN = mean, FUN.VALUE = TRUE) :
#   values must be type 'logical',
#   but FUN(X[[1]]) result is type 'double'
```

2. Calcul des quantiles, soit 5 valeurs par éléments

- lapply retourne donc une liste

```
lapply(x, FUN = quantile)
```

```
## $a
##  0%  25%  50%  75% 100%
##  1.0  1.5  2.0  2.5  3.0
##
## $b
##           0%           25%           50%           75%           100%
## -0.64979368 -0.47313867 -0.05452259  0.49577062  0.51551910
```

- sapply simplifie les résultats dans une matrix

```
sapply(x, FUN = quantile)
```

```
##           a           b
## 0%    1.0 -0.64979368
## 25%   1.5 -0.47313867
## 50%   2.0 -0.05452259
## 75%   2.5  0.49577062
## 100%  3.0  0.51551910
```

- Formattage avec vapply

```
vapply(x, FUN = quantile, FUN.VALUE = c(Min. = 0, "1st Qu." = 0,
  Median = 0, "3rd Qu." = 0, Max. = 0))
```

```
##           a           b
## Min.     1.0 -0.64979368
## 1st Qu.  1.5 -0.47313867
## Median   2.0 -0.05452259
## 3rd Qu.  2.5  0.49577062
## Max.     3.0  0.51551910
```

3. Et si on retourne un nombre variable d'éléments ?

```
la <- lapply(x, FUN = function(elm) elm)
sa <- sapply(x, FUN = function(elm) elm)
# vapply pas pertinent

identical(la, sa)
```

```
## [1] TRUE
```

mapply

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,  
       USE.NAMES = TRUE)
```

- **FUN** : la fonction à appliquer
- **...** : ensemble d'arguments, vecteurs ou listes
- **MoreArgs** : liste d'arguments supplémentaires pour la fonction
- **SIMPLIFY** : booléen ou caractère, pour simplifier les résultats
- **USE.NAMES** : booléen. Si noms il y a dans X, les utiliser dans les résultats ?

```
mapply(rep, 1:2, 2:1)
```

```
## [[1]]  
## [1] 1 1  
##  
## [[2]]  
## [1] 2
```

```
# en nommant les arguments
```

```
mapply(rep, times = 1:2, x = 2:1)
```

```
## [[1]]  
## [1] 2  
##  
## [[2]]  
## [1] 1 1
```

```
# en passant des arguments supplémentaires
```

```
mapply(rep, times = 1:2, MoreArgs = list(x = 100))
```

```
## [[1]]  
## [1] 100  
##  
## [[2]]  
## [1] 100 100
```

```
# Avec simplification des résultats
```

```
mapply(function(n, moy) mean(rnorm(n, moy)), n = c(100, 1000), moy = c(10, 0))
```

```
## [1] 10.11457452 -0.02116763
```

tapply

```
tapply(X, INDEX, FUN = NULL, ...,  
       default = NA, simplify = TRUE)
```

- **X** : la colonne / le vecteur utilisé(e) dans le calcul
- **INDEX** : liste du / des facteurs définissant les populations
- **FUN** : la fonction à appliquer
- **...** : ensemble d'arguments supplémentaires, à passer à la fonction
- **simplify** : booléen ou caractère, pour simplifier les résultats

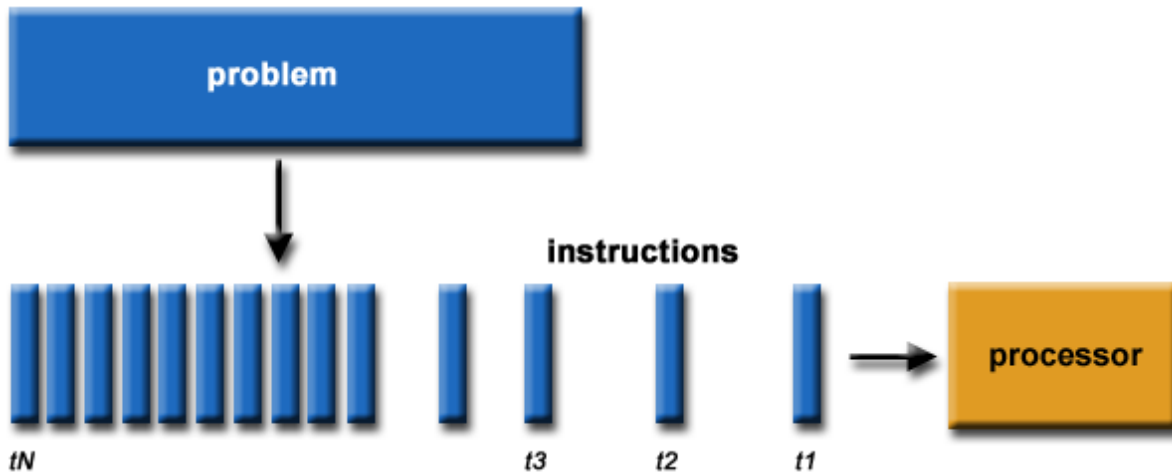


Figure 1:

```
head(warpbreaks, n = 4)
```

```
##   breaks wool tension
## 1    26    A      L
## 2    30    A      L
## 3    54    A      L
## 4    25    A      L
```

```
tapply(X = warpbreaks$breaks,
       INDEX = warpbreaks[, -1], FUN = sum)
```

```
##      tension
## wool   L   M   H
##   A 401 216 221
##   B 254 259 169
```

```
tapply(X = warpbreaks$breaks,
       INDEX = list(warpbreaks$wool), FUN = sum)
```

```
##   A   B
## 838 682
```

Le calcul parallèle

Concept

- Calcul séquentiel
 - un problème est divisé en une série d'instructions
 - les instructions sont exécutées les une après les autres
 - sur une seule unité de calcul
 - seulement une instruction s'exécute à la fois

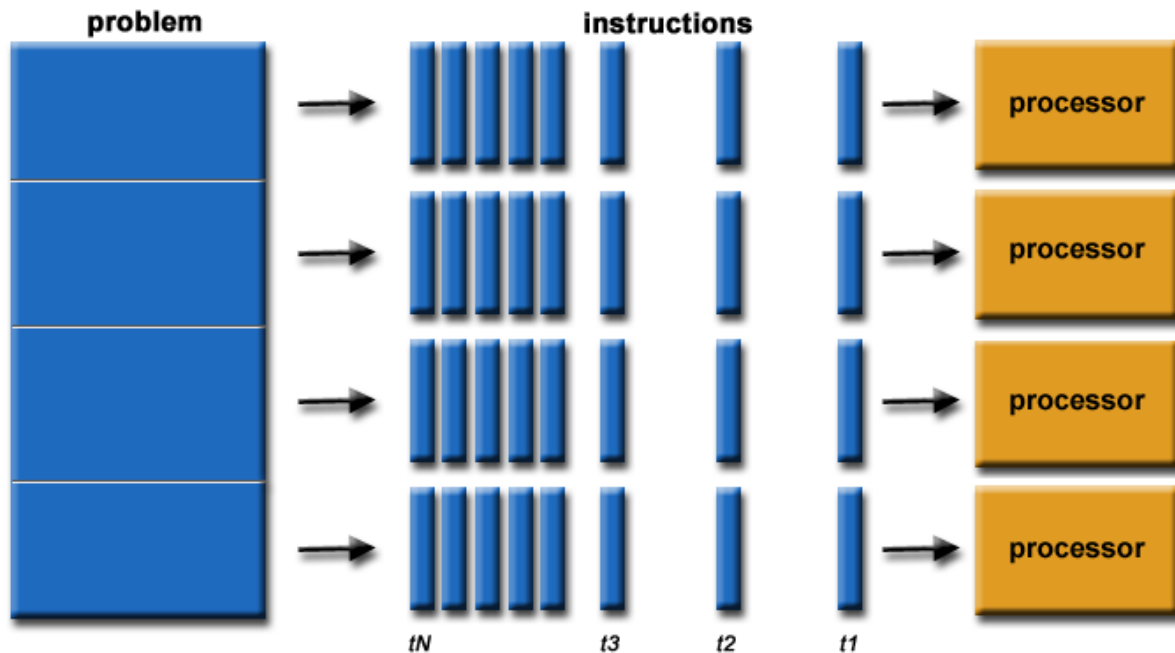


Figure 2:

- Calcul parallèle
 - un problème peut être divisé en plusieurs séries d'instructions indépendantes pouvant s'exécuter en même temps
 - les instructions de chaque série s'exécutent simultanément sur différentes unités de calcul
 - les résultats obtenus sur chaque unité de calcul sont renvoyés dans le processus parent
 - cela nécessite un mécanisme de contrôle et de synchronisation

'l'ensemble des techniques logicielles et matérielles permettant l'exécution simultanée de séquences d'instructions indépendantes sur des processeurs et/ou cœurs différents'

Le problème algorithmique est donc :

- pouvoir diviser tout ou une partie en sous-calculs indépendants
- pouvoir exécuter plusieurs instructions à un moment donné
- résoudre le problème en moins de temps qu'avec un calcul séquentiel

Les ressources matérielles à disposition :

- un unique ordinateur, avec plusieurs processeurs / cœurs
 - un cluster d'ordinateurs inter-connectés
-

Quand paralléliser ?

- quand chaque calcul commence à prendre un peu de temps...
- calculer plusieurs tâches rapides en parallèle prend en général plus de temps qu'avec un calcul séquentiel...
- faire attention au partage des données, et regarder l'évolution de la performance en fonction du nombre de cœurs
- le mieux : tester et comparer !



Figure 3:



Figure 4:

les outils dans R

- A la base, **R** est mono-cœur
- De nombreux packages permettant le calcul parallèle existent. Voir <https://cran.r-project.org/web/views/HighPerformanceComputing.html>

Nous nous focaliserons sur deux packages :

- le package **parallel**
 - inclu dans R depuis R.2.14.0
 - basé sur deux “anciens” packages : **snow** et **multicore**
 - propose une interface très proche de la ‘**Apply family**’
- le package **foreach**

```
require(parallel)
vignette("parallel")

require(foreach)
vignette("foreach")
```

le package parallel

Le processus général :

- ouverture d’un “cluster”
 - **makeCluster()**
 - ouverture de sessions **R** temporaires
 - fonction utile : **detectCores()**, nombre de CPU coeurs sur la machine
- utilisation du “cluster”
 - **clusterCall**, **clusterApply**, **clusterExport**, **clusterEvalQ**, ...
 - **parLapply**, **parSapply**, **parApply**, ...
- fermeture du “cluster”
 - sinon les sessions **R** temporaires restent ouvertes...
 - **stopCluster()**

exemple d’introduction

```
require(parallel)
nb.cores <- detectCores() # 4
nb.cores

## [1] 4

# mieux vaut éviter d'utiliser toutes les ressources
cl <- makeCluster(nb.cores - 1)
res <- clusterApply(cl, 1:7, function(x){ rnorm(x)})
str(res)

## List of 7
## $ : num 0.94
## $ : num [1:2] 0.373 -1.364
## $ : num [1:3] -1.328 0.435 0.643
```

```
## $ : num [1:4] -0.507 0.634 0.538 1.843
## $ : num [1:5] 2.113 -1.799 -0.633 1.033 0.561
## $ : num [1:6] 0.66 -0.416 -0.466 0.777 0.227 ...
## $ : num [1:7] -0.3585 0.9802 1.5405 0.0193 0.4211 ...
```

```
stopCluster(cl)
```

Points importants

chargements des données / packages

- les sessions **R** temporaires sont “vides” (sauf en Linux/Mac, avec l’option `makeCluster(, type=“FORK”)`)
 - aucunes variables / aucuns packages de la session principale sont présents
- **clusterExport** : exporte les variables / fonctions souhaitées
- **clusterEvalQ** : exécute un code dans toutes les sessions. Utile pour charger un package notamment

load-balancing

- Généralement, les p premiers calculs sont envoyés aux p sessions ouvertes
- les calculs suivants débutent lorsque **tous** les p calculs ont été effectués
- Dans le cas de calculs de temps différents, on perd de la performance
- des versions LB, **load-balancing**, existent pour enchaîner sur un nouveau calcul dès que le précédent se termine

Chargement des données : illustration

```
cl<-makeCluster(2)
add <- 10
mult <- function(x) x * 2

parLapply(cl, 1:10, function(x) mult(x) + add)

# les noeuds ne connaissent pas la variable et la fonction
# Error in checkForRemoteErrors(val) :
# 2 nodes produced errors; first error: objet 'mult' introuvable

# on les exporte avant de lancer le calcul

clusterExport(cl, varlist = c("add", "mult"))

res <- parLapply(cl, 1:10, function(x) mult(x) + add)

res[[1]] # 12

stopCluster(cl)
```


Chargement d'un package : illustration

```
cl<-makeCluster(2)
data(iris)

parLapply(cl, split(iris[, -c(5)], iris$Species), function(subdata){
  rpart(Sepal.Length~., subdata)
})

# les noeuds ne connaissent pas la variable et la fonction
# Error in checkForRemoteErrors(val) :
# 2 nodes produced errors; first error: impossible de trouver la fonction "rpart"

# on charge le package
clusterEvalQ(cl, {
  require(rpart)
})

res <- parLapply(cl, split(iris[, -c(5)], iris$Species), function(subdata){
  rpart(Sepal.Length~., subdata)
})

stopCluster(cl)
```

le package et la fonction foreach

- ressemble à une boucle **for**
- mais avec l'utilisation de l'opérateur **%do%** ou **%dopar%** pour du parallèle
- et **retourne un résultat**, une liste par défaut

```
require(foreach)

x <- foreach(i = 1:3) %do% sqrt(i)
# equivalent à lapply(1:3, sqrt)
x

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
```

structure du résultat : .combine

```
# un vecteur
x <- foreach(i = 1:3, .combine = "c") %do% sqrt(i)
x

## [1] 1.000000 1.414214 1.732051
```

```
# une matrice
x <- foreach(i=1:4, .combine = 'cbind') %do% rnorm(2)
x

##          result.1  result.2  result.3  result.4
## [1,] 0.09618023 -1.1134590  0.5752806 -1.7782451
## [2,] 0.46870827  0.5041963 -0.7396328  0.7502812

# une somme
x <- foreach(i = 1:3, .combine = "+") %do% i
x

## [1] 6
```

ajouter un filtre avant l'exécution

- Similaire à **if**, mais avec l'utilisation de **when**

```
require(numbers)
foreach(n = 1:50, .combine = c) %::% when (isPrime(n)) %do% n

## [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47
```

gérer les erreurs : **.errorhandling**

- par défaut, si une erreur se produit, l'exécution s'arrête
- on peut continuer le calcul et récupérer les erreurs potentielles en mettant l'option **.errorhandling** à *pass*

```
foreach(n = 1:2, .errorhandling = "pass") %do% ifelse(n == 2, stop("erreur"), n)

## [[1]]
## [1] 1
##
## [[2]]
## <simpleError in ifelse(n == 2, stop("erreur"), n): erreur>
```

Calcul parallèle

- même principe qu'avec **parallel**, sauf qu'il faut explicitement enregistrer le cluster
- avec **doParallel**, ou **doMC**, **doMPI**, **doRedis**, **doRNG**, **doSNOW**

```
require(foreach)
require(doParallel)

cl <- makeCluster(6)
# avec foreach, il faut enregistrer le cluster
registerDoParallel(cl)

res <- foreach(n = 1:6) %dopar% rnorm(x)
str(res)

## List of 6
## $ : num [1:6] -1.456 1.553 -0.402 -1.776 -0.362 ...
## $ : num [1:6] -0.3617 0.0426 -0.2722 0.2796 -1.6234 ...
```

```
## $ : num [1:6] 1.284 -0.493 -2.037 0.417 0.331 ...
## $ : num [1:6] -1.3211 -0.9591 -0.0182 0.6178 0.2576 ...
## $ : num [1:6] -0.523 -0.638 -1.079 0.574 0.307 ...
## $ : num [1:6] 1.2269 0.3766 1.1712 0.1639 0.0746 ...
```

```
stopCluster(cl)
```

Points importants

chargements des données / packages

- contrairement à l'utilisation du package **parallel**, toutes les variables de l'environnement courant sont exportées par défaut
- **.noexport** : ne pas exporter certaines variables
- **.export** : exporter des variables qui ne sont pas dans l'environnement courant
- **.packages** : chargement de package(s)

autres options utiles

- **.inorder** : résultats dans l'ordre d'entrée ? Défaut à **TRUE**. **FALSE** peut amener de meilleures performances
 - **.verbose** : utile pour déboguer
-

Exemples

```
cl<-makeCluster(2)
registerDoParallel(cl)

#####
# chargement des données
#####

add <- 10
mult <- function(x) x * 2

# pas besoin de charger les données avant
res <- foreach(n = 1:10) %dopar% (mult(n) + add)
res[[1]] # 12

#####
# chargement d'un package
#####

res <- foreach(data = split(iris[, -c(5)], iris$Species), .packages = "rpart") %dopar%
  rpart(Sepal.Length~., data)

stopCluster(cl)
```

Retour sur la notion d'environnement

```
y <- 10
f <- function(x, .export = NULL){
  cl<-makeCluster(2)
  registerDoParallel(cl)
  res <- foreach(i = x, .export = .export) %dopar% (i + y)
  stopCluster(cl)
  res
}

res <- f(2:10)
# Error in (x + y) : task 1 failed - "objet 'y' introuvable"

res <- f(2:10, .export = "y")
res[[1]] # 12
```

Monitoring / Profiling / Performance / Debug

Temps de calculs

Pour monitorer le temps de calculs, la fonction `system.time()` peut-être utilisée, mais le package `microbenchmark` permet de monitorer avec précision et en répétant les opérations.

```
library(microbenchmark)
x <- runif(1000)
microbenchmark(sqrt(x), x^{1/2}, times=1000)

## Unit: microseconds
##      expr   min    lq   mean median    uq   max neval
##  sqrt(x)  4.6   6.9  8.2246    7.0   8.6  63.4   1000
##  x^{      1/2 } 35.3 37.7 42.2016   38.4 41.9 149.4   1000
```

Profilage du code

Utiliser la fonction `Rprof` qui procède par échantillonnage : elle stoppe l'exécution du code par intervalles (`interval`) et différencie le temps de calcul réalisé par chaque fonction (`self.time`) et le temps global (`total.time`).

```
is.prime <- function(n){
  n == 2L || all(n %% 2L:ceiling(sqrt(n)) != 0)
}

all.prime <- function(n){
  v <- integer(0)
  for(i in 2:n){
    if(is.prime(i)){
      v <- c(v,i)
    }
  }
  v
}
```

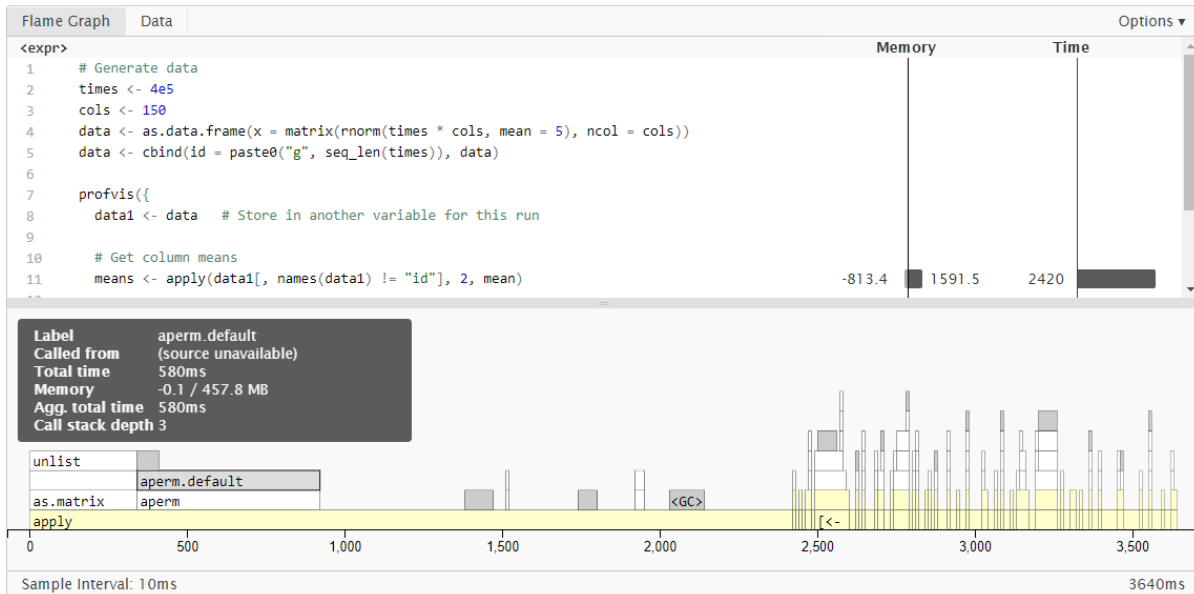


Figure 5:

```
Rprof("Rprof.out", interval = 0.001)
prime.number <- all.prime(100000)
Rprof(NULL)
```

Profilage du code

```
summaryRprof("Rprof.out")
```

```
##          self.time self.pct total.time total.pct
## "is.prime"      0.071   47.97      0.109    73.65
## "%%"           0.033   22.30      0.033    22.30
## "c"             0.029   19.59      0.029    19.59
## "all.prime"     0.010    6.76      0.148   100.00
## "all"           0.005    3.38      0.005    3.38

##          total.time total.pct self.time self.pct
## "all.prime"     0.148   100.00      0.010    6.76
## "is.prime"      0.109    73.65      0.071   47.97
## "%%"           0.033   22.30      0.033   22.30
## "c"             0.029   19.59      0.029   19.59
## "all"           0.005    3.38      0.005    3.38
```

Profilage du code

D'autres outils existent, avec notamment les packages **proftools** ou **profvis**

<https://rstudio.github.io/profvis/>

<https://cran.r-project.org/web/packages/proftools/vignettes/proftools.pdf>

Impact mémoire

- Dans **R** de base, avec la fonction `object.size()`. **Problème** : ne prend pas en compte toute la complexité potentielle des objets (environnements rattachés)
- Avec le package **pryr**
 - `object_size()`
 - `mem_used()` : mémoire utilisée, `mem_change(code)` : impact du code sur la mémoire

```
# différence integer / numeric
v_int <- rep(1L, 1e8) ; v_num <- rep(1, 1e8)
object_size(v_int); object_size(v_num)
```

```
## 400 MB
```

```
## 800 MB
```

```
mem_change(x <- 1:1e6) ; mem_change(rm(x))
```

```
## -2.66 kB
```

```
## 592 B
```

Accélérer son code

R non efficace pour interpréter et exécuter des boucles for et donc A EVITER!

- Vectorisation
- Fonctions de type **Apply**
- Utilisation du package **compiler** :

<http://homepage.divms.uiowa.edu/~luke/R/compiler/compiler.pdf>

- Implémenter les points chauds de calcul avec des langages compilés et utiliser le package **Rcpp**

<http://www.rcpp.org/>

Accélérer son code

Initialiser l'espace pour un résultat. Sinon **R** prend du temps pour agrandir itérativement la mémoire allouée à un objet :

```
x <- rnorm(1000000) ; y <- rnorm(1000000)
res <- integer(1000000)
```

```
# calcul de la somme via une boucle avec initialisation
system.time(for(i in 1:length(x)){
  res[i] <- x[i] + y[i]
})
```

```
##   user  system elapsed
##  0.11    0.00    0.11
```

```
res <- c()
# sans initialisation
system.time(for(i in 1:length(x)){
  res[i] <- x[i] + y[i]
})
```

```
> f(10)
Error in "a" + d : non-numeric argument to binary operator
4 i(c) at exceptions-example.R#3
3 h(b) at exceptions-example.R#2
2 g(a) at exceptions-example.R#1
1 f(10)

traceback()
# 4: i(c) at exceptions-example.R#3
# 3: h(b) at exceptions-example.R#2
# 2: g(a) at exceptions-example.R#1
# 1: f(10)
```

Figure 6:

```
##      user  system elapsed
##    0.26    0.00    0.27
```

Un petit mot sur le débogage

- Pour voir simplement les informations : utilisation de *print()* dans la fonction
- Quand une erreur se produit, information du **traceback**
 - Disponible par défaut dans la console RStudio
 - via la fonction *traceback()* dans R
- Utilisation de la fonction *browser()* n'importe où dans le code : elle stoppe l'exécution et lance un environnement dans lequel on peut accéder aux variables actuelles et continuer l'exécution
- Dans **RStudio**, insertion de points d'arrêt (menu *Debug*)

Plus d'informations ici : <https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio>

Aller plus loin...!

- The R Manuals : <https://cran.r-project.org/manuals.html>
- R Contributed Documentation : <https://cran.r-project.org/other-docs.html>
- Advanced R by Hadley Wickham : <http://adv-r.had.co.nz/>
- R packages by Hadley Wickham : <http://r-pkgs.had.co.nz/>
- Tests using testthat : <http://r-pkgs.had.co.nz/tests.html>
- Code coverage with covr : <https://github.com/r-lib/covr>
- How-to go parallel in R - basics + tips : <http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/>
- State of the Art in Parallel Computing withR : <http://www.jstatsoft.org/v31/i01/paper>
- R tutorial on the Apply family of functions : <http://www.r-bloggers.com/r-tutorial-on-the-apply-family-of-functions/>
- A Tutorial on Loops in R - Usage and Alternatives : <http://blog.datacamp.com/tutorial-on-loops-in-r/>