

ISV51: Programmation sous R

Développer sous R

L3 GBI – Université d'Evry

semestre d'automne 2015

http://julien.cremeriefamily.info/teachings_L3BI_ISV51.html

Plan

Programmer en R

Accélérer son code

Plan

Programmer en R

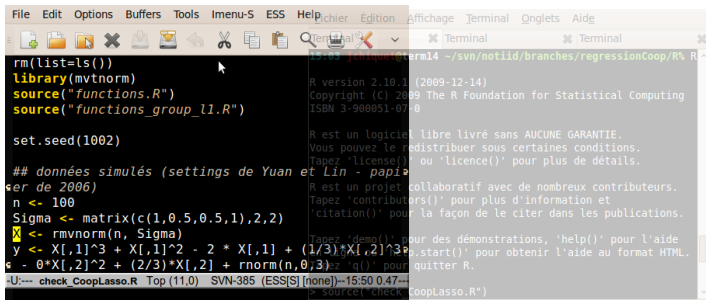
- Structures de contrôle

- Les fonctions

Accélérer son code

Environnement de développement

Première solution



The screenshot shows an R development environment with a script editor on the left and a terminal window on the right. The script editor contains the following R code:

```
rm(list=ls())
library(mvtnorm)
source("fonctions.R")
source("fonctions_group_11.R")

set.seed(1002)

## données simulées (settings de Yuan et Lin - papier de 2006)
n <- 100
Sigma <- matrix(c(1,0.5,0.5,1),2,2)
X <- rmvnorm(n, Sigma)
y <- X[,1]^3 + X[,1]^2 - 2 * X[,1] + (1/3)*X[,2]^3 - 0*X[,2]^2 + (2/3)*X[,2] + rnorm(n,0,3)
```

The terminal window shows the R version 2.10.1 (2009-12-14) and the R Foundation for Statistical Computing logo. It also displays the R license text:

```
R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.
```

1. un éditeur de texte (vos fonctions / scripts)
2. un terminal avec R (tester, « sourcer »)

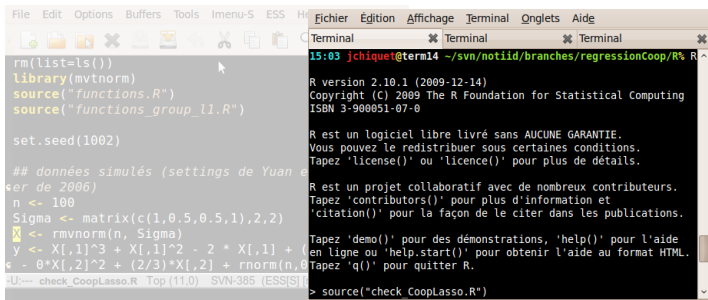
« Sourcer »

`source("un_script.R")` : exécute la série de commandes de `mon_script.R`

`source("mes_fonctions.R")` : charge le contenu (les fonctions) de `mes_fonctions.R`

Environnement de développement

Première solution



The screenshot shows an R development environment with two windows. The left window is a script editor containing R code. The right window is a terminal window showing the R startup sequence and the execution of the script.

```
rm(list=ls())
library(mvtnorm)
source("fonctions.R")
source("fonctions_group_11.R")

set.seed(1002)

## données simulés (settings de Yuan et al. 2006)
n <- 100
Sigma <- matrix(c(1,0.5,0.5,1),2,2)
X <- rmvnorm(n, Sigma)
y <- X[,1]^3 + X[,1]^2 - 2 * X[,1] + 0.5 * X[,2]^2 + (2/3)*X[,2] + rnorm(n, 0, 1)

# check CoopLasso.R
source("check_CoopLasso.R")
```

```
R version 2.10.1 (2009-12-14)
Copyright (C) 2009 The Foundation for Statistical Computing
ISBN 3-900051-07-0

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

> source("check_CoopLasso.R")
```

1. un éditeur de texte (vos fonctions / scripts)
2. un terminal avec R (tester, « sourcer »)

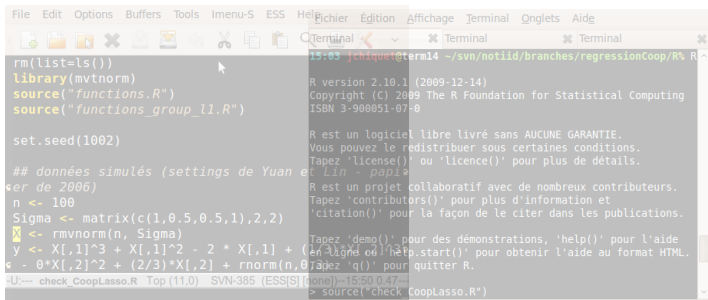
« Sourcer »

`source("un_script.R")` : exécute la série de commandes de `mon_script.R`

`source("mes_fonctions.R")` : charge le contenu (les fonctions) de `mes_fonctions.R`

Environnement de développement

Première solution



The screenshot shows an R development environment with two windows. The left window is a script editor containing R code for generating simulated data and fitting a model. The right window is a terminal showing the R version and copyright information, along with a prompt to source a file.

```
rm(list=ls())
library(mvtnorm)
source("fonctions.R")
source("fonctions_group_ll.R")

set.seed(1002)

## données simulées (settings de Yuan et al. 2006)
n <- 100
Sigma <- matrix(c(1,0.5,0.5,1),2,2)
X <- rmvnorm(n, Sigma)
y <- X[,1]^3 + X[,1]^2 - 2 * X[,1] + (en ligne ou) help.start() pour obtenir l'aide au format HTML.
  - 0*X[,2]^2 + (2/3)*X[,2] + rnorm(n,0)
-U-- check_CoopLasso.R Top(11.0) SVN-385 (ESS) [done] --15:50 0.47--
> source("check_CoopLasso.R")
```

1. un éditeur de texte (vos fonctions / scripts)
2. un terminal avec R (tester, « sourcer »)

« Sourcer »

`source("un_script.R")` : exécute la série de commandes de `mon_script.R`

`source("mes_fonctions.R")` : charge le contenu (les fonctions) de `mes_fonctions.R`

Environnement de développement

R-studio, environnement de travail intégré

The screenshot displays the RStudio integrated development environment. The main editor window shows a script with the following R code:

```
1 k <- 2+2
2 y <- 3+5
3
```

The console window at the bottom left shows the execution of the script, outputting the values of the variables:

```
1:1 (Top Level) >
R Script >
--Documents\Teachings\2015-2016\L3_GBI\ISV51\td1-ivs1 - RStudio />
en ligne ou "help.start()" pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

WARNING: Your CRAN mirror is set to "http://cran.rstudio.com/" which has an insecure (non-HTTPS) URL. The repository was
s likely specified in .Rprofile or Rprofile.site so if you wish to change it you may need to edit one of those files. You
ou should either switch to a repository that supports HTTPS or change your RStudio options to not require HTTPS downloa
ds.

To learn more and/or disable this warning message see the "Use secure download method for HTTP" option in Tools -> Globa
al Options -> Packages.

> 2
> 2+2
[1] 4
> x <- 2+2
> y <- 3+5
> |
```

The Environment pane on the right shows the current state of the workspace:

Values	
x	4
y	8

The Packages pane at the bottom right lists installed and available packages:

Name	Description	Version
<input type="checkbox"/> acepack	ace() and avar() for selecting regression transformations	1.3-3.3
<input type="checkbox"/> aricode	Compute rand index	2015.06.12
<input type="checkbox"/> BiocInstaller	Install/Update Bioconductor and CRAN Packages	1.18.2
<input type="checkbox"/> biotools	Tools for Biometry and Applied Statistics in Agricultural Science	2.1
<input type="checkbox"/> bitops	Bitwise Operations	1.0-6
<input type="checkbox"/> blockseg	Two dimensional change-points detection	1.0
<input type="checkbox"/> blocseg	Two dimensional change-points detection	1.0
<input type="checkbox"/> car	Companion to Applied Regression	2.0-25
<input type="checkbox"/> caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.17.1
<input type="checkbox"/> cgdscr	R-Based API for accessing the MSKCC Cancer Genomics Data Server (CGDS).	1.1.33
<input type="checkbox"/> clusterpath	Fast agglomerative convex clustering, non-Rcpp implementation	1.2
<input type="checkbox"/> colorspace	Color Space Manipulation	1.2-6
<input type="checkbox"/> crayon	Colored Terminal Output	1.2.1
<input type="checkbox"/> dichromat	Color Schemes for Dichromats	2.0-0
<input type="checkbox"/> digest	Create Cryptographic Hash Digests of R Objects	0.6-8
<input type="checkbox"/> doParallel	Foreach parallel adaptor for the parallel package	1.0.8
<input type="checkbox"/> ellipse	Functions for drawing ellipses and ellipse-like confidence	0.3-8

Plan

Programmer en R

Structures de contrôle

Les fonctions

Accélérer son code

Regrouper les expressions

Syntaxe

```
{expr_1; expr_2; ...; expr_n }
```

ou

```
{  
  expr_1  
  ...  
  expr_n  
}
```

Remarques sur les groupes

- ▶ La dernière valeur du groupe est retournée ;
- ▶ un groupe d'expressions peut être passé à une fonction, réutilisé dans une expression plus grande, etc.

Regrouper les expressions

Exemples

```
expr1 <- {a<-3; b<-5; a*b}
```

```
expr1
```

```
## [1] 15
```

```
tmp <- 12
```

```
expr2 <- {a<-3; b<-5; tmp<-a*b+tmp}
```

```
expr2
```

```
## [1] 27
```

```
tmp
```

```
## [1] 27
```

Exécution conditionnelle : if,if/else,ifelse

Syntaxe

```
if (condition) {  
    expr_1  
} else {  
    expr_2  
}
```

ou (forme vectorielle)

```
ifelse(condition, a, b)
```

Remarques

- ▶ `condition` est une valeur logique : penser à `&`, `|`, `!`, ...
- ▶ le `else` est optionnel,
- ▶ `elseif` permet d'imbriquer les conditionnements.

Exécution conditionnelle : if,if/else,ifelse

Exemples

```
partiel <- 11
DS <- 14
if (partiel > 6 & mean(DS,partiel) >10) {
  cat("\nreçu(e).")
} else {
  cat("\nrecalé(e).")
}

##
## reçu(e).
```

Fonctionnement en vectoriel de ifelse

```
partiel <- c(11,5,6,12,9,8,14)
DS <- c(14,16,12,12,19,12,7)
ifelse(partiel > 6 & rowMeans(cbind(DS,partiel)) >10, "reçu", "recalé")

## [1] "reçu" "recalé" "recalé" "reçu" "reçu" "recalé" "reçu"
```

Exécution conditionnelle : switch

Syntaxe

```
switch (expr,  
    expr.1 = faire.1,  
    expr.2 = faire.2,  
    ...,  
    faire.default  
)
```

Remarques

- ▶ `expr` est une variable contenant une chaîne de caractère ou un entier.
- ▶ Si `expr` est un entier i , la i^{e} expression `faire.i` est évalué et renvoyée.
- ▶ Si `expr` contient une chaîne, l'expression `faire.i` telle que `expr == expr.1` est évaluée.

Exécution conditionnelle : switch

Exemples

Avec un entier

```
expr <- 2  
switch(expr, cat("je vaux 1"), cat("je vaux 2"))
```

```
## je vaux 2
```

```
expr <- 3  
switch(expr, cat("je vaux 1"), cat("je vaux 2"))
```

Avec une chaîne

```
stat <- "variance"  
ma.fonction <- switch(stat,  
  "moyenne" = mean,  
  "variance" = var, NULL)  
ma.fonction(1:10)
```

```
## [1] 9.166667
```

Exécution répétée : boucle for

Syntaxe

```
for (var in set) {  
    expr(var)  
}
```

ou

```
for (var in set)  
    expr(var)
```

à fuir pour éviter les effets de bords sournois !

Remarques sur la boucle for

- ▶ `var` est la variable incrémentée,
- ▶ `set` est un vecteur définissant les valeurs successives,
- ▶ *lente* comparée aux opérateurs matriciels.

Exécution répétée : boucle for I

Exemples

```
for (i in sample(1:5)) {  
  cat(i)  
}
```

```
## 35142
```

```
v <- numeric(7)  
for (i in seq_along(v)) {  
  v[i] <- i*3  
}  
v
```

```
## [1] 3 6 9 12 15 18 21
```


Exécution répétée : boucle for II

Exemples

```
data(iris)
cat("\nNoms des colonnes:")

##
## Noms des colonnes:

for (nom in colnames(iris)) {
  cat("",nom)
}

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

Exécution répétée : boucles `while` et `repeat`

Syntaxe

```
while (condition) {  
    expr  
}
```

ou

```
repeat {  
    expr  
}
```

Remarque

- ▶ Comme pour `for`, éviter les imbrications sources de lenteur.

Contrôle des boucles : break, next

Exemples d'utilisation

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

ou

```
while (condition1){  
  expr_1  
  if (condition2) {next}  
  expr_2  
}
```

Remarque

- break est la seule manière d'interrompre une boucle repeat.

Exécution répétée : while

Exemples

Parcours des lignes du tableau `iris` tant qu'on a pas rencontré un individu ayant certaines caractéristiques.

```
data(iris)
i <- 0 ## numéro individu courant
cond <- FALSE
while (!cond) {
  i <- i + 1
  if (iris[i, ]$Sepal.Length > 6)
    cond <- TRUE
}
cat("\nL individu", i, "est le premier à avoir une longueur de sépale > 6.")

##
## L individu 51 est le premier à avoir une longueur de sépale > 6.
```

Plan

Programmer en R

Structures de contrôle

Les fonctions

Accélérer son code

Définir une fonction

Syntaxe

```
nom_de_la_fonction <- function(arg1,arg2, ...) {  
  expression  
  
  return(var)  
}
```

Remarques

- ▶ `return` peut être omis (à éviter) : dans ce cas la dernière valeur calculée est renvoyée.
- ▶ peut être tapée directement dans l'interpréteur ou dans un fichier externe `fonctions.R`, chargé par source.

Un exemple simple

Moyenne empirique d'un vecteur

Avec suppression des valeurs manquantes.

```
moyenne <- function(x) {  
  x.not.na <- x[!is.na(x)]  
  ## moyenne empirique  
  resultat <- sum(x.not.na) / length(x.not.na)  
  
  return(resultat)  
}
```

Tests

```
moyenne(rnorm(100))
```

```
## [1] 0.002743981
```

```
moyenne(c(1,-5,3,NA,8.7))
```

```
## [1] 1.925
```

Les arguments, leurs valeurs par défaut

Propriétés

- ▶ les arguments peuvent être passés dans le **désordre** s'ils sont **nommés** : `var=object`,
- ▶ on peut définir une valeur par défaut pour n'importe quel argument lors de la définition de la fonction : `var=10`.
- ▶ en cas de **sorties multiples**, les sorties doivent être renvoyées sous forme de liste.

Remarques

- ▶ Les valeurs par défaut rendent la lecture des fonctions beaucoup plus aisée pour l'utilisateur : **imposer peu d'arguments obligatoires**.
- ▶ Les noms des éléments de la liste définie dans la fonction sont conservés à l'extérieur de la fonction.

Les arguments, leurs valeurs par défaut

Propriétés

- ▶ les arguments peuvent être passés dans le **désordre** s'ils sont **nommés** : `var=object`,
- ▶ on peut définir une valeur par défaut pour n'importe quel argument lors de la définition de la fonction : `var=10`.
- ▶ en cas de **sorties multiples**, les sorties doivent être renvoyées sous forme de liste.

Remarques

- ▶ Les valeurs par défaut rendent la lecture des fonctions beaucoup plus aisée pour l'utilisateur : **imposer peu d'arguments obligatoires**.
- ▶ Les noms des éléments de la liste définie dans la fonction sont conservés à l'extérieur de la fonction.

Les arguments, leurs valeurs par défaut

Propriétés

- ▶ les arguments peuvent être passés dans le **désordre** s'ils sont **nommés** : `var=object`,
- ▶ on peut définir une valeur par défaut pour n'importe quel argument lors de la définition de la fonction : `var=10`.
- ▶ en cas de **sorties multiples**, les sorties doivent être renvoyées sous forme de liste.

Remarques

- ▶ Les valeurs par défaut rendent la lecture des fonctions beaucoup plus aisée pour l'utilisateur : **imposer peu d'arguments obligatoires**.
- ▶ Les noms des éléments de la liste définie dans la fonction sont conservés à l'extérieur de la fonction.

Les arguments, leurs valeurs par défaut

Propriétés

- ▶ les arguments peuvent être passés dans le **désordre** s'ils sont **nommés** : `var=object`,
- ▶ on peut définir une valeur par défaut pour n'importe quel argument lors de la définition de la fonction : `var=10`.
- ▶ en cas de **sorties multiples**, les sorties doivent être renvoyées sous forme de liste.

Remarques

- ▶ Les valeurs par défaut rendent la lecture des fonctions beaucoup plus aisée pour l'utilisateur : **imposer peu d'arguments obligatoires**.
- ▶ Les noms des éléments de la liste définie dans la fonction sont conservés à l'extérieur de la fonction.

Un exemple (un tout petit peu) plus avancé

Résumé numérique d'un vecteur

```
resume <- function(x,na.rm=TRUE,affiche=FALSE) {  
  mu <- mean(x,na.rm=na.rm)  
  s2 <- var(x,na.rm=na.rm)  
  if (affiche) {  
    cat("\nMoyenne:",mu,"et variance:",s2)  
  }  
  return(list(moyenne = mu, variance = s2))  
}
```

```
out <- resume(rnorm(100))  
out$variance  
  
## [1] 1.009893  
  
out <- resume(affiche=TRUE,x=rexp(100,0.5))  
  
##  
## Moyenne: 2.229719 et variance: 4.20312
```

Fonction anonyme

Définition

Il s'agit d'une fonction qui ne porte pas de nom.

↪ Elle est définie « à la volée » au moment de son utilisation.

Utilisation

- ▶ couplée à une fonction type `xapply`,
- ▶ généralement courte,
- ▶ possède peu d'arguments,
- ▶ fait sens "localement", dans un contexte particulier du programme.

Fonction anonyme : exemples

Utilisation avec tapply

Dans les données iris, on cherche la somme des carrés de la longueur de sépale pour chaque espèce.

```
with(iris, tapply(Sepal.Length, Species, function(x) {return(sum(x^2))}))  
##      setosa versicolor  virginica  
## 1259.09    1774.86    2189.90
```

Utilisation avec apply

On veut calculer la moyenne de chaque ligne de la matrice A en ne conservant que les valeurs entre 0 et 1.

```
A <- matrix(rnorm(100*200), 100, 200)  
head(apply(A, 1, function(x) {sum(x[x>0 & x<1])}))  
## [1] 25.44734 26.36134 31.44209 32.89236 30.05999 35.07290
```

Fonction anonyme : exemples II

Utilisation avec sapply/lapply

On veut estimer la distribution discrétiser dans chaque colonne de iris.

```
lapply(iris, function(x) { ## x est la colonne courante
  if(is.factor(x)) {
    return(table(x))
  } else {
    return(table(cut(x, seq(min(x), max(x), len=5))))
  }
})
```

```
## $Sepal.Length
```

```
##
## (4.3,5.2] (5.2,6.1] (6.1,7] (7,7.9]
##      44      50      43      12
##
```

```
## $Sepal.Width
```

```
##
## (2,2.6] (2.6,3.2] (3.2,3.8] (3.8,4.4]
##      23      83      37      6
##
```

```
## $Petal.Length
```

```
##
## (1,2.48] (2.48,3.95] (3.95,5.43] (5.43,6.9]
##      49      11      61      28
##
```

```
## $Petal.Width
```

```
##
## (0.1,0.7] (0.7,1.3] (1.3,1.9] (1.9,2.5]
##      45      28      43      29
##
```

```
## $Species
```

Plan

Programmer en R

Accélérer son code

- Analyse du code

- Vectorisation

- Privilégier les objets simples

- Intégration de code C/C++

Plan

Programmer en R

Accélérer son code

Analyse du code

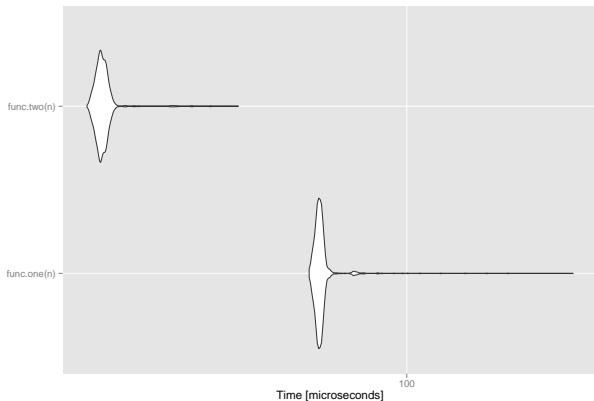
Vectorisation

Privilégier les objets simples

Intégration de code C/C++

How to quickly benchmark your code

```
func.one <- function(n) {return(rnorm(n,0,1))}  
func.two <- function(n) {return(rpois(n,1))}  
  
library(microbenchmark)  
n <- 1000  
res <- microbenchmark(func.one(n), func.two(n), times=1000)  
autoplot(res)
```



How to profile your code I

Suppose you want to evaluate which part of the following function is hot :

```
## generate data, center/scale and perform ridge regression
my.func <- function(n,p) {

  require(MASS)

  ## draw data
  x <- matrix(rnorm(n*p),n,p)
  y <- rnorm(n)

  ## center/scale
  xs <- scale(x)
  ys <- y-mean(y)

  ## return ridge's coefficients
  ridge <- lm.ridge(y~x+0,lambda=1)

  return(ridge$coef)
}
```

How to profile your code II

One can rely on the default `Rprof` function, with somewhat technical outputs

```
Rprof(file="profiling.out", interval=0.05)
res <- my.func(1000,500)
Rprof(NULL)
```

```
summaryRprof("profiling.out")$by.self
```

##	self.time	self.pct	total.time	total.pct
## "La.svd"	0.90	81.82	0.90	81.82
## "lm.ridge"	0.05	4.55	1.05	95.45
## "svd"	0.05	4.55	0.95	86.36
## "%*%"	0.05	4.55	0.05	4.55
## "aperm.default"	0.05	4.55	0.05	4.55

How to profile your code III

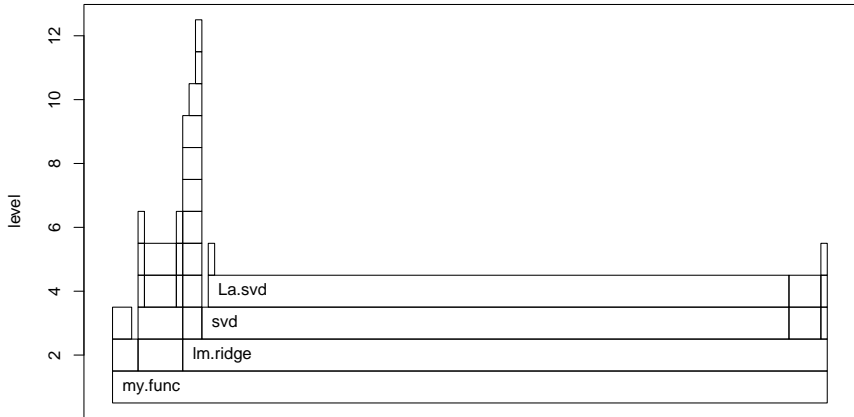
```
summaryRprof("profiling.out")$by.total
```

##	total.time	total.pct	self.time	self.pct
## "<Anonymous>"	1.10	100.00	0.00	0.00
## "block_exec"	1.10	100.00	0.00	0.00
## "call_block"	1.10	100.00	0.00	0.00
## "doTryCatch"	1.10	100.00	0.00	0.00
## "eval"	1.10	100.00	0.00	0.00
## "evaluate_call"	1.10	100.00	0.00	0.00
## "FUN"	1.10	100.00	0.00	0.00
## "handle"	1.10	100.00	0.00	0.00
## "in_dir"	1.10	100.00	0.00	0.00
## "knit"	1.10	100.00	0.00	0.00
## "lapply"	1.10	100.00	0.00	0.00
## "my.func"	1.10	100.00	0.00	0.00
## "process_file"	1.10	100.00	0.00	0.00
## "process_group"	1.10	100.00	0.00	0.00
## "process_group.block"	1.10	100.00	0.00	0.00
## "try"	1.10	100.00	0.00	0.00
## "tryCatch"	1.10	100.00	0.00	0.00
## "tryCatchList"	1.10	100.00	0.00	0.00
## "tryCatchOne"	1.10	100.00	0.00	0.00
## "withCallingHandlers"	1.10	100.00	0.00	0.00
## "withVisible"	1.10	100.00	0.00	0.00
## "lm.ridge"	1.05	95.45	0.05	4.55
## "svd"	0.95	86.36	0.05	4.55
## "La.svd"	0.90	81.82	0.90	81.82
## "%*%"	0.05	4.55	0.05	4.55
## "aperm.default"	0.05	4.55	0.05	4.55
## "aperm"	0.05	4.55	0.00	0.00
## "apply"	0.05	4.55	0.00	0.00
## "scale"	0.05	4.55	0.00	0.00
## "scale.default"	0.05	4.55	0.00	0.00

How to profile your code III

The profr package is maybe a little easier to understand...

```
library(profr)
profiling <- profr({my.func(1000,500)}, interval=0.01)
plot(profiling)
```



Plan

Programmer en R

Accélérer son code

Analyse du code

Vectorisation

Privilégier les objets simples

Intégration de code C/C++

Use the vector capabilities of R

Any algebraic operation should be thought in a “vectorized” way

```
exp2.1 <- sum(2^(0:10)/c(1,cumprod(1:10))) ## good
exp2.2 <- 1
for(k in 1:10) ## bad
  exp2.2 <- exp2.2 + 2^k/factorial(k)
```

Even non-algebraic operation should be thought as algebraic :

```
outer(1:4,c("A","B","C","D"),FUN=paste,sep="-")
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "1-A" "1-B" "1-C" "1-D"
## [2,] "2-A" "2-B" "2-C" "2-D"
## [3,] "3-A" "3-B" "3-C" "3-D"
## [4,] "4-A" "4-B" "4-C" "4-D"
```

Use the [a-z]*apply family, especially with factor (e.g. tapply)

```
data <- rnorm(100)
sexe <- factor(sample(c("H","F"),100,rep=TRUE))
mean.1 <- tapply(data, sexe, mean) ## good
mean.2 <- c() ## complicated
for (l in levels(sexe))
  mean.2 <- c(mean.2, mean(data[sexe == l]))
```


Use the vector capabilities of R

Any algebraic operation should be thought in a “vectorized” way

```
exp2.1 <- sum(2^(0:10)/c(1,cumprod(1:10))) ## good
exp2.2 <- 1
for(k in 1:10) ## bad
  exp2.2 <- exp2.2 + 2^k/factorial(k)
```

Even non-algebraic operation should be thought as algebraic :

```
outer(1:4,c("A","B","C","D"),FUN=paste,sep="-")

##      [,1] [,2] [,3] [,4]
## [1,] "1-A" "1-B" "1-C" "1-D"
## [2,] "2-A" "2-B" "2-C" "2-D"
## [3,] "3-A" "3-B" "3-C" "3-D"
## [4,] "4-A" "4-B" "4-C" "4-D"
```

Use the [a-z]*apply family, especially with factor (e.g. tapply)

```
data <- rnorm(100)
sexe <- factor(sample(c("H","F"),100,rep=TRUE))
mean.1 <- tapply(data, sexe, mean) ## good
mean.2 <- c() ## complicated
for (l in levels(sexe))
  mean.2 <- c(mean.2, mean(data[sexe == l]))
```

Use the vector capabilities of R

Any algebraic operation should be thought in a “vectorized” way

```
exp2.1 <- sum(2^(0:10)/c(1,cumprod(1:10))) ## good
exp2.2 <- 1
for(k in 1:10) ## bad
  exp2.2 <- exp2.2 + 2^k/factorial(k)
```

Even non-algebraic operation should be thought as algebraic :

```
outer(1:4,c("A","B","C","D"),FUN=paste,sep="-")

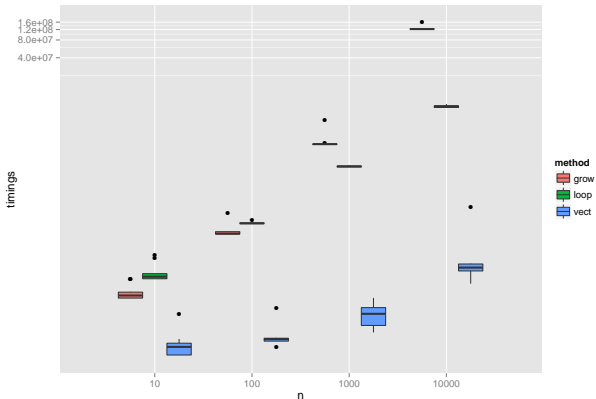
##      [,1] [,2] [,3] [,4]
## [1,] "1-A" "1-B" "1-C" "1-D"
## [2,] "2-A" "2-B" "2-C" "2-D"
## [3,] "3-A" "3-B" "3-C" "3-D"
## [4,] "4-A" "4-B" "4-C" "4-D"
```

Use the [a-z]*apply family, especially with factor (e.g. tapply)

```
data <- rnorm(100)
sexe <- factor(sample(c("H","F"),100,rep=TRUE))
mean.1 <- tapply(data, sexe, mean) ## good
mean.2 <- c() ## complicated
for (l in levels(sexe))
  mean.2 <- c(mean.2, mean(data[sexe == l]))
```

Preallocate whenever it is possible

```
grow <- function(n) {vec <- numeric(0); for (i in 1:n) vec <- c(vec,i)}  
loop <- function(n) {vec <- numeric(n); for (i in 1:n) vec[i] <- i}  
vect <- function(n) {1:n}
```



Do not stack objects I

Even if it is tempting when the final size is unknown.

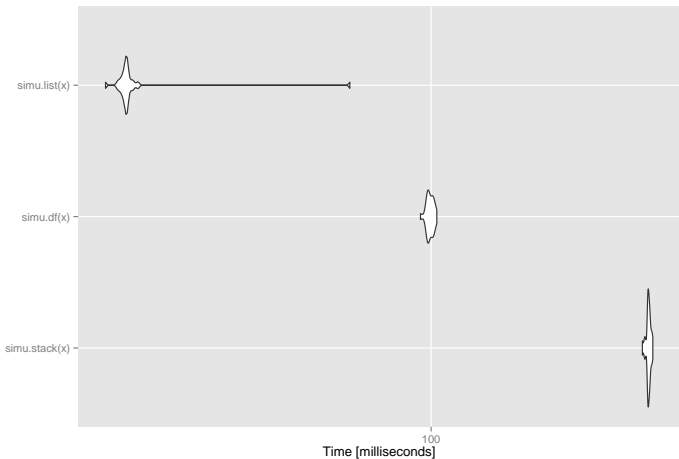
```
simu.stack <- function(x) { ## x is a n x p matrix
  out <- data.frame(mean = numeric(0), sd = numeric(0))
  for (i in 1:n)
    out <- rbind(out, data.frame(mean = mean(x[i,]), sd = sd(x[i, ])) )
  return(out)
}

simu.df <- function(x) {
  out <- data.frame(mean = numeric(n), sd = numeric(n))
  for (i in 1:n)
    out[i, ] <- c(mean = mean(x[i,]), sd = sd(x[i, ]))
  return(out)
}

simu.list <- function(x) {
  my.list <- lapply(1:n, function(i) c(mean(x[i,]), sd(x[i, ])))
  out <- data.frame(do.call(rbind, my.list))
  colnames(out) <- c("mean", "sd")
  return(out)
}
```

Do not stack objects II

```
n <- 1000; p <- 10; x <- matrix(rnorm(n*p), n, p)
res <- microbenchmark(simu.stack(x), simu.df(x), simu.list(x), times=20)
```



Parallelizing is very easy I

Do some parallel computation as soon as you do simulations (this should happen sometimes)

```
library(parallel) ## embedded with R since version 2.9 or something
cores <- detectCores() ## How many cores do I have?
print(cores)

## [1] 4
```

My simulations study estimate the test error from ridge regression

```
one.simu <- function(i) {
  ## draw data
  n <- 1000; p <- 500
  x <- matrix(rnorm(n*p),n,p) ; y <- rnorm(n)
  ## return ridge's coefficients
  train <- 1:floor(n/2)
  test  <- setdiff(1:n,train)
  ridge <- lm.ridge(y~x+0,lambda=1,subset=train)
  err <- (y[test] - x[test, ] %*% ridge$coef )^2
  return(list(err = mean(err), sd = sd(err)))
}
```

Parallelizing is very easy II

```
out <- mclapply(1:8, one.simu, mc.cores=cores)
head(do.call(rbind, out))
```

```
##      err      sd
## [1,] 14.51535 19.72119
## [2,]  9.789891 13.42868
## [3,] 10.1473  15.31773
## [4,] 10.92042 15.25741
## [5,] 16.4902  24.44994
## [6,] 13.47291 19.86978
```

Plan

Programmer en R

Accélérer son code

Analyse du code

Vectorisation

Privilégier les objets simples

Intégration de code C/C++

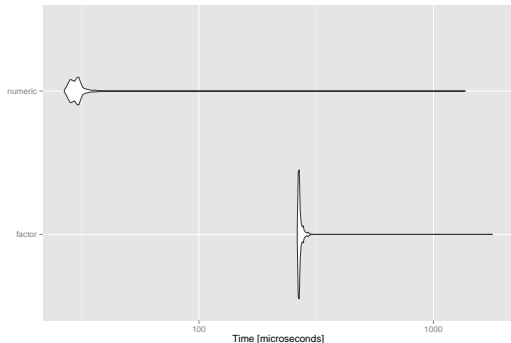
Factor conversion are slow (nlevels)

Do not use factor if you need to perform just one operation on it.

```
nlevels.factor <- function(n,K) {  
  x <- sample(1:K, n, rep=TRUE)  
  return(nlevels(factor(x)))  
}
```

```
nlevels.numeric <- function(n,K) {  
  x <- sample(1:K, n, rep=TRUE)  
  return(length(unique(x)))  
}
```

```
res <- microbenchmark(factor = nlevels.factor(1000,10),  
  numeric = nlevels.numeric(1000,10), times=1000)
```



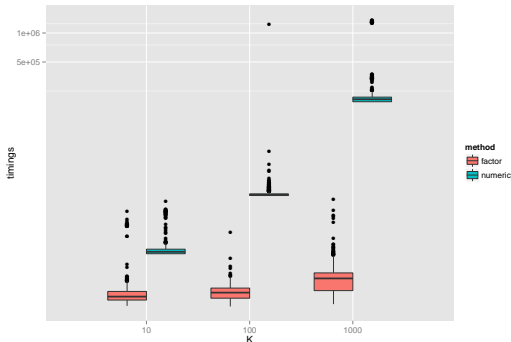
Operations on factors are fast (nlevels)

Use factor if you need repeated operations on the same vector.

```
nk <- 20
seq.K <- c(10,100,1000)
res <- do.call(rbind, lapply(seq.K, function(K) {
  x1 <- rep(1:K,nk)
  x2 <- factor(x1)
  out <- microbenchmark(factor = nlevels(x2),
                        numeric = length(unique(x1)), times=1000)
  return(data.frame(method = out$expr, timings = out$time, K = factor(K)))
}))
```

```
head(res)
```

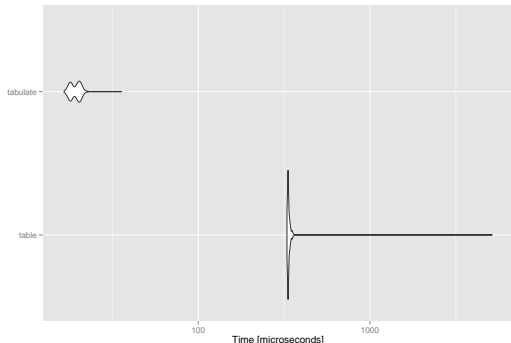
```
##      method timings  K
## 1 factor    14013 10
## 2 factor     2518 10
## 3 factor     2138 10
## 4 factor     1577 10
## 5 numeric    17790 10
## 6 factor     2447 10
```



Avoid table whenever you can

table is a complex function that should not be use for simple operations like counting the occurrences of integers in a vector.

```
n <- 1000
K <- 10
res <- microbenchmark(table      = table      (sample(1:K, n, rep=TRUE)),
                       tabulate  = tabulate  (sample(1:K, n, rep=TRUE)),
                       times=1000)
```



Plan

Programmer en R

Accélérer son code

- Analyse du code

- Vectorisation

- Privilégier les objets simples

- Intégration de code C/C++

Interfacing C++ with R is really easy I

For a vector $\mathbf{x} = (x_1, \dots, x_n)$, consider the simple task of computing

$$y_k = \sum_{i=1}^k \log(x_i), \quad k = 1, \dots, n.$$

```
R.vect <- function(x) {return(cumsum(log(x)))}
```

One can easily integrate some C++ version of this code in R.

```
library(RcppArmadillo)
library(inline)

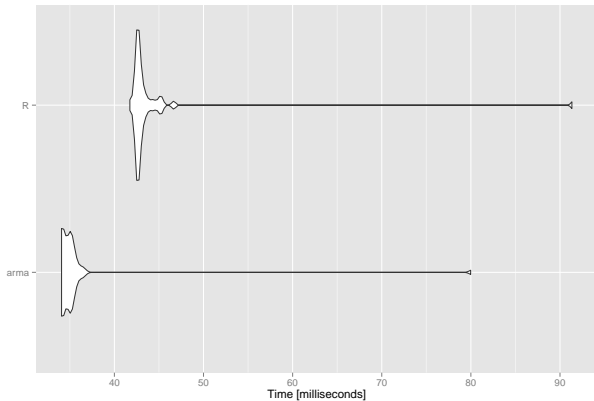
code.arma <- '
using namespace Rcpp;
using namespace arma;

vec x = as<vec>(X) ;
return(wrap(cumsum(log(x)))) ;
'

C.arma <- cxxfunction(signature(X="numeric"), code.arma, plugin="RcppArmadillo")
```

Interfacing C++ with R is really easy III

```
x <- runif(1e6)
res <- microbenchmark(arma = C.arma(x), R = R.vect(x), times=40)
```



Pour aller plus loin



The R inferno, Patrick Burns

<http://www.burns-stat.com/documents/books/the-r-inferno/>



FasteR! HigheR! StrongeR!, Noam Ross

<http://www.noamross.net/blog/2013/4/25/faster-talk.html>



Seamless R and C++ integration with Rcpp, Dirk EddelBuettel

<http://dirk.eddelbuettel.com>



Hadley Wickham, ggplot2, an implementation of the grammar of graphics

<http://had.co.nz/>, <http://ggplot2.org/>, <http://yihui.name/knitr/>