Université de Montpellier

Master 1 Biostat / MIND

Développement logiciel avec R

Mathieu Ribatet

Table des matières

Remarques préliminaires

Ceci n'est pas un cours de programmation à proprement parlé mais vise à vous apprendre la programmation scientifique avec le langage `R`. Cela dit voici un avertissement

aucun cours ne pourra réellement vous apprendre un langage informatique, ce n'est que vous et vous seul qui le ferez ! ! !

Si ce cours n'est pas dur techniquement, il va vous demander un travail **plus que régulier**. Vous êtes prévenu ! ! !

Chapitre 1

Importation/Exportation des données et représentations graphiques

1.1 Importation/Exportation de données

Tôt ou tard, le statisticien devra analyser des données et ces dernières ne seront vraisemblablement pas disponibles directement dans le logiciel statistique qu'il utilise pour ses analyses. Il s'agit donc d'être capable **d'importer** ces données.

Les données de « base » seront souvent sous un format texte `documenté`, c'est à dire que le format d'écriture est renseigné, au sein d'un tableur, Excel par exemple, ou encore dans une grande base de donnée. Plus rarement, cela peut arriver si un de vos collègues à déjà importé les données, les données pourront être directement au format spécifique du logiciel `R`.

Pour faire simple, voici les fonctions qu'il faut à tout prix connaître :

Importation `read.table` (`scan`) et `load` ;

Exportation `save` et `write.table`.

Typiquement vous utiliserez la fonction `read.table` pour importer des données dans `R` des données enregistrées dans un format texte, i.e., .txt, .csv, ou autres formats plus exotiques. La fonction `save` vous servira à enregistrer des données de `R` au format de base de `R` afin de les réutiliser plus tard ; la fonction `write.table` vous permettra quand à elle de sauvegarder des données de `R` vers un fichier texte.

Remarque. La fonction `scan` sera très rarement utilisée—bien que très puissante.

Je vous mets plus bas les arguments de ces fonctions et nous allons commenter les plus importants—d'où les espaces vides pour vos notes. En parallèle je vous conseille de regarder l'aide de chacune de ces fonctions.

```
> args(read.table)

function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
    col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
```

```
    encoding = "unknown", text, skipNul = FALSE)
NULL
```

```
> args(save)

function (..., list = character(), file = stop("'file' must be specified"),
    ascii = FALSE, version = NULL, envir = parent.frame(), compress = isTRUE(!ascii),
    compression_level, eval.promises = TRUE, precheck = TRUE)
NULL
```

```
> args(write.table)

function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
    eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"), fileEncoding = "")
NULL
```

Exemple 1.1 (Hubble)**.** En 1929 Edwin Hubble s'est intéressé à la relation liant la distance de nébuleuses à la planète Terre à leurs vitesses radiales—aussi appelée vitesses de fuite. Hubble espérait ainsi comprendre un peu mieux la naissance de l'univers et la manière dont celui ci évolue. Il en déduisit la fameuse loi de Hubble confirmant l'expansion de l'univers, i.e.,

$$v = H_0 d,$$

où v est la vitesse radiale, H_0 la constante de Hubble et d la distance.

Les données correspondent à 24 nébuleuses différentes. La distance est en *megaparsec* (environ 3.09×10^{19} kilomètres) et la vitesse radiale en km/s.

```
> ## Le fichier de donnees se trouve dans le repertoire Data
> data <- read.table("Data/hubble.txt")##ne marche meme pas

Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
la ligne 2 n'avait pas 5 'el'ements

> data <- read.table("Data/hubble.txt", sep = ";")
> data[1:3,]##toujours pas bon

       V1                  V2        V3
1  Galaxy  recession_velocity  distance
2 NGC0300                 133         2
3 NGC0925                 664      9.16

> data <- read.table("Data/hubble.txt", sep = ";", header = TRUE)
> data[1:3,]##Ah enfin !!!

    Galaxy recession_velocity distance
1  NGC0300                133     2.00
2  NGC0925                664     9.16
3 NGC1326A               1794    16.14
```

Question 1.1. *Dans l'exemple Hubble ci dessus, quel est le type de la variable* `data` *?*

Réponse.

□

Une fois les données importées nous pouvons commencer à se les approprier.

```
> summary(data)##fournit un r<c3><a9>sum<c3><a9> statistique succinct

      Galaxy   recession_velocity    distance
 IC4182  : 1   Min.    :  80.0     Min.   : 2.00
 NGC0300 : 1   1st Qu.: 616.5     1st Qu.: 8.53
 NGC0925 : 1   Median : 827.0     Median :13.08
 NGC1326A: 1   Mean   : 924.4     Mean   :12.05
 NGC1365 : 1   3rd Qu.:1423.2     3rd Qu.:15.87
 NGC1425 : 1   Max.   :1794.0     Max.   :21.98
 (Other) :18
```

Question 1.2. *Et si mes données sont fournies dans un tableur (type Excel) ? C'est quand même très utilisé en entreprise... Comment je fais ? ? ?*

Réponse.

□

Faire un exemple d'utilisation des fonctions `write.table`, `save` et `load`. Je vous laisse un peu de place pour mettre vos commentaires.

1.2 Les graphiques

1.2.1 La base

« Une figure valant mille mots, ... »

On a tous entendu au moins une fois cette phrase et elle souligne donc l'importance d'une figure. C'est encore plus le cas lors de la rédaction de rapports scientifiques—votre futur job quoi !!! Pour autant ce n'est pas facile de faire une **bonne** figure (et si je vous provoque un peu, la plupart des figures faites dans vos premiers rapports sont ignobles ;-). La raison est très simple :

les figures produites sur un écran n'ont pas vocation à se retrouver telles quelles dans un rapport.

Voici quelques points essentiels :

- quel type de graphiques (histogramme, scatter plot, coplot, bar plot, boxplot, QQ-plot, ...)
- les proportions de la figure (**aspect ratio**). Imaginez vous payer 10 euros pour une place de cinéma et l'écran est un carré de 2m sur 2m !!!
- Les marges et la taille de la police.

Remarque. Notez que je ne parle pas du titre car le titre de la figure se fait dans mon logiciel de traitement de texte (LaTeX, Word, ...) et **pas** lors de la création de la figure.

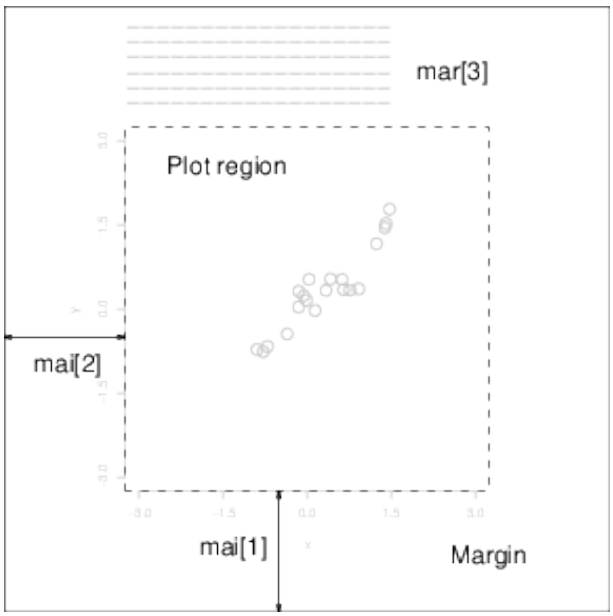


Figure 1.1 – *La structure des fenêtre graphiques de R.*

Table 1.1 – *Liste des fonctions graphiques (de base) que j'utilise souvent avec certaines de leurs options.*

Nom	Objectif	Options
`plot`	trace $\{(x, y)\}$	`type`, `col`, `xlim`, `xlab`, `axes`, `bty`, `pch`, `add`, `lty`
`points`	ajoute les points $\{(x, y)\}$	`pch`, `col`
`lines`	trace des segments	`col`, `lty`, `lwd`
`abline`	trace la droite $y = bx + a$	`col`, `lty`, `lwd`
`barplot`	digramme en bâton	`beside`
`hist`	trace un histogramme	`col`, `freq`
`boxplot`	trace un boxplot	—
`legend`	ajoute une légende	`bty`, `lty`, `inset`
`text`	ajoute du texte	`cex`, `pos`

La Figure 1.1 illustre la structure interne des fenêtres graphiques dans `R`. Nous pouvons voir qu'une fenêtre graphique possède principalement deux régions :

— Une région de dessin (plot region) ;
— une région de marge (margin).

Remarque. Bien entendu il est possible de compliquer les choses en divisant la fenêtre graphique en plusieurs sous graphique—nous verrons ça plus tard, pour le moment faisons simple.

Pour faire simple (mais peut être trop simpliste), on peut classer les graphiques statistiques en deux grandes familles :

— Visualisation des données brutes. Il s'agit ici de représenter une variable y en fonction d'une autre x (fonction `plot`) afin d'identifier une quelconque structure de dépendance. Si l'on a plus de deux variables alors on peut faire de tels graphiques deux à deux (fonction `pairs`) ou encore un `coplot`.
— Représentation synthétique des données. Typiquement trop d'informations tue l'information et l'on cherche donc à « résumer « nos observations via un dot plot (fonction `dotchart`), un histogramme (fonction `hist`), un boxplot (fonction `boxplot`).

Le Tableau 1.1 répertorie les fonctions graphiques de base de `R` les plus fréquemment utilisées avec les options que j'utilise fréquemment. Je vous conseille **fortement** d'aller

voir l'aide de chacune de ces fonctions afin de les maîtriser par coeur.

1.2.2 Graphiques avancés

La section précédente présentait que brièvement les graphiques mais `R` permet de faire des graphiques bien plus complexes. Une des fonctions clés est la fonction `par` qui admet un nombre impressionnant d'arguments possibles. Je liste seulement ceux que j'utilise le plus souvent :

`mar` un vecteur de taille 4 spécifiant l'espacement pour chacune des marges de la figure—ordre : bas, gauche, haut, droit. Typiquement j'utilise `mar = c(4, 5, 0.5, 0.5)`.

`mfrow` un vecteur de taille deux correspondant à la taille d'une matrice—la fenêtre graphique est alors divisée de manière respective et les graphiques sont remplies lignes par lignes.

`ps` un entier déterminant la taille de la police—point size.

`bty` une chaîne de caractère parmi "o", "l", "7", "u", "]" ou "n" spécifiant comment le type « d'encadrement autour du graphique ». Typiquement j'utilise quand je ne veux pas d'encadrement `bty = "n"` sinon je laisse la valeur par défaut.

Exemple 1.2. Faites un histogramme pour deux échantillons de tailles 100, l'un provenant d'une $N(0, 1)$, l'autre d'une Student(20). Vous devrez le faire sur une même figure mais dans deux sous figures différentes.

Lorsque l'on souhaite « diviser » la fenêtre graphique en plusieurs sous graphiques, la fonction `par` nous aide mais divise de manière régulière l'espace. Il peut arriver que l'on souhaite que cela soit différent. La fonction `layout` permet cela. Elle prend principalement trois arguments

`mat` qui est une matrice d'entier définissant comment la fenêtre graphique doit être divisée ;

`widths, heights` qui sont des vecteur spécifiant la largeur et la hauteur de chacune des sous figures.

Remarque. Il peut parfois être utile de mettre des 0 dans l'argument `mat` ainsi cette région ne sera jamais utilisée.

Remarque. La fonction `show.layout` permet de visualiser le partitionnement.

Exemple 1.3. Reprendre l'exemple précédent mais en divisant la fenêtre graphique telle que l'histogramme de l'échantillon Gaussien prenne les 2/3 de la largeur—les hauteurs étant les mêmes.

Le logiciel `R` peut également écrire les maths ce qui peut être très utilise pour avoir les mêmes notations dans votre rapport que sur vos graphiques. Pour voir toutes les capacités, il suffit de lancer dans `R`

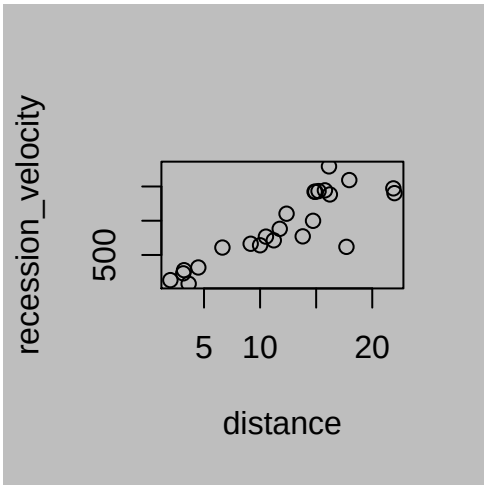


Figure 1.2 – *Scatterplot des données Hubble, c'est moche on est tous d'accord. Qu'est ce qui ne va pas ? J'ai mis en gris le fond de la figure pour que vous puissiez mieux voir.*

```
> demo(plotmath)
```

1.3 Application

Reprenons nos données de Hubble et essayons de faire un graphique—d'abord moche puis que l'on va arranger. La Figure 1.2 est un scatterplot obtenu par `R` de ces données. Faites quelques commentaires.

Remarque (Pour votre culture générale). La constante de Hubble permet d'avoir une estimation de l'âge de l'univers puisque, selon cette loi, chaque étoile a parcouru une distance d avec une vitesse v depuis le Big Bang. Ainsi l'âge de l'univers est estimé par

$$\text{Age} = \frac{d}{v} = \frac{1}{H_0} \approx 0.01306 \times 3.09 \times 10^{19}/(60 \times 60 \times 24 \times 365.25) \approx 13 \text{ milliards d'années}$$

1.4 Exercices

Exercice 1.1 (Alcool et tabac)**.**

a) Importez les données—dans `R` et votre cerveau aussi!!!!

b) Faites un scatterplot de ces données.

c) Que pensez vous du comportement de l'Irelande du nord ?

Bonus : Utilisez la fonction `lm` qui ajuste un modèle linéaire—en utilisant avec ou sans l'Irelande du nord. Que constatez vous ?

Exercice 1.2 (QQ-plot)**.**

a) La fonction `qqplot` permet de faire un QQ-plot. En utilisant la fonction `rnorm` pour simuler un échantillon gaussien de taille n, faire un QQ-plot selon une loi Normale.

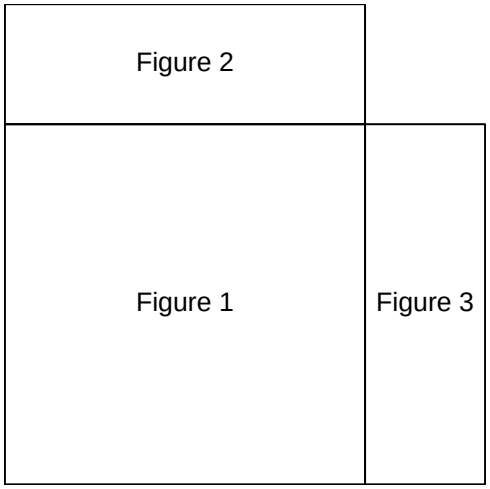
b) Comparez votre graphique à celui produit par la fonction `qqnorm`.

c) La fonction `rstudent` simule des échantillons selon la loi de Student. Faite un QQ-plot pour comparer l'échantillon Gaussien à celui issue d'une Student à $\nu = 1, 10, 20, 40$ degrés de liberté.

Astuce : Utilisez l'aide de `R` pour comprendre le fonctionnement des fonctions citées plus haut.

Exercice 1.3 (Légende)**.**
Sur le même graphique, tracez les fonctions $f(x) = \exp(-x^2/2)$ et $g(x) = \exp(-x^2/6)$. Vous utiliserez des couleurs et une légende afin que l'on sache qui est qui...

Exercice 1.4. Quelle commande utiliseriez vous pour obtenir la division de la fenêtre graphique suivante ?



Exercice 1.5 (Appel pour le service militaire aux USA)**.** Le congrès américain en 1970 décida, pour le service militaire, d'appeler les candidats à l'aide d'un processus aléatoire. Les 366 jours de naissance possible furent mises dans une machine de type « Loto ». Puis les boules furent tirées successivement appelant ainsi successivement les candidats. Les données sont dans le fichier `draftLottery.txt`.

a) Importez les données.

b) Représentez graphiquement les données et décrivez brièvement ce que vous visualisez.

Exercice 1.6 (Vendredi 13)**.** Des chercheurs se sont intéressés à l'impact psychologique des Vendredi 13 sur la consommation. Ainsi ils ont compté (entre autre) le nombre de clients les vendredi 13 et ainsi que ce même nombre de client les vendredi de la semaine précédente—donc un vendredi 6. Les données sont dans le fichier `vendredi13.txt`.

a) Importez les données.

b) Faites une représentation graphique appropriée pour essayer de savoir si le vendredi 13 a un impact économique sur le shopping...

Exercice 1.7 (R graph gallery)**.** Allez jeter un oeil sur la `R graph gallery` (`http://rgraphgallery.blogspot.fr`) et regardez le code de figures qui vous plaisent.

Exercice 1.8 (C'est vous le prof !)**.** Essayez de réfléchir à des problèmes que l'on pourrait résoudre avec un ordinateur—et notamment avec `R`. J'essaierai de les résoudre—ne soyez pas trop durs ;-)

Chapitre 2

Programmation scientifique

Dans ce chapitre nous allons apprendre la syntaxe de base de `R` pour faire de la programmation scientifique. Mais avant de rentrer dans le vif du sujet, nous allons commencer par donner les bonnes habitudes—habitudes qui sont totalement indépendantes de `R`.

2.1 Structurer son travail

Savoir structurer son travail est certainement la chose la plus importante dans le monde professionnel. En fait c'est grâce à cela que l'on vous trouvera **efficace**. Ne négligez donc surtout pas les conseils que je vais vous donner, croyez moi ils vous seront très utiles et bien plus rapidement que vous ne le pensez.

2.1.1 Séparation Modèle / Analyse

En informatique, il y a une structure de plus en plus recommandé pour l'élaboration de logiciels type (app iOs ou autr) c'est le modèle **MVC** : Model / View / Controller. Nous matheux c'est un peu différent mais pas tant que ça au final. Nous utiliserons principalement le modèle Modèle / Analyse :

Modèle Regroupe l'implémentation de méthodes statistiques **générique** ;

Analyse Code qui **exécute** a proprement parlé les méthodes statistiques précédentes sur un **jeu de données fixé**.

Exemple 2.1. Je suis analyste financier et mon chef souhaite que je fasse de l'allocation optimale de portefeuille selon la théorie de Markowitz.

Mon modèle sera donc la méthodologie développée par Markowitz et que j'aurais codé sous `R`.

Mon analyse sera un petit bout de code `R` qui lit les données fournie par mon chef et exécute le modèle de Markowitz.

Clairement mon chef me trouve efficace car le premier jour j'ai mis un peu de temps mais le deuxième jour c'était instantanné—enfin je ne lui aie pas donné les résultats tout de suite car j'ai pris un café...

Pour résumer, je vous encourage vivement à utiliser l'arborescence suivante pour tous vos projets :

```
- CodeR/
  +- Models/
  |  +- fichier1.R
  |  +- fichier2.R
```

```
|  +- ...
+- Analysis/
|  +- script1.R
|  +- script2.R
|  +- ...
```

Remarque. Les noms que j'ai donné aux fichiers, dossiers sont bien entendu purement indicatifs, seule la structure est importante.

2.1.2 Les bonnes habitudes

Vous avez de la chance ! Oui de la chance car vous débutez et vous pouvez donc en apprenant prendre directement les bons réflexes—car vous savez que changer ses (mauvaises) habitudes est difficile.

Les bonnes habitudes relèvent sur deux points principaux :

— rendre le code **lisible** ;

— rendre le code **lisible au sens scientifique** ;

Le premier point n'est rien d'autre que la mise en forme de votre code et si vous utilisez un éditeur de code adapté à `R`, cela devrait se faire automatiquement. Les points essentiels ici sont

— l'indentation—devrait être automatique avec un bon éditeur ;

— aérer son code, notamment en mettant des espaces entre les opérateurs mathématiques et en « cassant » les lignes de code trop longues ;

— être consistent tout au long de votre code—ainsi le lecteur prendra petit à petit vos habitudes de codage. Par exemple, comment s'ouvrent et se ferment les blocs de code `if` / `else`.

Le dernier point est plus difficile mais pour autant ultra important. Plus difficile car il laisse une infinité de choix possibles. Avez vous déjà remarqués que vos profs utilisent souvent les mêmes notations pour les variables ? ? ? Genre

$$\text{Soit } Z \sim N(0,1), \quad \text{Soit } x \in \mathbb{R}, \quad \text{Soit } n \in \mathbb{N} \dots$$

Il est par exemple tout à fait juste d'écrire

$$\text{Soit } x \in \mathbb{N}$$

mais ça pique un peu les yeux non ? Bah c'est pareil lorsque l'on fait du code. J'ai des variables et je **dois à tout prix trouver des noms pertinents pour ces variables**.

Remarque. Comme la plupart des langages scientifiques sont en anglais, je vous conseille fortement de donner des noms anglais à vos variables.

Exemple 2.2 (Maxima annuels)**.** Il s'agit ici de calculer les maxima annuels à partir de données journalières.

```
> ## Affichons les premieres lignes de nos donnees
> ## pour mieux comprendre la suite
> data[1:5,]

  year day.of.year       obs
1 2011           1 11.946647
2 2011           2 11.225273
3 2011           3 10.068509
4 2011           4  5.382407
5 2011           5 13.076579
```

```
> years <- min(data$year):max(data$year)
> ans <- NULL
>
> for (year in years){
+     idx <- which(data$year == year)
+     ans <- c(ans, max(data$obs[idx]))
+ }
```

2.2 Les bases du langages R

On apprend réellement le langage par soi même donc je vais faire très vite mais **vous devez faire votre travail personnel afin d'apprendre ce langage.** Je liste plus bas les points les plus fréquemment rencontrés.

```
> ## Les types de bases : scalaires, vecteurs, matrices, booleens
> x <- 2 #affecte 2 a la variable x
> x

[1] 2

> y <- c(1, -5, 3) ## cree le vecteur y = (1, -5, 3)
> y

[1]  1 -5  3

> z <- c(y, x) ## cree le vecteur z = (1, -5, 3, 2)
> z

[1]  1 -5  3  2

> A <- matrix(c(1, 0.5, 0.5, 1), 2, 2) ## cree une matrice (2, 2)
> A

     [,1] [,2]
[1,]  1.0  0.5
[2,]  0.5  1.0

> flag <- c(TRUE, FALSE, TRUE)
> flag

[1]  TRUE FALSE  TRUE

> ## Selection / Deselection
> y[1]

[1] 1

> y[c(1, 3)]

[1] 1 3
```

```
> y[-c(1,3)]

[1] -5

> A[1,]

[1] 1.0 0.5

> A[,1]

[1] 1.0 0.5

> A[-1,]

[1] 0.5 1.0

> A[,-1,drop=FALSE]

     [,1]
[1,]  0.5
[2,]  1.0

> ## Des types plus avances et souvent tr<c3><a8>s utiles
> data <- data.frame(type = x, taille = y, groupe = c("A", "B", "A"))
> data

  type taille groupe
1    2      1      A
2    2     -5      B
3    2      3      A

> data$type##ou encore

[1] 2 2 2

> data[,1]

[1] 2 2 2

> data <- list(type = x, taille = y, groupe = "A")
> data$taille## ou encore

[1]  1 -5  3

> data[[2]]

[1]  1 -5  3

> ## Les valeurs speciales de R
> NULL## semblable a l'ensemble vide

NULL

> c(NA, NaN, Inf, -Inf)##not available, not a number, +/- infini

[1]   NA  NaN  Inf -Inf

> 0 / 0

[1] NaN

> 1 / 0

[1] Inf
```

```
> ## Les operations arithmetiques usuelles
> 3 + 2 * 5^2 ## connait les priorites bien sur

[1] 53

> c(exp(log(3)), cos(pi), choose(5, 2))

[1]  3 -1 10

> A %*% solve(A) ## mult. matricielle et inversion

     [,1] [,2]
[1,]    1    0
[2,]    0    1

> 1 + z ## capacites vectorielles

[1]  2 -4  4  3

> 1 + A

     [,1] [,2]
[1,]  2.0  1.5
[2,]  1.5  2.0

> 1 / A ## vrai en R, tellement faux en maths !!!

     [,1] [,2]
[1,]    1    2
[2,]    2    1
```

```
> ## Boucles et tests
> flag <- TRUE
> if (flag == TRUE) {
+     flag <- FALSE
+     print("flag etait vrai mais est faux maintenant !")
+ } else {
+     flag <- !flag
+     print("flag etait faux maix est vrai maintenant !")
+ }

[1] "flag etait vrai mais est faux maintenant !"

> ans <- 0
> for (i in 1:3){
+     ans <- ans + i
+     print(ans)
+ }

[1] 1
[1] 3
[1] 6
```

```
> flag <- TRUE
> while (flag) {
+  x <- rnorm(1)

+  flag <- x > 0
+ }
> x

[1] -2.550139
```

Si vous respectez ce que je vous ai dit plus tôt sur la séparation Modèle / Analyse, dans votre dossier `Models`, vous devriez donc avoir des fonctions `R`. Voici une fonction `R` qui calcule la valeur absolue—cette fonction existe déjà bien entendu et s'appelle `abs`.

```
> myabs <- function(x) {
+     if (x <= 0)
+         x <- -x

+     return(x)
+ }
```

Le code suivant crée juste une fonction mais ne fait rien d'autre. Il faut l'appeler pour qu'il se passe quelque chose :

```
> myabs(2)

[1] 2

> myabs(-2)

[1] 2
```

Question 2.1. *Commentez l'exécution suivante :*

```
> x <- -3
> myabs(x)

[1] 3

> x

[1] -3
```

Réponse.

□

Remarque. Boucle `for` ou `while` ? La boucle `for` s'utilise lorsque l'on sait à l'avance que l'on va faire n (bloc d') instructions alors que la boucle `while` sera utilisée lorsque l'on ne sait pas à l'avance combien de fois il va falloir répéter l'instruction.

Remarque. Fonction et argument de sortie Par construction, les fonctions `R` ne peuvent renvoyer qu'un seul argument. Si vous avez besoin de retourner plusieurs objets alors typiquement vous utiliserez une `list`—j'utilise d'ailleurs ce type d'objet uniquement dans ce cas !

R Reference Card

by Tom Short, EPRI PEAC, tshort@epri-peac.com 2004-11-07

Getting help

Most R functions have online documentation.

help(topic) documentation on topic
~~**?topic** id.~~
help.search("topic") search the help system
apropos("topic") the names of all objects in the search list matching the regular expression "topic"
help.start() start the HTML version of help
str(a) display the internal *str*ucture of an R object
summary(a) gives a "summary" of a, usually a statistical summary but it is *generic* meaning it has different operations for different classes of a
ls() show objects in the search path; specify pat="pat" to search on a pattern
ls.str() str() for each variable in the search path
dir() show files in the current directory
methods(a) shows S3 methods of a
methods(class=class(a)) lists all the methods to handle objects of class a

Input and output

load() load the datasets written with save
data(x) loads specified data sets
library(x) load add-on packages
read.table(file) reads a file in table format and creates a data frame from it; the default separator sep="" is any whitespace; use header=TRUE to read the first line as a header of column names; use as.is=TRUE to prevent character vectors from being converted to factors; use comment.char="" to prevent "#" from being interpreted as a comment; use skip=n to skip n lines before reading data; see the help for options on row naming, NA treatment, and others
read.csv("filename",header=TRUE) id. but with defaults set for reading comma-delimited files
read.delim("filename",header=TRUE) id. but with defaults set for reading tab-delimited files
read.fwf(file,widths,header=FALSE,sep="",as.is=FALSE) read a table of *f*ixed *width f*ormatted data into a 'data.frame'; widths is an integer vector, giving the widths of the fixed-width fields
save(file,...) saves the specified objects (...) in the XDR platform-independent binary format
save.image(file) saves all objects
cat(..., file="", sep=" ") prints the arguments after coercing to character; sep is the character separator between arguments
print(a, ...) prints its arguments; generic, meaning it can have different methods for different objects
format(x,...) format an R object for pretty printing
write.table(x,file="",row.names=TRUE,col.names=TRUE, sep=" ") prints x after converting to a data frame; if quote is TRUE, character or factor columns are surrounded by quotes ("); sep is the field separator; eol is the end-of-line separator; na is the string for missing values; use col.names=NA to add a blank column header to get the column headers aligned correctly for spreadsheet input
sink(file) output to file, until sink()

Most of the I/O functions have a file argument. This can often be a character string naming a file or a connection. file="" means the standard input or output. Connections can include files, pipes, zipped files, and R variables.

On windows, the file connection can also be used with description = "clipboard". To read a table copied from Excel, use
x <- read.delim("clipboard")
To write a table to the clipboard for Excel, use
write.table(x,"clipboard",sep="\t",col.names=NA)
For database interaction, see packages RODBC, DBI, RMySQL, RPgSQL, and ROracle. See packages XML, hdf5, netCDF for reading other file formats.

Data creation

c(...) generic function to combine arguments with the default forming a vector; with recursive=TRUE descends through lists combining all elements into one vector
from:to generates a sequence; ":" has operator priority; 1:4 + 1 is "2,3,4,5"
seq(from,to) generates a sequence by= specifies increment; length= specifies desired length
seq(along=x) generates 1, 2, ..., length(along); useful for for loops
rep(x,times) replicate x times; use each= to repeat "each" element of x each times; rep(c(1,2,3),2) is 1 2 3 1 2 3; rep(c(1,2,3),each=2) is 1 1 2 2 3 3
data.frame(...) create a data frame of the named or unnamed arguments; data.frame(v=1:4,ch=c("a","B","c","d"),n=10); shorter vectors are recycled to the length of the longest
list(...) create a list of the named or unnamed arguments; list(a=c(1,2),b="hi",c=3i);
array(x,dim=) array with data x; specify dimensions like dim=c(3,4,2); elements of x recycle if x is not long enough
matrix(x,nrow=,ncol=) matrix; elements of x recycle
factor(x,levels=) encodes a vector x as a factor
gl(n,k,length=n*k,labels=1:n) generate levels (factors) by specifying the pattern of their levels; k is the number of levels, and n is the number of replications
expand.grid() a data frame from all combinations of the supplied vectors or factors
rbind(...) combine arguments by rows for matrices, data frames, and others
cbind(...) id. by columns

Slicing and extracting data

Indexing vectors

x[n]	n^{th} element
x[-n]	all *but* the n^{th} element
x[1:n]	first n elements
x[-(1:n)]	elements from n+1 to the end
x[c(1,4,2)]	specific elements
x["name"]	element named "name"
x[x > 3]	all elements greater than 3
x[x > 3 & x < 5]	all elements between 3 and 5
x[x %in% c("a","and","the")]	elements in the given set

Indexing lists

x[n]	list with elements n
x[[n]]	n^{th} element of the list
x[["name"]]	element of the list named "name"
x$name	id.

Indexing matrices

x[i,j]	element at row i, column j
x[i,]	row i
x[,j]	column j
x[,c(1,3)]	columns 1 and 3
x["name",]	row named "name"

Indexing data frames (matrix indexing plus the following)

x[["name"]]	column named "name"
x$name	id.

Variable conversion

as.array(x), as.data.frame(x), as.numeric(x), as.logical(x), as.complex(x), as.character(x), ... convert type; for a complete list, use methods(as)

Variable information

is.na(x), is.null(x), is.array(x), is.data.frame(x), is.numeric(x), is.complex(x), is.character(x), ... test for type; for a complete list, use methods(is)
length(x) number of elements in x
dim(x) Retrieve or set the dimension of an object; dim(x) <- c(3,2)
dimnames(x) Retrieve or set the dimension names of an object
nrow(x) number of rows; NROW(x) is the same but treats a vector as a one-row matrix
ncol(x) and **NCOL(x)** id. for columns
class(x) get or set the class of x; class(x) <- "myclass"
unclass(x) remove the class attribute of x
attr(x,which) get or set the attribute which of x
attributes(obj) get or set the list of attributes of obj

Data selection and manipulation

which.max(x) returns the index of the greatest element of **x**
which.min(x) returns the index of the smallest element of **x**
rev(x) reverses the elements of x
sort(x) sorts the elements of x in increasing order; to sort in decreasing order: rev(sort(x))
cut(x,breaks) divides x into intervals (factors); breaks is the number of cut intervals or a vector of cut points
match(x, y) returns a vector of the same length than x with the elements of x which are in y (NA otherwise)
which(x == a) returns a vector of the indices of x if the comparison operation is true (TRUE), in this example the values of i for which x[i] == a (the argument of this function must be a variable of mode logical)
choose(n, k) computes the combinations of k events among n repetitions $= n!/[(n-k)!k!]$
na.omit(x) suppresses the observations with missing data (NA) (suppresses the corresponding line if x is a matrix or a data frame)
na.fail(x) returns an error message if x contains at least one NA

unique(x) if x is a vector or a data frame, returns a similar object but with the duplicate elements suppressed

table(x) returns a table with the numbers of the differents values of x (typically for integers or factors)

subset(x, ...) returns a selection of x with respect to criteria (..., typically comparisons: x$V1 < 10); if x is a data frame, the option select gives the variables to be kept or dropped using a minus sign

sample(x, size) resample randomly and without replacement size elements in the vector x, the option replace = TRUE allows to resample with replacement

prop.table(x,margin=) table entries as fraction of marginal table

Math

sin,cos,tan,asin,acos,atan,atan2,log,log10,exp

max(x) maximum of the elements of x

min(x) minimum of the elements of x

range(x) id. then c(min(x), max(x))

sum(x) sum of the elements of x

diff(x) lagged and iterated differences of vector x

prod(x) product of the elements of x

mean(x) mean of the elements of x

median(x) median of the elements of x

quantile(x,probs=) sample quantiles corresponding to the given probabilities (defaults to 0,.25,.5,.75,1)

weighted.mean(x, w) mean of x with weights w

rank(x) ranks of the elements of x

var(x) or cov(x) variance of the elements of x (calculated on $n-1$); if x is a matrix or a data frame, the variance-covariance matrix is calculated

sd(x) standard deviation of x

cor(x) correlation matrix of x if it is a matrix or a data frame (1 if x is a vector)

var(x, y) or cov(x, y) covariance between x and y, or between the columns of x and those of y if they are matrices or data frames

cor(x, y) linear correlation between x and y, or correlation matrix if they are matrices or data frames

round(x, n) rounds the elements of x to n decimals

log(x, base) computes the logarithm of x with base base

scale(x) if x is a matrix, centers and reduces the data; to center only use the option center=FALSE, to reduce only scale=FALSE (by default center=TRUE, scale=TRUE)

pmin(x,y,...) a vector which *i*th element is the minimum of x[i], y[i],...

pmax(x,y,...) id. for the maximum

cumsum(x) a vector which *i*th element is the sum from x[1] to x[i]

cumprod(x) id. for the product

cummin(x) id. for the minimum

cummax(x) id. for the maximum

union(x,y),intersect(x,y),setdiff(x,y),setequal(x,y), is.element(el,set) "set" functions

Re(x) real part of a complex number

Im(x) imaginary part

Mod(x) modulus; abs(x) is the same

Arg(x) angle in radians of the complex number

Conj(x) complex conjugate

convolve(x,y) compute the several kinds of convolutions of two sequences

fft(x) Fast Fourier Transform of an array

mvfft(x) FFT of each column of a matrix

filter(x,filter) applies linear filtering to a univariate time series or to each series separately of a multivariate time series

Many math functions have a logical parameter na.rm=FALSE to specify missing data (NA) removal.

Matrices

t(x) transpose

diag(x) diagonal

%*% matrix multiplication

solve(a,b) solves a %*% x = b for x

solve(a) matrix inverse of a

rowsum(x) sum of rows for a matrix-like object; **rowSums(x)** is a faster version

colsum(x),colSums(x) id. for columns

rowMeans(x) fast version of row means

colMeans(x) id. for columns

Advanced data processing

apply(X,INDEX,FUN=) a vector or array or list of values obtained by applying a function FUN to margins (INDEX) of X

lapply(X,FUN) apply FUN to each element of the list X

tapply(X,INDEX,FUN=) apply FUN to each cell of a ragged array given by X with indexes INDEX

by(data,INDEX,FUN) apply FUN to data frame data subsetted by INDEX

merge(a,b) merge two data frames by common columns or row names

xtabs(a b,data=x) a contingency table from cross-classifying factors

aggregate(x,by,FUN) splits the data frame x into subsets, computes summary statistics for each, and returns the result in a convenient form; by is a list of grouping elements, each as long as the variables in x

stack(x, ...) transform data available as separate columns in a data frame or list into a single column

unstack(x, ...) inverse of stack()

reshape(x, ...) reshapes a data frame between 'wide' format with repeated measurements in separate columns of the same record and 'long' format with the repeated measurements in separate records; use (direction="wide") or (direction="long")

Strings

paste(...) concatenate vectors after converting to character; sep= is the string to separate terms (a single space is the default); collapse= is an optional string to separate "collapsed" results

substr(x,start,stop) substrings in a character vector; can also assign, as substr(x, start, stop) <- value

strsplit(x,split) split x according to the substring split

grep(pattern,x) searches for matches to pattern within x; see ?regex

gsub(pattern,replacement,x) replacement of matches determined by regular expression matching sub() is the same but only replaces the first occurrence.

tolower(x) convert to lowercase

toupper(x) convert to uppercase

match(x,table) a vector of the positions of first matches for the elements of x among table

x %in% table id. but returns a logical vector

pmatch(x,table) partial matches for the elements of x among table

nchar(x) number of characters

Dates and Times

The class Date has dates without times. POSIXct has dates and times, including time zones. Comparisons (e.g. >), seq(), and difftime() are useful. Date also allows + and −. ?DateTimeClasses gives more information. See also package chron.

as.Date(s) and **as.POSIXct(s)** convert to the respective class; format(dt) converts to a string representation. The default string format is "2001-02-21". These accept a second argument to specify a format for conversion. Some common formats are:

%a, %A Abbreviated and full weekday name.
%b, %B Abbreviated and full month name.
%d Day of the month (01–31).
%H Hours (00–23).
%I Hours (01–12).
%j Day of year (001–366).
%m Month (01–12).
%M Minute (00–59).
%p AM/PM indicator.
%S Second as decimal number (00–61).
%U Week (00–53); the first Sunday as day 1 of week 1.
%w Weekday (0–6, Sunday is 0).
%W Week (00–53); the first Monday as day 1 of week 1.
%y Year without century (00–99). Don't use.
%Y Year with century.
%z (output only.) Offset from Greenwich; -0800 is 8 hours west of.
%Z (output only.) Time zone as a character string (empty if not available).

Where leading zeros are shown they will be used on output but are optional on input. See ?strftime.

Plotting

plot(x) plot of the values of x (on the *y*-axis) ordered on the *x*-axis

plot(x, y) bivariate plot of x (on the *x*-axis) and y (on the *y*-axis)

hist(x) histogram of the frequencies of x

barplot(x) histogram of the values of x; use horiz=FALSE for horizontal bars

dotchart(x) if x is a data frame, plots a Cleveland dot plot (stacked plots line-by-line and column-by-column)

pie(x) circular pie-chart

boxplot(x) "box-and-whiskers" plot

sunflowerplot(x, y) id. than plot() but the points with similar coordinates are drawn as flowers which petal number represents the number of points

stripplot(x) plot of the values of x on a line (an alternative to boxplot() for small sample sizes)

coplot(x~y | z) bivariate plot of x and y for each value or interval of values of z

interaction.plot (f1, f2, y) if f1 and f2 are factors, plots the means of y (on the *y*-axis) with respect to the values of f1 (on the *x*-axis) and of f2 (different curves); the option fun allows to choose the summary statistic of y (by default fun=mean)

matplot(x,y) bivariate plot of the first column of x *vs.* the first one of y, the second one of x *vs.* the second one of y, etc.

fourfoldplot(x) visualizes, with quarters of circles, the association between two dichotomous variables for different populations (x must be an array with dim=c(2, 2, k), or a matrix with dim=c(2, 2) if $k = 1$)

assocplot(x) Cohen–Friendly graph showing the deviations from independence of rows and columns in a two dimensional contingency table

mosaicplot(x) 'mosaic' graph of the residuals from a log-linear regression of a contingency table

pairs(x) if x is a matrix or a data frame, draws all possible bivariate plots between the columns of x

plot.ts(x) if x is an object of class "ts", plot of x with respect to time, x may be multivariate but the series must have the same frequency and dates

ts.plot(x) id. but if x is multivariate the series may have different dates and must have the same frequency

qqnorm(x) quantiles of x with respect to the values expected under a normal law

qqplot(x, y) quantiles of y with respect to the quantiles of x

contour(x, y, z) contour plot (data are interpolated to draw the curves), x and y must be vectors and z must be a matrix so that dim(z)=c(length(x), length(y)) (x and y may be omitted)

filled.contour(x, y, z) id. but the areas between the contours are coloured, and a legend of the colours is drawn as well

image(x, y, z) id. but with colours (actual data are plotted)

persp(x, y, z) id. but in perspective (actual data are plotted)

stars(x) if x is a matrix or a data frame, draws a graph with segments or a star where each row of x is represented by a star and the columns are the lengths of the segments

symbols(x, y, ...) draws, at the coordinates given by x and y, symbols (circles, squares, rectangles, stars, thermometres or "boxplots") which sizes, colours ... are specified by supplementary arguments

termplot(mod.obj) plot of the (partial) effects of a regression model (mod.obj)

The following parameters are common to many plotting functions:

add=FALSE if TRUE superposes the plot on the previous one (if it exists)

axes=TRUE if FALSE does not draw the axes and the box

type="p" specifies the type of plot, "p": points, "l": lines, "b": points connected by lines, "o": id. but the lines are over the points, "h": vertical lines, "s": steps, the data are represented by the top of the vertical lines, "S": id. but the data are represented by the bottom of the vertical lines

xlim=, ylim= specifies the lower and upper limits of the axes, for example with xlim=c(1, 10) or xlim=range(x)

xlab=, ylab= annotates the axes, must be variables of mode character

main= main title, must be a variable of mode character

sub= sub-title (written in a smaller font)

Low-level plotting commands

points(x, y) adds points (the option type= can be used)

lines(x, y) id. but with lines

text(x, y, labels, ...) adds text given by labels at coordinates (x,y); a typical use is: plot(x, y, type="n"); text(x, y, names)

mtext(text, side=3, line=0, ...) adds text given by text in the margin specified by side (see axis() below); line specifies the line from the plotting area

segments(x0, y0, x1, y1) draws lines from points (x0,y0) to points (x1,y1)

arrows(x0, y0, x1, y1, angle= 30, code=2) id. with arrows at points (x0,y0) if code=2, at points (x1,y1) if code=1, or both if code=3; angle controls the angle from the shaft of the arrow to the edge of the arrow head

abline(a,b) draws a line of slope b and intercept a

abline(h=y) draws a horizontal line at ordinate y

abline(v=x) draws a vertical line at abcissa x

abline(lm.obj) draws the regression line given by lm.obj

rect(x1, y1, x2, y2) draws a rectangle which left, right, bottom, and top limits are x1, x2, y1, and y2, respectively

polygon(x, y) draws a polygon linking the points with coordinates given by x and y

legend(x, y, legend) adds the legend at the point (x,y) with the symbols given by legend

title() adds a title and optionally a sub-title

axis(side, vect) adds an axis at the bottom (side=1), on the left (2), at the top (3), or on the right (4); vect (optional) gives the abcissa (or ordinates) where tick-marks are drawn

rug(x) draws the data x on the *x*-axis as small vertical lines

locator(n, type="n", ...) returns the coordinates (*x*,*y*) after the user has clicked n times on the plot with the mouse; also draws symbols (type="p") or lines (type="l") with respect to optional graphic parameters (...); by default nothing is drawn (type="n")

Graphical parameters

These can be set globally with **par(...)**: many can be passed as parameters to plotting commands.

adj controls text justification (0 left-justified, 0.5 centred, 1 right-justified)

bg specifies the colour of the background (ex. : bg="red", bg="blue", ... the list of the 657 available colours is displayed with colors())

bty controls the type of box drawn around the plot, allowed values are: "o", "l", "7", "c", "u" ou "]" (the box looks like the corresponding character); if bty="n" the box is not drawn

cex a value controlling the size of texts and symbols with respect to the default; the following parameters have the same control for numbers on the axes, cex.axis, the axis labels, cex.lab, the title, cex.main, and the sub-title, cex.sub

col controls the color of symbols and lines; use color names: "red", "blue" see colors() or as "#RRGGBB"; see rgb(), hsv(), gray(), and rainbow(); as for cex there are: col.axis, col.lab, col.main, col.sub

font an integer which controls the style of text (1: normal, 2: italics, 3: bold, 4: bold italics); as for cex there are: font.axis, font.lab, font.main, font.sub

las an integer which controls the orientation of the axis labels (0: parallel to the axes, 1: horizontal, 2: perpendicular to the axes, 3: vertical)

lty controls the type of lines, can be an integer or string (1: "solid", 2: "dashed", 3: "dotted", 4: "dotdash", 5: "longdash", 6: "twodash", or a string of up to eight characters (between "0" and "9") which specifies alternatively the length, in points or pixels, of the drawn elements and the blanks, for example lty="44" will have the same effect than lty=2

lwd a numeric which controls the width of lines, default 1

mar a vector of 4 numeric values which control the space between the axes and the border of the graph of the form c(bottom, left, top, right), the default values are c(5.1, 4.1, 4.1, 2.1)

mfcol a vector of the form c(nr,nc) which partitions the graphic window as a matrix of nr lines and nc columns, the plots are then drawn in columns

mfrow id. but the plots are drawn by row

pch controls the type of symbol, either an integer between 1 and 25, or any single character within ""

ps an integer which controls the size in points of texts and symbols

pty a character which specifies the type of the plotting region, "s": square, "m": maximal

tck a value which specifies the length of tick-marks on the axes as a fraction of the smallest of the width or height of the plot; if tck=1 a grid is drawn

tcl a value which specifies the length of tick-marks on the axes as a fraction of the height of a line of text (by default tcl=-0.5)

xaxt if xaxt="n" the *x*-axis is set but not drawn (useful in conjonction with axis(side=1, ...))

yaxt if yaxt="n" the *y*-axis is set but not drawn (useful in conjonction with axis(side=2, ...))

Lattice (Trellis) graphics

xyplot(y~x) bivariate plots (with many functionalities)

barchart(y~x) histogram of the values of y with respect to those of x

dotplot(y~x) Cleveland dot plot (stacked plots line-by-line and column-by-column)

densityplot(~x) density functions plot

histogram(~x) histogram of the frequencies of x

bwplot(y~x) "box-and-whiskers" plot

qqmath(~x) quantiles of x with respect to the values expected under a theoretical distribution

stripplot(y~x) single dimension plot, x must be numeric, y may be a factor

qq(y~x) quantiles to compare two distributions, x must be numeric, y may be numeric, character, or factor but must have two 'levels'

splom(~x) matrix of bivariate plots

parallel(~x) parallel coordinates plot

levelplot(z~x*y|g1*g2) coloured plot of the values of z at the coordinates given by x and y (x, y and z are all of the same length)

wireframe(z~x*y|g1*g2) 3d surface plot

cloud(z~x*y|g1*g2) 3d scatter plot

In the normal Lattice formula, `y ~x|g1*g2` has combinations of optional conditioning variables `g1` and `g2` plotted on separate panels. Lattice functions take many of the same arguments as base graphics plus also `data=` the data frame for the formula variables and `subset=` for subsetting. Use `panel=` to define a custom panel function (see `apropos("panel")` and `?llines`). Lattice functions return an object of class trellis and have to be `print`-ed to produce the graph. Use `print(xyplot(...))` inside functions where automatic printing doesn't work. Use `lattice.theme` and `lset` to change Lattice defaults.

Optimization and model fitting

`optim(par, fn, method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN")` general-purpose optimization: `par` is initial values, `fn` is function to optimize (normally minimize)

`nlm(f,p)` minimize function `f` using a Newton-type algorithm with starting values `p`

`lm(formula)` fit linear models; `formula` is typically of the form `response termA + termB + ...`; use `I(x*y) + I(x^2)` for terms made of nonlinear components

`glm(formula,family=)` fit generalized linear models, specified by giving a symbolic description of the linear predictor and a description of the error distribution; `family` is a description of the error distribution and link function to be used in the model; see `?family`

`nls(formula)` nonlinear least-squares estimates of the nonlinear model parameters

`approx(x,y=)` linearly interpolate given data points; `x` can be an xy plotting structure

`spline(x,y=)` cubic spline interpolation

`loess(formula)` fit a polynomial surface using local fitting

Many of the formula-based modeling functions have several common arguments: `data=` the data frame for the formula variables, `subset=` a subset of variables used in the fit, `na.action=` action for missing values: `"na.fail"`, `"na.omit"`, or a function. The following generics often apply to model fitting functions:

`predict(fit,...)` predictions from `fit` based on input data

`df.residual(fit)` returns the number of residual degrees of freedom

`coef(fit)` returns the estimated coefficients (sometimes with their standard-errors)

`residuals(fit)` returns the residuals

`deviance(fit)` returns the deviance

`fitted(fit)` returns the fitted values

`logLik(fit)` computes the logarithm of the likelihood and the number of parameters

`AIC(fit)` computes the Akaike information criterion or AIC

Statistics

`aov(formula)` analysis of variance model

`anova(fit,...)` analysis of variance (or deviance) tables for one or more fitted model objects

`density(x)` kernel density estimates of `x`

`binom.test()`, **`pairwise.t.test()`**, **`power.t.test()`**, **`prop.test()`**, **`t.test()`**, ... use `help.search("test")`

Distributions

`rnorm(n, mean=0, sd=1)` Gaussian (normal)

`rexp(n, rate=1)` exponential

`rgamma(n, shape, scale=1)` gamma

`rpois(n, lambda)` Poisson

`rweibull(n, shape, scale=1)` Weibull

`rcauchy(n, location=0, scale=1)` Cauchy

`rbeta(n, shape1, shape2)` beta

`rt(n, df)` 'Student' (t)

`rf(n, df1, df2)` Fisher–Snedecor (F) (χ^2)

`rchisq(n, df)` Pearson

`rbinom(n, size, prob)` binomial

`rgeom(n, prob)` geometric

`rhyper(nn, m, n, k)` hypergeometric

`rlogis(n, location=0, scale=1)` logistic

`rlnorm(n, meanlog=0, sdlog=1)` lognormal

`rnbinom(n, size, prob)` negative binomial

`runif(n, min=0, max=1)` uniform

`rwilcox(nn, m, n)`, `rsignrank(nn, n)` Wilcoxon's statistics

All these functions can be used by replacing the letter `r` with `d`, `p` or `q` to get, respectively, the probability density (`dfunc(x, ...)`), the cumulative probability density (`pfunc(x, ...)`), and the value of quantile (`qfunc(p, ...)`, with $0 < p < 1$).

Programming

`function( arglist ) expr` function definition

`return(value)`

`if(cond) expr`

`if(cond) cons.expr else alt.expr`

`for(var in seq) expr`

`while(cond) expr`

`repeat expr`

`break`

`next`

Use braces {} around statements

`ifelse(test, yes, no)` a value with the same shape as `test` filled with elements from either `yes` or `no`

`do.call(funname, args)` executes a function call from the name of the function and a list of arguments to be passed to it

2.3 Exercices

Exercice 2.1 (`cbind / rbind`)**.** Utilisez l'aide de `R` pour apprendre le fonctionnement des fonctions `rbind` et `cbind`.

Exercice 2.2 (`if / else`)**.** Reprenez l'exemple de la fonction `myabs` et faîtes en sorte que cette fonction marche si on lui passe un vecteur.

Exercice 2.3 (Boucles `for`)**.** Écrivez une boucle qui permet d'afficher les n premières lettres de l'alphabet.

Exercice 2.4 (Combinaisons)**.**

a) Écrivez une fonction qui calcule $\binom{n}{k}$ et comparez vos résultats avec la fonction de base de `R` nommée `choose`.

b) Testez votre fonction pour $n = 200$ et $k = 50$.

Exercice 2.5 (Dichotomie)**.** Pour mémoire la méthode de dichotomie est basée sur le théorème des valeurs intermédiaires—cf. wikipédia pour ceux qui ont oublié... Voici un pseudo code de la méthode

1. Tant que $b - a > \epsilon$, faire
   + Calculez $m = (a + b)/2$ et $f(m)$;
   + Si $f(a)$ et $f(m)$ sont de même signe alors posez $a \leftarrow m$;
   + Sinon posez $b \leftarrow m$.
2. Renvoyer m.

a) Écrivez une fonction qui implémente la méthode de dichotomie citée plus haut.

b) Testez votre fonction pour trouver les zéros des fonctions suivantes $f(x) = 3x + 1$, $g(x) = x^2 - 1$.

Exercice 2.6 (Vraisemblance)**.**

a) Implémentez des fonctions qui calcule la vraisemblance d'un échantillon issu d'une loi exponentielle.

b) Utilisez la fonction `nlm` pour écrire une fonction calculant l'estimateur du maximum de vraisemblance.

c) Faites tourner votre fonction et vérifiez que vous obtenez le résultat attendu.

Bonus : Comment feriez vous pour obtenir les erreurs standards de cet estimateur ?

Exercice 2.7 (Ensemble de Mandelbrot)**.** L'ensemble de Mandelbrot est une fractale vivant dans $\mathbb{C}$ définit comme l'ensemble des points $c \in \mathbb{C}$ tels que la suite $\{z_n : n \geq 0\}$ donnée par

$$z_{n+1} = z_n^2 + c, \qquad z_0 = 0, \quad n \in \mathbb{N},$$

est bornée. Il n'est pas très difficile de montrer que la suite $\{z_n : n \geq 0\}$ diverge dès lors qu'il existe $n_0 \in \mathbb{N}$ tel que $|z_{n_0}| > 2$.

a) En utilisant la propriété précédente, réfléchissez à une méthode (approximative) pour tester l'appartenance d'un point de $\mathbb{C}$ à l'ensemble de Mandelbrot.

b) Écrire une fonction qui représente graphiquement l'ensemble de Mandelbrot—la fonction `image` pourra vous être utile.

Bonus : Essayez d'obtenir une figure colorée où la couleur dépendra de la vitesse de divergence de la suite $\{z_n : n \geq 0\}$.

Chapitre 3

Programmation avancée

3.1 Debbugage

Dès que l'on commence à faire de la vraie programmation (entendez par là que l'on crée ses propres fonctions), on sera confronté tôt ou tard à des bugs. Il est donc indispensable de savoir **debbuguer**.

Remarque. Avec le temps on devient meilleur et l'on progresse. C'est tout à fait normal de perdre beaucoup de temps au début. C'est la première étape. La deuxième étape vous permettra de connaître les « type de bugs » les plus fréquents—cela peut changer d'un codeur à un autre d'ailleurs. L'étape ultime étant de contourner directement ces types de bugs et ainsi d'être plus à l'aise—et de s'attaquer à des codes plus durs et recommencer le cercle d'apprentissage !

3.1.1 Les outils à disposition

Les fonctions à connaître pour le debbugage sont

`traceback` essaye de remonter l'arborescence du bug ;

`debug/undebug` permet d'exécuter une fonction pas à pas ;

`browser` permet de lancer le debbugage à un endroit spécifique de la fonction.

Nous allons voir quelques exemples permettant de comprendre l'utilisation des trois fonctions citées plus haut.

Exemple 3.1 (Maximum de vraisemblance pour une loi Gamma)**.** Rappelons que la fonction `nlm` de `R` permet de minimiser une fonction—cf. son aide pour plus d'info. Ainsi il est aisé en `R` d'écrire un code qui calcule l'estimateur du maximum de vraisemblance pour cette loi.

```
> fitgamma <- function(obs){

+     n.obs <- length(obs)

+     ## Definissons une fonction qui calcule - log vraisemblance
+     nllik <- function(par){
+         shape <- par[1]
+         rate <- par[2]

+         -sum(dgamma(data, shape, rate, log = TRUE))
```

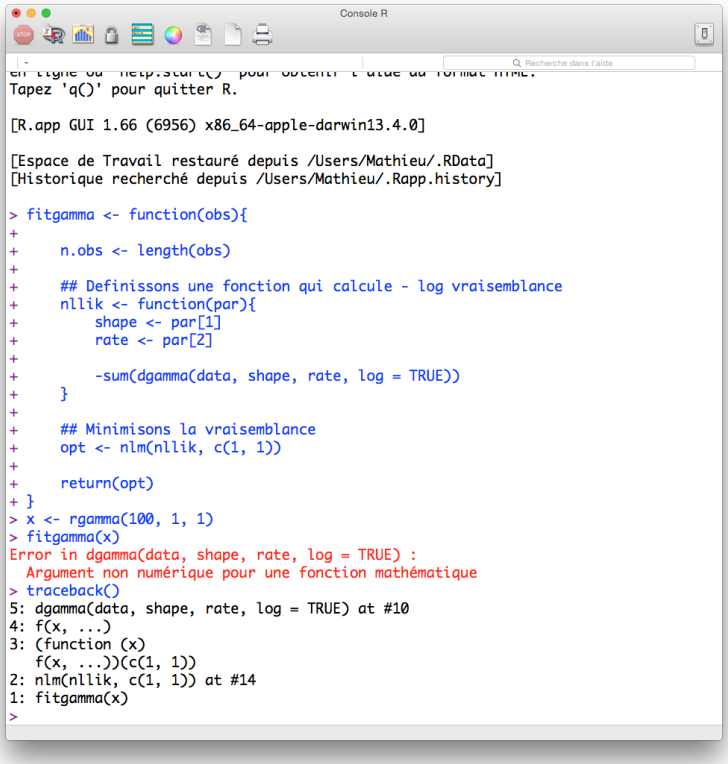


Figure 3.1 – *Utilisation de la fonction* `traceback`.

```
+     }

+     ## Minimisons la vraisemblance
+     opt <- nlm(nllik, c(1, 1))

+     return(opt)
+ }
```

Puis testons notre code sur un exemple artificiel...

```
> n.obs <- 100
> x <- rgamma(n.obs, 1, 1)
>
> fitgamma(x)

Error in dgamma(data, shape, rate, log = TRUE): Argument non num'erique pour
une fonction math'ematique
```

Ouppss visiblement on a une erreur. Utilisons `traceback`.

```
> traceback()
```

La sortie de l'exécution précédente est donnée par la Figure 3.1.

Question 3.1. *Que comprenez vous des messages renvoyés par* `R` *? Où se trouve l'erreur selon vous ?*

Réponse.

□

Les fonctions `debug` et `browser` sont interactives donc difficiles à expliquer sur un format papier. Nous allons reprendre notre exemple précédent et le traiter en live. Voici le point de départ...

```
> x <- rgamma(50, 0.1, 0.5)
> fitgamma(x)

Warning in dgamma(obs, shape, rate, log = TRUE): production de NaN
Warning in nlm(nllik, c(1, 1)): NA / Inf remplac'e par la valeur maximale positive
Warning in dgamma(obs, shape, rate, log = TRUE): production de NaN
Warning in nlm(nllik, c(1, 1)): NA / Inf remplac'e par la valeur maximale positive
Warning in dgamma(obs, shape, rate, log = TRUE): production de NaN
Warning in nlm(nllik, c(1, 1)): NA / Inf remplac'e par la valeur maximale positive
Warning in dgamma(obs, shape, rate, log = TRUE): production de NaN
Warning in nlm(nllik, c(1, 1)): NA / Inf remplac'e par la valeur maximale positive
Warning in dgamma(obs, shape, rate, log = TRUE): production de NaN
Warning in nlm(nllik, c(1, 1)): NA / Inf remplac'e par la valeur maximale positive
Warning in dgamma(obs, shape, rate, log = TRUE): production de NaN
Warning in nlm(nllik, c(1, 1)): NA / Inf remplac'e par la valeur maximale positive

$minimum
[1] -221.2481

$estimate
[1] 0.1267832 0.5126170

$gradient
```

```
[1]  3.447553e-05 -5.115908e-07

$code
[1] 1

$iterations
[1] 14
```

La fonction semble marcher mais nous avons tout plein d'avertissements inquiétants. **Il faut à tout prix viser des codes ne faisant apparaitre aucun message d'avertissement—ou alors ceux que vous aurez *VOUS MÊME* construit.**

3.1.2 Prévenir les bugs

Les fonctions à connaître pour le debbugage sont

`warning` envoie un message d'avertissement à l'utilisateur ;

`try` permet de tester une exécution potentiellement dangereuse ;

`stop` envoie un message d'erreur à l'utilisateur et intérompt l'exécution.

Comme expliqué plus haut, la fonction `warning` permet d'afficher à l'écran (ou plus tard si ils sont trop nombreux) des messages d'avertissement. Voici un exemple

```
> matrix(1:3, 2, 2)

Warning in matrix(1:3, 2, 2): la longueur des donn'ees [3] n'est pas un diviseur
ni un multiple du nombre de lignes [2]

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

Question 3.2. *Que veux dire cet avertissement ?*

Réponse.

□

Son utilisation est très simple :

```
> warning("attention ce que vous avez fait semble louche...")

Warning: attention ce que vous avez fait semble louche...
```

La fonction `try` est quand à elle très utile pour ne pas « casser » un code lors d'instructions potentiellement non valides. Par exemple lorsque l'on souhaite inverser une matrice non inversible :

```
> A <- matrix(1:3, 1, 3)
> iA <- try(solve(A))
> iA

[1] "Error in solve.default(A) : 'a' (1 x 3) doit ^etre carr'ee\n"
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in solve.default(A): 'a' (1 x 3) doit ^etre carr'ee>
```

Enfin la fonction `stop` permet d'arrêter brutalement le code en affichant un message d'erreur. D'ailleurs c'est ce qui se passe lorsque l'on essaye d'inverser la matrice `A` précédente

```
> solve(A)

Error in solve.default(A): 'a' (1 x 3) doit ^etre carr'ee

> stop("C'etait definitivement louche ce que vous avez fait !!!")

Error in eval(expr, envir, enclos): C'etait definitivement louche ce que vous
avez fait !!!
```

Remarque. Ces fonctions seront majoritairement utilisées à l'intérieur de vos **propres** fonction pour aider l'utilisateur à les utiliser correctement mais aussi prévenir certains bugs.

3.2 Optimisation de code

Dans cette section nous allons voir comment le choix d'écriture d'un code peut impacter son efficacité. Rendre un code plus performant s'appelle **optimisation de code**.

Remarque. À ne jamais oublier, avant de s'intéresser à l'optimisation de code il faut bien sûr avoir un **code qui fonctionne** ! Il nous servira par la suite de **test** afin de comparer notre nouveau code avec l'ancien : est il plus rapide ? Donne t il les mêmes résultats ?

Voici quelques conseils pour optimiser votre code :
- Évitez quand c'est possible l'usage de boucle `for` en utilisant les capacités vectorielles de `R` ;
- Sortez toutes les constantes des boucles `for` ;
- Si une succession de `if` / `else if` / ... / `else` est présente, essayez de mettre le cas le plus fréquent en premier ;
- lorsque c'est possible allouer la bonne taille à vos objets.

En tout premier lieu, il faut identifier le « goulot d'étranglement de votre code ». Ceci peut se faire via les fonctions `Rprof` et `summaryRprof`.

Exemple 3.2. Écrire une fonction qui calcule la matrice de covariance empirique à partir d'observations regroupées dans une matrice de la forme

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix},$$

i.e., n observations et p variables. Pour rappel la covariance empirique est données par $S = (s_{i,j})_{i,j=1,\dots,p}$ avec

$$s_{i,j} = \frac{1}{n-1} \sum_{k=1}^{n} (x_{k,i} - \bar{x}_i)(y_{k,j} - \bar{x}_j),$$

et

$$\bar{x}_i = \frac{1}{n} \sum_{k=1}^{n} x_{k,i}.$$

```
> mycovariance <- function(obs){
+     n.obs <- nrow(obs)
+     n.var <- ncol(obs)

+     cov <- matrix(0, n.var, n.var)
+     for (i in 1:n.var)
+         for (j in 1:n.var)
+             cov[i,j] <- sum((obs[,i] - mean(obs[,i])) *
+                             (obs[,j] - mean(obs[,j]))) /(n.obs - 1)

+     return(cov)
+ }
>
```

```
> data <- matrix(rnorm(5 * 100), 100)
> mycovariance(data)

            [,1]        [,2]       [,3]        [,4]       [,5]
[1,]  1.06901807 -0.10946653 0.23207044  0.08182961 0.10810823
[2,] -0.10946653  0.86376388 0.03170612 -0.01101673 0.05940567
[3,]  0.23207044  0.03170612 1.28406355  0.01836325 0.24365230
[4,]  0.08182961 -0.01101673 0.01836325  0.93346639 0.08066951
[5,]  0.10810823  0.05940567 0.24365230  0.08066951 1.13263346

> ## a comparer avec la fonction de base de R
> cov(data)

            [,1]        [,2]       [,3]        [,4]       [,5]
[1,]  1.06901807 -0.10946653 0.23207044  0.08182961 0.10810823
[2,] -0.10946653  0.86376388 0.03170612 -0.01101673 0.05940567
[3,]  0.23207044  0.03170612 1.28406355  0.01836325 0.24365230
[4,]  0.08182961 -0.01101673 0.01836325  0.93346639 0.08066951
[5,]  0.10810823  0.05940567 0.24365230  0.08066951 1.13263346

> ## Test sur un exemple plus "gourmand"
> data <- matrix(rnorm(50 * 100), 100)
> Rprof("profile.Rout")##cree un fichier de profilage nomme profile.Rout
> ans <- mycovariance(data)
> Rprof(NULL)##indique que le profilage est fini
> summaryRprof("profile.Rout")##affiche le profilage

$by.self
               self.time self.pct total.time total.pct
"mycovariance"      0.02    33.33       0.06    100.00
"mean"              0.02    33.33       0.04     66.67
"mean.default"      0.02    33.33       0.02     33.33

$by.total
                      total.time total.pct self.time self.pct
"mycovariance"              0.06    100.00      0.02    33.33
"<Anonymous>"               0.06    100.00      0.00     0.00
"block_exec"                0.06    100.00      0.00     0.00
"call_block"                0.06    100.00      0.00     0.00
"doTryCatch"                0.06    100.00      0.00     0.00
"eval"                      0.06    100.00      0.00     0.00
"evaluate_call"             0.06    100.00      0.00     0.00
"handle"                    0.06    100.00      0.00     0.00
"in_dir"                    0.06    100.00      0.00     0.00
"knit"                      0.06    100.00      0.00     0.00
"process_file"              0.06    100.00      0.00     0.00
"process_group"             0.06    100.00      0.00     0.00
"process_group.block"       0.06    100.00      0.00     0.00
"try"                       0.06    100.00      0.00     0.00
"tryCatch"                  0.06    100.00      0.00     0.00
"tryCatchList"              0.06    100.00      0.00     0.00
```

```
"tryCatchOne"               0.06    100.00      0.00     0.00
"withCallingHandlers"       0.06    100.00      0.00     0.00
"withVisible"               0.06    100.00      0.00     0.00
"mean"                      0.04     66.67      0.02    33.33
"mean.default"              0.02     33.33      0.02    33.33

$sample.interval
[1] 0.02

$sampling.time
[1] 0.06
```

Remarque. Dans la sortie précédente, ne tenez pas compte de certaine ligne qui apparaissent par rapport à mon document de cours. Dans une session `R` vous n'auriez donc pas toutes ces lignes.

Question 3.3. *Proposer des améliorations à ce code.*

Réponse.

□

3.3 Méthodes

Comme je vous l'ai déjà dit ceci n'est pas un cours de programmation donc je vais vous apprendre le fonctionnement des classes et des méthodes de manière pragmatique.

Remarque. Il existe deux manières différentes pour cela en `R` l'approche `S3` et l'approche `S4`. Bien que moins récente, nous ne verrons que l'approche `S3` beaucoup plus simple à

mettre en oeuvre.

Brièvement,

— une classe peut être perçu comme une étiquette que l'on peut coller sur des objets ;
— les méthodes sont des fonctions qui ne vont s'appliquer que sur un objet d'une certaine classe, i.e., ayant une étiquette spécifique.

Exemple 3.3. Liste de course C'est un exemple bien inutile mais qui je pense est très pédagogique. Dans `R` j'ai créé l'objet suivant contenant ma liste de course

```
> mylist <- data.frame(item = c("banane", "kiwi", "eau", "orangina"),
+                      category = c("fruit", "fruit", "boisson", "boisson"))
```

Si je demande un résumé de cette liste de course, j'obtiens

```
> summary(mylist)## pas terrible non ?

       item      category
 banane  :1   boisson:2
 eau     :1   fruit  :2
 kiwi    :1
 orangina:1

> class(mylist)##affiche la classe

[1] "data.frame"
```

Essayons de faire mieux. Je commence par affecter une classe nommée `shoppinglist` à mon objet `mylist`

```
> class(mylist) <- "shoppinglist"
```

puis je crée une méthode qui me fait un résumé plus approprié pour une liste de course :

```
> summary.shoppinglist <- function(object, ...){
+     n.fruit <- sum(object$category == "fruit")
+     n.boisson <- sum(object$category == "boisson")

+     cat("Dans votre liste de course, il y a", n.fruit, "fruit(s) et",
+         n.boisson, "boisson(s).\n")
+ }
>
> summary(mylist)

Dans votre liste de course, il y a 2 fruit(s) et 2 boisson(s).

> class(mylist)

[1] "shoppinglist"
```

Question 3.4. *Que pensez vous du code ci-dessous ?*

```
> summary.shoppinglist <- function(x){
+     n.fruit <- sum(x$category == "fruit")
+     n.boisson <- sum(x$category == "boisson")

+     cat("Dans votre liste de course, il y a", n.fruit, "fruit(s) et",
+         n.boisson, "boisson(s).\n")
+ }
> summary(mylist)

Dans votre liste de course, il y a 2 fruit(s) et 2 boisson(s).
```

Réponse.

□

Remarque. Bien que l'exemple ci dessus soit vraiment bidon, les classes et méthodes peuvent grandement améliorer l'expérience utilisateur et les possibilités sont infinies. Je pourrais par exemple faire des graphiques spécifiques le tout en utilisant **apparemment** la même syntaxe que `R` via la fonction `plot` ! C'est donc chaudement recommandé !!!!!

3.4 Application

L'exemple que nous allons traiter ici est issu de la géostatistique et concerne l'estimation d'un processus Gaussien. **Il n'est pas nécessaire que vous compreniez toutes les maths qu'il y aura derrière mais juste la partie optimisation de code ! ! !**

De manière pas très formelle, vous pouvez pensez qu'un processus Gaussien $\{Y(x) \colon x \in \mathcal{X}\}$, $\mathcal{X} \subset \mathbb{R}^d$, $d \geq 1$,est une fonction (aléatoire)

$$\begin{aligned} \mathcal{X} &\longrightarrow \mathbb{R} \\ x &\longmapsto Y(x) \end{aligned}$$

telle que pour tout $x_1, \ldots, x_k \in \mathcal{X}$, $k \geq 1$, le vecteur aléatoire $\{Y(x_1), \ldots, Y(x_k)\}$ suit une loi Normale multivariée d'espérance $\mu = (\mu_1, \ldots, \mu_k)$ avec $\mu_j = \mathbb{E}\{Y(x_j)\}$, $j = 1, \ldots, k$ et de matrice de covariance $\Sigma = (\sigma_{i,j})_{i,j=1,\ldots,k}$ avec $\sigma_{i,j} = \mathrm{Cov}\{Y(x_i), Y(x_j)\}$.

Les processus Gaussiens sont très fréquemment utilisée en statistiques spatiales (modélisation de champs de températures, de précipitations, de la topographie) mais aussi en finance pour modéliser les actions—modèle de Black–Scholes entre autre.

Si l'on parle de températures, clairement il nous est impossible de mesurer cette température continûment sur notre région d'étude $\mathcal{X}$; de sorte que l'on dispose d'observations uniquement en un nombre fini de stations météorologiques ayant pour coordonnées spatiales $x_1, \ldots, x_k$.

Si l'on note $\mathbf{y} = (y_1, \ldots, y_k)$ où $y_j \equiv y(x_j)$ correspond à l'observation relevée en la station x_j, $j = 1, \ldots, k$, alors la vraisemblance s'écrit

$$L(\mu, \Sigma; \mathbf{y}) = (2\pi)^{-k/2} |\Sigma|^{-1/2} \exp\left\{ -\frac{(\mathbf{y} - \mu)^\top \Sigma^{-1} (\mathbf{y} - \mu)}{2} \right\}.$$

Pour faire simple nous allons faire quelques hypothèses supplémentaires :
— la moyenne du processus Gaussien est constante, i.e., $\mu = (\mu, \ldots, \mu)$;
— et la covariance du processus Gaussien est régie par une fonction de type exponentielle, i.e., $\sigma_{i,j} = \tau \exp(-\|x_i - x_j\|/\lambda)$, $\tau, \lambda > 0$.

```
> n.obs <- 100
> n.site <- 50
>
> coord <- matrix(runif(2 * n.site, 0, 100), n.site)
> Sigma <- 5 * exp(-as.matrix(dist(coord)) / 30)
> obs <- t(t(chol(Sigma)) %*% matrix(rnorm(n.site * n.obs), n.site)) + 20
```

```
> nllik1 <- function(par, obs, coord){
+     mu <- par[1]
+     tau <- par[2]
+     lambda <- par[3]

+     if ((tau <= 0) || (lambda <= 0))
+         ## Non feasible values for these parameters
+         return(Inf)

+     distance <- as.matrix(dist(coord))
+     Sigma <- tau * exp(-distance / lambda)

+     n.obs <- nrow(obs)
+     n.site <- nrow(coord)

+     nllik <- 0
+     for (i in 1:n.obs)
+         nllik <- nllik + 0.5 * n.site * log(2 * pi) + 0.5 * log(det(Sigma)) +
+             0.5 * (obs[i,] - mu) %*% solve(Sigma) %*% (obs[i,] - mu)

+     return(nllik)
+ }
>
> init <- c(20, 5, 30)
> system.time(fit1 <- nlm(nllik1, init, obs = obs, coord = coord))

Warning in nlm(nllik1, init, obs = obs, coord = coord): NA / Inf remplac'e
par la valeur maximale positive

   user  system elapsed
  2.121   0.009   2.130

> fit1

$minimum
[1] 8601.066

$estimate
```

```
[1] 20.051461  4.896485 29.719253

$gradient
[1] -1.239181e-04 -7.912712e-05  1.836173e-06

$code
[1] 1

$iterations
[1] 11
```

```
> nllik2 <- function(par, obs, coord){
+     mu <- par[1]
+     tau <- par[2]
+     lambda <- par[3]

+     if ((tau <= 0) || (lambda <= 0))
+         ## Non feasible values for these parameters
+         return(Inf)

+     distance <- as.matrix(dist(coord))
+     Sigma <- tau * exp(-distance / lambda)
+     iSigma <- solve(Sigma)
+     det <- det(Sigma)

+     n.obs <- nrow(obs)
+     n.site <- nrow(coord)

+     nllik <- 0.5 * n.obs * (n.site * log(2 * pi) + log(det))
+     for (i in 1:n.obs)
+         nllik <- nllik + 0.5 * (obs[i,] - mu) %*% iSigma %*% (obs[i,] - mu)

+     return(nllik)

+ }
>
> system.time(fit2 <- nlm(nllik2, init, obs = obs, coord = coord))

Warning in nlm(nllik2, init, obs = obs, coord = coord): NA / Inf remplac'e
par la valeur maximale positive

   user  system elapsed
  0.121   0.003   0.125

> fit2

$minimum
[1] 8601.066

$estimate
```

```
[1] 20.051461  4.896485 29.719254

$gradient
[1] -1.250067e-04 -7.912712e-05  2.264613e-06

$code
[1] 1

$iterations
[1] 11
```

```
> nllik3 <- function(par, obs, coord){
+     mu <- par[1]
+     tau <- par[2]
+     lambda <- par[3]

+     if ((tau <= 0) || (lambda <= 0))
+         ## Non feasible values for these parameters
+         return(Inf)

+     distance <- as.matrix(dist(coord))
+     Sigma <- tau * exp(-distance / lambda)
+     Sigma.chol <- chol(Sigma) ## Sigma.chol^T Sigma.chol = Sigma

+     ## Rmk: |Sigma| = |Sigma.chol^T| |Sigma.chol| = |Sigma.chol|^2
+     ## But since Sigma.chol is triangular the determinant is just
+     ##the prod. of the diagonal elements... Hence
+     log.det <- 2 * sum(log(diag(Sigma.chol)))

+     ## Rmk: (y - mu)^T Sigma^(-1) (y - mu) can be computed by
+     ## (y - mu)^T Sigma.chol^(-1) Sigma.chol^(-T) (y - mu)
+     ## so we only need to compute dummy = Sigma.chol^(-T) (y - mu), i.e.,
+     ## solve w.r.t. dummy the linear system Sigma.chol^T dummy = y - mu
+     dummy <- backsolve(Sigma.chol, t(obs - mu), transpose = TRUE)
+     mahal <- sum(dummy^2)

+     nllik <- 0.5 * (n.obs * (n.site * log(2 * pi) + log.det) + mahal)
+     return(nllik)
+ }
>
> system.time(fit3 <- nlm(nllik3, init, obs = obs, coord = coord))

Warning in nlm(nllik3, init, obs = obs, coord = coord): NA / Inf remplac'e
par la valeur maximale positive

   user  system elapsed
  0.025   0.000   0.025

> fit3
```

```
$minimum
[1] 8601.066

$estimate
[1] 20.051461  4.896485 29.719255

$gradient
[1] -1.248253e-04 -7.838414e-05  2.387024e-06

$code
[1] 1

$iterations
[1] 11
```

3.5 Exercices

Exercice 3.1 (Processus Gaussien)**.** En reprenant le code que nous avons fait pour l'ajustement d'un processus Gaussien. Construisez une fonction qui ajuste le processus gaussien, puis créez une méthode afin que l'affichage du modèle ajusté soit plus sympathique à lire. Par exemple, votre méthode pourra affiché les estimations des paramètres, leurs erreurs standards ainsi que le nombre d'itérations faites lors de l'optimisation numérique.

Exercice 3.2 (Ajustement d'une loi Gamma)**.**

a) Reprenez la fonction `fitgamma` que nous avons créer afin de renvoyer les erreurs standards de l'estimateur du maximum de vraisemblance.

b) Créer une classe pour votre loi gamma ajustée ainsi qu'une méthode d'affichage qui renvoie l'estimation, les erreurs standards associées et le critère AIC.

Exercice 3.3 (La fonction `apply`)**.**

a) Apprenez comment utiliser la fonction `apply` de `R`.

b) Créez une matrice de taille $(500, 3)$ dans laquelle la première colonne set un 500-échantillon d'une $N(0, 1)$, la deuxième un 500–échantillon d'une $N(0, 4)$ et la dernière un 500–échantillon d'une $N(0, 9)$.

c) Calculez les quantiles à 0.025 et 0.975 pour chacun de ces échantillons à l'aide de la fonction `quantile`.

d) Si vous avez écrit plusieurs lignes de code pour répondre à la question précédente, revenez à la question a).

Exercice 3.4 (QQ-plot)**.** Rappelons (ou pas !) qu'un QQ-plot peut servir à deux choses distinctes :

— comparer deux échantillons de mêmes tailles pour voir s'ils sont issus de la même loi ;
— vérifier graphiquement si un échantillon suit une loi de probabilité donnée.

Dans cet exercice, nous allons nous intéresser au second cas seulement. Soit $x_{(1)} < \cdots < x_{(n)}$ des réalisations indépendantes d'une variable aléatoire dont la fonction de répartition est supposée être F. Le QQ-plot consiste alors à représenter les points

$$\left\{\left(F^{-1}\left(\frac{i}{n+1}\right), x_{(i)}\right) : i = 1, \ldots, n\right\}.$$

a) Écrivez une fonction qui compare un n-échantillon à une loi $N(0,1)$.

b) Testez votre fonction sur des 100-échantillons issue d'une loi de Student à 5, 10, 30, 40 degrée de liberté. Que constatez vous ?

Exercice 3.5 (Temps de calcul)**.** Utilisez la fonction `system.time` afin d'évaluer le gain de performance entre les deux approches suivantes :

```
> A <- matrix(rnorm(500 * 100), 500)
>
> ans1 <- rep(NA, 500)
> for (i in 1:ncol(A))
+     ans1[i] <- sd(A[,i])
>
> ans2 <- apply(A, 2, sd)
```

Concluez.

Conclusion

Ce cours est maintenant terminé mais sachez que votre apprentissage de `R` est loin d'être terminé. Idéalement il faudrait que vous continuiez à travailler avec `R`. Pour cela vous pouvez utiliser les exercices de stats de vos profs, les faire sur le papier puis essayer avec `R`.

Notez que nous n'avons pas vraiment abordé toutes les capacités statistiques de `R` car vous n'avez pas encore une culture proba/stat assez developpée. Cela dit la plupart des thèmes que vous allez rencontrer en cours sont déjà implémentés de base, e.g., modèles linéaires (généralisés ou non), modèle de survie, ACP, ...