# R pour les débutants

Emmanuel Paradis

*Institut des Sciences de l'Évolution*
*Université Montpellier II*
*F-34095 Montpellier cédex 05*
*France*

E-mail : *paradis@isem.univ-montp2.fr*

Je remercie Julien Claude, Christophe Declercq, Élodie Gazave, Friedrich Leisch et Mathieu Ros pour leurs commentaires et suggestions sur des versions précédentes de ce document. J'exprime également ma reconnaissance à tous les membres du *R Development Core Team* pour leurs efforts considérables dans le développement de R et dans l'animation de la liste de discussion électronique 'r-help'. Merci également aux utilisateurs de R qui par leurs questions ou commentaires m'ont aidé à écrire "R pour les débutants".

© 2002, Emmanuel Paradis (16 août 2002)

## Table des matières

# 1 Préambule

Le but du présent document est de fournir un point de départ pour les novices intéressés par R. J'ai fait le choix d'insister sur la compréhension du fonctionnement de R, bien sûr dans le but d'une utilisation courante plutôt qu'experte. Les possibilités offertes par R étant très vastes, il est utile pour le débutant d'assimiler certaines notions et concepts afin d'évoluer plus aisément par la suite. J'ai essayé de simplifier au maximum les explications pour les rendre accessibles à tous, tout en donnant les détails utiles, parfois sous forme de tableaux.

R est un système d'analyse statistique et graphique créé par Ross Ihaka et Robert Gentleman[1]. R est à la fois un logiciel et un langage qualifié de dialecte du langage S créé par AT&T Bell Laboratories. S est disponible sous la forme du logiciel S-PLUS commercialisé par la compagnie Insightful[2]. Il y a des différences importantes dans la conception de R et de S : ceux qui veulent en savoir plus sur ce point peuvent se reporter à l'article de Ihaka & Gentleman (1996) ou au R-FAQ[3] dont une copie est également distribuée avec le logiciel.

R est distribué librement sous les termes de la *GNU General Public Licence*[4] ; son développement et sa distribution sont assurés par plusieurs statisticiens rassemblés dans le *R Development Core Team*.

R est disponible sous plusieurs formes : le code écrit principalement en C (et certaines routines en Fortran), surtout pour les machines Unix et Linux, ou des exécutables précompilés pour Windows, Linux (Debian, Mandrake, RedHat, SuSe), Macintosh et Alpha Unix. Les fichiers pour installer R, à partir du code ou des exécutables, sont distribués à partir du site internet du *Comprehensive R Archive Network* (CRAN)[5] où se trouvent aussi les instructions à suivre pour l'installation sur chaque système. En ce qui concerne les distributions de Linux (Debian, ...), les exécutables sont généralement disponibles pour les versions les plus récentes de ces distributions *et* de R ; consultez le site du CRAN si besoin.

R comporte de nombreuses fonctions pour les analyses statistiques et les graphiques ; ceux-ci sont visualisés immédiatement dans une fenêtre propre et peuvent être exportés sous divers formats (jpg, png, bmp, ps, pdf, emf, pictex, xfig ; les formats disponibles peuvent dépendre du système d'exploitation). Les résultats des analyses statistiques sont affichés à l'écran, certains résultats partiels (valeurs de $P$, coefficients de régression, résidus, ...) peuvent être sauvés à part, exportés dans un fichier ou utilisés dans des analyses ultérieures.

Le langage R permet, par exemple, de programmer des boucles qui vont analyser successivement différents jeux de données. Il est aussi possible de combiner dans le même programme différentes fonctions statistiques pour réaliser des analyses plus complexes. Les utilisateurs de R peuvent bénéficier des nombreux programmes écrits pour S et disponibles sur internet[6], la plupart de ces programmes étant directement utilisables avec R.

De prime abord, R peut sembler trop complexe pour une utilisation par un non-spécialiste. Ce n'est pas forcément le cas. En fait, R privilégie la flexibilité. Alors qu'un logiciel classique affichera directement les résultats d'une analyse, avec R ces résultats sont stockés dans un "objet", si bien qu'une analyse peut être faite sans qu'aucun résultat ne soit affiché. L'utilisateur peut être déconcerté par ceci, mais cette facilité se révèle extrêmement utile. En effet, l'utilisateur peut alors extraire uniquement la portion des résultats qui l'intéressent. Par exemple, si l'on doit faire une série de 20 régressions et que l'on veuille comparer les coefficients des différentes régressions, R pourra afficher uniquement les coefficients estimés : les résultats tiendront donc sur une ligne, alors

[1] Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5 : 299–314.

[2] voir http://www.insightful.com/products/splus/default.html pour plus d'information

[3] http://cran.r-project.org/doc/FAQ/R-FAQ.html

[4] pour plus d'infos : http://www.gnu.org/

[5] http://cran.r-project.org/

[6] par exemple : http://stat.cmu.edu/S/

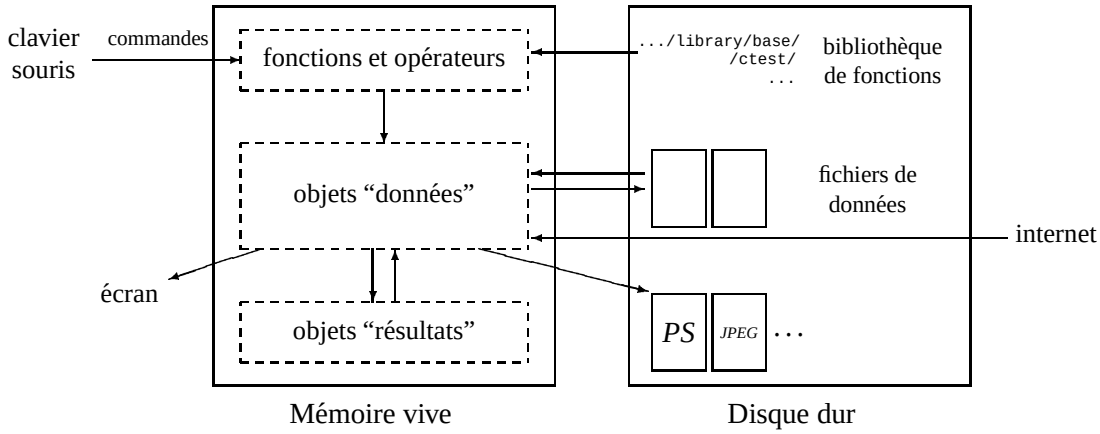



FIG. 1 – Une vue schématique du fonctionnement de R.

qu’un logiciel plus classique pourra ouvrir 20 fenêtres de résultats. On verra d’autres exemples illustrant la flexibilité d’un système comme R vis-à-vis des logiciels classiques.

## 2 Quelques concepts avant de démarrer

Une fois R installé sur votre ordinateur, il suffit de lancer l’exécutable correspondant pour démarrer le programme. L’attente de commandes (par défaut en forme de crochet ‘>’) apparait alors indiquant que R est prêt à exécuter les commandes. Sous Windows, certaines commandes (accès à l’aide, ouverture de fichiers, . . .) peuvent être exécutées par les menus. L’utilisateur novice a alors toutes les chances de se demander “Je fais quoi maintenant ?” Il est en effet très utile d’avoir quelques idées sur le fonctionnement de R lorsqu’on l’utilise pour la première fois : c’est ce que nous allons voir maintenant.

Nous allons dans un premier temps voir schématiquement comment R travaille. Ensuite nous décrirons l’opérateur “assigner” qui permet de créer des objets, puis comment gérer basiquement les objets en mémoire, et finalement comment utiliser l’aide en ligne qui, contrairement à beaucoup de logiciels, est extrêmement utile dans une utilisation courante.

### 2.1 Comment R travaille

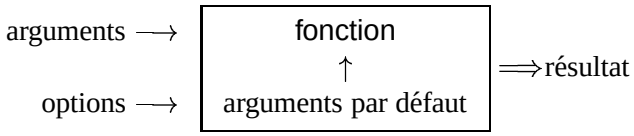
R est un langage *orienté-objet* : voici une expression bien compliquée qui masque toute la simplicité et la flexibilité de R. Le fait que R soit un langage peut effrayer plus d’un utilisateur potentiel pensant “Je ne sais pas programmer”. Cela ne devrait pas être le cas pour deux raisons. D’abord, R est un langage interprété et pas compilé, c’est-à-dire que les commandes tapées au clavier sont directement exécutées sans qu’il soit besoin de construire un programme complet comme cela est le cas pour la plupart des langages informatiques (C, Fortran, Pascal, . . .).

Ensuite, la syntaxe de R est très simple et intuitive. Par exemple, une régression linéaire pourra être faite avec la commande `lm(y ~ x)`. Avec R, une fonction, pour être exécutée, s’écrit *toujours* avec des parenthèses, même si elles ne contiennent rien (par exemple `ls()`). Si l’utilisateur tape le nom de la fonction sans parenthèses, R affichera le contenu des instructions de cette fonction. Dans la suite de ce document, les noms des fonctions sont généralement écrits avec des parenthèses pour les distinguer des autres objets sauf si le texte indique clairement qu’il s’agit d’une fonction.

*Orienté-objet* signifie que les variables, les données, les fonctions, les résultats, etc. sont stockés dans la mémoire de l’ordinateur sous forme d’*objets* qui ont chacun un *nom*. L’utilisa-

teur va agir sur ces objets avec des *opérateurs* (arithmétiques, logiques et de comparaison) et des *fonctions* (qui sont elles-mêmes des objets).

L'utilisation des opérateurs est relativement intuitive, on en verra les détails plus loin (p. 21). Une fonction de R peut être schématisée comme suit :




Les arguments peuvent être des objets ("données", formules, expressions, ...) dont certains peuvent être définis par défaut dans la fonction ; ces valeurs par défaut peuvent être modifiées par l'utilisateur avec les options. Une fonction de R peut ne nécessiter aucun argument de la part de l'utilisateur : soit tous les arguments sont définis par défaut (et peuvent être changés avec les options), ou soit aucun argument n'est défini. On verra plus en détail l'utilisation et la construction des fonctions (p. 53). La présente description est pour le moment suffisante pour comprendre comment R opère.

Toutes les actions de R sont effectuées sur les objets présents dans la mémoire vive de l'ordinateur : aucun fichier temporaire n'est utilisé (FIG. 1). Les lectures et écritures de fichiers sont utilisées pour la lecture et l'enregistrement des données et des résultats (graphiques, ...). L'utilisateur exécute des fonctions par l'intermédiaire de commandes. Les résultats sont affichés directement à l'écran, ou stockés dans un objet, ou encore écrits sur le disque (en particulier pour les graphiques). Les résultats étant eux-mêmes des objets, ils peuvent être considérés comme des données et être analysés à leur tour. Les fichiers de données peuvent être lus sur le disque de l'ordinateur local ou sur un serveur distant via internet.

Les fonctions disponibles sont stockées dans une bibliothèque localisées sur le disque dans le répertoire R_HOME/library (R_HOME désignant le répertoire où R est installé). Ce répertoire contient des *packages* de fonctions, eux-mêmes présents sur le disque sous forme de répertoires. Le package nommé `base` est en quelque sorte le cœur de R et contient les fonctions de base du langage pour la lecture et la manipulation des données, des fonctions graphiques, et certaines fonctions statistiques (modèles linéaires et analyse de variance notamment). Chaque package a un répertoire nommé R avec un fichier qui a pour nom celui du package (par exemple, pour `base`, ce sera le fichier R_HOME/library/base/R/base). Ce fichier est au format ASCII et inclut les fonctions du package.

La commande la plus simple consiste à taper le nom d'un objet pour afficher son contenu. Par exemple, si un objet `n` contient la valeur 10 :

```
> n
[1] 10
```

Le chiffre 1 entre crochets indique que l'affichage commence au premier élément de `n`. Cette commande est une utilisation implicite de la fonction `print` et l'exemple ci-dessus est identique à `print(n)` (dans certaines situations, la fonction `print` doit être utilisée de façon explicite, par exemple au sein d'une fonction ou d'une boucle).

Le nom d'un objet doit obligatoirement commencer par une lettre (A-Z et a-z) et peut comporter des lettres, des chiffres (0-9), et des points (.). Il faut savoir aussi que R distingue, pour les noms des objets, les majuscules des minuscules, c'est-à-dire que `x` et `X` pourront servir à nommer des objets distincts (même sous Windows).

## 2.2 Créer, lister et effacer les objets en mémoire

Un objet peut être créé avec l'opérateur "assigner" qui s'écrit avec une flèche composée d'un signe moins accollé à un crochet, ce symbole pouvant être orienté dans un sens ou dans l'autre :

```
> n <- 15
> n
[1] 15
> 5 -> n
> n
[1] 5
> x <- 1
> X <- 10
> x
[1] 1
> X
[1] 10
```

Si l'objet existe déjà, sa valeur précédente est effacée (la modification n'affecte que les objets en mémoire vive, pas les données sur le disque). La valeur ainsi donnée peut être le résultat d'une opération et/ou d'une fonction :

```
> n <- 10 + 2
> n
[1] 12
> n <- 3 + rnorm(1)
> n
[1] 2.208807
```

La fonction `rnorm(1)` génère une variable aléatoire normale de moyenne zéro et variance unité (p. 15). On peut simplement taper une expression sans assigner sa valeur à un objet, le résultat est alors affiché à l'écran mais n'est pas stocké en mémoire :

```
> (10 + 2) * 5
[1] 60
```

Dans nos exemples, on omettra l'assignement si cela n'est pas nécessaire à la compréhension.

La fonction `ls` permet d'afficher une liste simple des objets en mémoire, c'est-à-dire que seuls les noms des objets sont affichés.

```
> name <- "Carmen"; n1 <- 10; n2 <- 100; m <- 0.5
> ls()
[1] "m"    "n1"   "n2"   "name"
```

Notons l'usage du point-virgule pour séparer des commandes distinctes sur la même ligne. Si l'on veut lister uniquement les objets qui contiennent un caractère donné dans leur nom, on utilisera alors l'option `pattern` (qui peut s'abréger avec `pat`) :

```
> ls(pat = "m")
[1] "m"    "name"
```

Pour restreindre la liste aux objets dont le nom commence par le caractère en question :

```
> ls(pat = "^m")
[1] "m"
```

La fonction `ls.str()` affiche des détails sur les objets en mémoire :

```
> ls.str()
m :  num 0.5
n1 :  num 10
n2 :  num 100
name :  chr "Carmen"
```

L'option `pattern` peut également être utilisée comme avec `ls()`. Une autre option utile de `ls.str()` est `max.level` qui spécifie le niveau de détails de l'affichage des objets composites. Par défaut, `ls.str()` affiche les détails de tous les objets contenus en mémoire, y compris les colonnes des jeux de données, matrices et listes, ce qui peut faire un affichage très long. On évite d'afficher tous les détails avec l'option `max.level = -1` :

```
> M <- data.frame(n1, n2, m)
> ls.str(pat = "M")
M : 'data.frame':       1 obs. of  3 variables:
 $ n1: num 10
 $ n2: num 100
 $ m : num 0.5
> ls.str(pat="M", max.level=-1)
M : 'data.frame':       1 obs. of  3 variables:
```

Pour effacer des objets de la mémoire, on utilise la fonction `rm()` : `rm(x)` pour effacer l'objet `x`, `rm(x, y)` pour effacer les objets `x` et `y`, `rm(list=ls())` pour effacer tous les objets en mémoire ; on pourra ensuite utiliser les mêmes options citées pour `ls()` pour effacer sélectivement certains objets : `rm(list=ls(pat = "^m"))`.

### 2.3 L'aide en ligne

L'aide en ligne de R est extrêment utile pour l'utilisation des fonctions. L'aide est disponible directement pour une fonction donnée, par exemple :

```
> ?lm
```

affichera, dans R, l'aide pour la fonction `lm()` (*linear model*). La commande `help(lm)` ou `help("lm")` aura le même effet. C'est cette fonction qu'il faut utiliser pour accéder à l'aide avec des caractères non-conventionnels :

```
> ?*
Error: syntax error
> help("*")
Arithmetic               package:base               R Documentation

Arithmetic Operators
...
```

L'appel de l'aide ouvre une page (le comportement exact dépend du système d'exploitation) avec sur la première ligne des informations générales dont le nom du package où se trouvent la (ou les) fonction(s) ou les opérateurs documentés. Ensuite vient un titre suivi de paragraphes qui chacun apporte une information bien précise.

**Description:** brève description.

**Usage:** pour une fonction donne le nom avec tous ses arguments et les éventuelles valeurs par défaut (options) ; pour un opérateur donne l'usage typique.

**Arguments:** pour une fonction détaille chacun des arguments.

**Details:** description détaillée.

**Value:** le cas échéant, le type d'objet retourné par la fonction ou l'opérateur.

**See Also:** autres rubriques d'aide proches ou similaires à celle documentée.

**Examples:** des exemples qui généralement peuvent être exécutés sans ouvrir l'aide avec la fonction `examples()`.

Pour un débutant, il est conseillé de regarder le paragraphe **Examples:**. En général, il est utile de lire attentivement le paragraphe **Arguments:**. D'autres paragraphes peuvent être rencontrés, tel **Note:**, **References:** ou **Author(s):**.

Par défaut, la fonction `help` ne recherche que dans les packages chargés en mémoire. L'option `try.all.packages`, dont le défaut est `FALSE`, permet de chercher dans tous les packages si sa valeur est `TRUE` :

```
> help("bs")
Error in help("bs") : No documentation for ‘bs’ in specified
packages and libraries:
  you could try ‘help.search("bs")’
> help("bs", try.all.packages = TRUE)
  topic ‘bs’ is not in any loaded package
  but can be found in package ‘splines’ in library ‘D:/rw1041/library’
```

On peut ouvrir l'aide au format html (qui sera lu avec Netscape, par exemple) en tapant :

```
> help.start()
```

Une recherche par mots-clefs est possible avec cette aide html. La rubrique **See Also:** contient ici des liens hypertextes vers les pages d'aide des autres fonctions. La recherche par mots-clefs est également possible avec la fonction `help.search` mais celle-ci est encore expérimentale (version 1.5.0 de R).

La fonction `apropos` trouve les fonctions qui contiennent dans leur nom la chaîne de caractère passée en argument ; seuls les packages chargés en mémoire sont cherchés :

```
> apropos(help)
[1] "help"           "help.search"    "help.start"
[4] "link.html.help"
```

# 3 Les données avec R

## 3.1 Les objects

Nous avons vu que R manipule des objets : ceux-ci sont caractérisés bien sûr par leur nom et leur contenu, mais aussi par des *attributs* qui vont spécifier le type de données représenté par un objet. Afin de comprendre l'utilité de ces attributs, considérons une variable qui prendrait les valeurs 1, 2 ou 3 : une telle variable peut représenter une variable entière (par exemple, le nombre d'œufs dans un nid), ou le codage d'une variable catégorique (par exemple, le sexe dans certaines populations de crustacés : mâle, femelle ou hermaphrodite).

Il est clair que le traitement statistique de cette variable ne sera pas le même dans les deux cas : avec R, les attributs de l'objet donnent l'information nécessaire. Plus techniquement, et plus généralement, l'action d'une fonction sur un objet va dépendre des attributs de celui-ci.

Les objets ont tous deux attributs *intrinsèques* : le *mode* et la *longueur*. Le mode est le type des éléments d'un objet ; il en existe quatre principaux : numérique, caractère, complexe[7], et logique (`FALSE` ou `TRUE`). D'autres modes existent qui ne représentent pas des données, par exemple

[7] Il sera peu fait état du mode complexe dans ce document.

fonction ou expression. La longueur est le nombre d'éléments de l'objet. Pour connaître le mode et la longueur d'un objet on peut utiliser, respectivement, les fonctions `mode` et `length` :

```
> x <- 1
> mode(x)
[1] "numeric"
> length(x)
[1] 1
> A <- "Gomphotherium"; compar <- TRUE; z <- 1i
> mode(A); mode(compar); mode(z)
[1] "character"
[1] "logical"
[1] "complex"
```

Quelque soit le mode, les valeurs manquantes sont représentées par `NA` (*not available*). Une valeur numérique très grande peut être spécifiée avec une notation exponentielle :

```
> N <- 2.1e23
> N
[1] 2.1e+23
```

R représente correctement des valeurs numériques qui ne sont pas finies, telles que $\pm\infty$ avec `Inf` et `-Inf`, ou des valeurs qui ne sont pas des nombres avec `NaN` (*not a number*).

```
> x <- 5/0
> x
[1] Inf
> exp(x)
[1] Inf
> exp(-x)
[1] 0
> x - x
[1] NaN
```

Une valeur de mode caractère est donc entrée entre des guillemets doubles `"`. Il est possible d'inclure ce dernier caractère dans la valeur s'il suit un antislash `\`. L'ensemble des deux caractères `\"` sera traité de façon spécifique par certaines fonctions telle que `cat` pour l'affichage à l'écran, ou `write.table` pour écrire sur le disque (p. 13, l'option `qmethod` de cette fonction).

```
> cit <- "She said: \"Double quotes can be included in R's strings.\""
> cit
[1] "She said: \"Double quotes can be included in R's strings.\""
> cat(cit)
She said: "Double quotes can be included in R's strings."
```

Le tableau suivant donne un aperçu des objets représentant des données.

| objet | modes | plusieurs modes possibles dans le même objet ? |
|---|---|---|
| vecteur | numérique, caractère, complexe *ou* logique | Non |
| facteur | numérique *ou* caractère | Non |
| array | numérique, caractère, complexe *ou* logique | Non |
| matrice | numérique, caractère, complexe *ou* logique | Non |
| data.frame | numérique, caractère, complexe *ou* logique | Oui |
| ts | numérique, caractère, complexe *ou* logique | Oui |
| liste | numérique, caractère, complexe, logique, fonction, expression, ... | Oui |

Un vecteur est une variable dans le sens généralement admis. Un facteur est une variable catégorique. Un *array* est un tableau à $k$ dimensions, une matrice étant un cas particulier d'array avec $k = 2$. À noter que les éléments d'un array ou d'une matrice sont tous du même mode. Un *data.frame* est un tableau de données composé de un ou plusieurs vecteurs et/ou facteurs ayant tous la même longueur mais pouvant être de modes différents. Un *ts* est un jeu de données de type séries temporelles (*time series*) et comporte donc des attributs supplémentaires comme la fréquence et les dates. Enfin, une *liste* peut contenir n'importe quel type d'objet, y compris des listes !

Pour un vecteur, le mode et la longueur suffisent pour décrire les données. Pour les autres objets, d'autres informations sont nécessaires et celles-ci sont données par les attributs dits *non-intrinsèques*. Parmi ces attributs, citons *dim* qui correspond au nombre de dimensions d'un objet. Par exemple, une matrice composée de 2 lignes et 2 colonnes aura pour dim le couple de valeurs [2, 2] ; par contre sa longueur sera de 4.

### 3.2 Lire des données dans un fichier

Pour les lectures et écritures dans les fichiers, R utilise le répertoire de travail. Pour connaître ce répertoire on peut utiliser la commande `getwd()` (*get working directory*), et on peut le modifier avec, par exemple, `setwd("C:/data")` ou `setwd("/home/paradis/R")`. Il est nécessaire de préciser le chemin d'accès au fichier s'il n'est pas dans le répertoire de travail.[8]

R peut lire des données stockées dans des fichiers texte (ASCII) à l'aide des fonctions suivantes : `read.table` (qui a plusieurs variantes, *cf.* ci-dessous), `scan` et `read.fwf`. R peut également lire des fichiers dans d'autres formats (Excel, SAS, SPSS, ...) et accéder à des bases de données de type SQL, mais les fonctions nécessaires ne sont pas dans le package `base`. Ces fonctionnalités sont très utiles pour une utilisation un peu plus avancée de R, mais on se limitera ici à la lecture de fichiers au format ASCII.

La fonction `read.table` a pour effet de créer un data.frame et est donc le moyen principal pour lire des tableaux de données. Par exemple, si on a un fichier nommé data.dat, la commande :

```
> mydata <- read.table("data.dat")
```

créera un data.frame nommé `mydata`, et chaque variable sera nommée, par défaut, `V1`, `V2`, ... et pourront être accédées individuellement par `mydata$V1`, `mydata$V2`, ..., ou par `mydata["V1"]`, `mydata["V2"]`, ..., ou encore par `mydata[, 1]`, `mydata[, 2]`, ...[9] Il y a plu-

[8] Sous Windows, il est pratique de créer un raccourci de Rgui.exe puis éditer ses propriétés et modifier le répertoire dans le champ "Démarrer en :" sous l'onglet "Raccourci" : ce répertoire sera ensuite le répertoire de travail en démarrant R depuis ce raccourci.

[9] Il y a toutefois une différence : `mydata$V1` et `mydata[, 1]` sont des vecteurs alors que `mydata["V1"]` est un data.frame. On verra plus loin (p. 16) des détails sur la manipulation des objets.

sieurs options dont voici les valeurs par défaut (c'est-à-dire celles utilisées par R si elles sont omises par l'utilisateur) et les détails dans le tableau qui suit :

```
read.table(file, header = FALSE, sep = "", quote = "\"'", dec = ".",
           row.names, col.names, as.is = FALSE, 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 = "#")
```

| | |
|---|---|
| `file` | le nom du fichier (entre `""` ou une variable de mode caractère), éventuellement avec son chemin d'accès (le symbole \est interdit et doit être remplacé par /, même sous Windows), ou un accès distant à un fichier de type URL (http://...) |
| `header` | une valeur logique (`FALSE` ou `TRUE`) indicant si le fichier contient les noms des variables sur la 1ère ligne |
| `sep` | le séparateur de champ dans le fichier, par exemple `sep="\t"` si c'est une tabulation |
| `quote` | les caractères utilisés pour citer les variables de mode caractère |
| `dec` | le caractère utilisé pour les décimales |
| `row.names` | un vecteur contenant les noms des lignes qui peut être un vecteur de mode character, ou le numéro (ou le nom) d'une variable du fichier (par défaut : `1`, `2`, `3`, ...) |
| `col.names` | un vecteur contenant les noms des variables (par défaut : `V1`, `V2`, `V3`, ...) |
| `as.is` | contrôle la conversion des variables caractères en facteur (si `FALSE`) ou les conserve en caractères (`TRUE`) ; `as.is` peut être un vecteur logique ou un vecteur numérique précisant les variables conservées en caractère |
| `na.strings` | indique la valeur des données manquantes (sera converti en `NA`) |
| `colClasses` | un vecteur de caractères donnant les classes à attribuer aux colonnes |
| `nrows` | le nombre maximum de lignes à lire (les valeurs négatives sont ignorées) |
| `skip` | le nombre de lignes à sauter avant de commencer la lecture des données |
| `check.names` | si `TRUE`, vérifie que les noms des variables sont valides pour R |
| `fill` | si `TRUE` et que les lignes n'ont pas tous le même nombre de variables, des "blancs" sont ajoutés |
| `strip.white` | (conditionnel à `sep`) si `TRUE`, efface les espaces (= blancs) avant et après les variables de mode caractère |
| `blank.lines.skip` | si `TRUE`, ignore les lignes "blanches" |
| `comment.char` | un caractère qui définit des commentaires dans le fichier de données, les lignes commençant par ce caractère sont ignorées (pour désactiver cet argument, utiliser `comment.char = ""`) |

Les variantes de `read.table` sont utiles car elles ont des valeurs par défaut différentes :

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, ...)
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, ...)
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, ...)
```

La fonction `scan` est plus flexible que `read.table`. Une différence est qu'il est possible de spécifier le mode des variables, par exemple :

```
> mydata <- scan("data.dat", what = list("", 0, 0))
```

lira dans le fichier data.dat trois variables, la première de mode caractère et les deux suivantes de mode numérique. Une autre distinction importante est que `scan()` peut être utilisée pour créer différents objets, vecteurs, matrices, data.frame, listes, ... Dans l'exemple ci-dessus, `mydata` est

une liste de trois vecteurs. Par défaut, c'est-à-dire si `what` est omis, `scan()` crée un vecteur numérique. Si les données lues ne correspondent pas au(x) mode(s) attendu(s) (par défaut ou spécifiés par `what`), un message d'erreur est retourné. Les options sont les suivantes.

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
     quote = if (sep=="\n") "" else "'\"", dec = ".",
     skip = 0, nlines = 0, na.strings = "NA",
     flush = FALSE, fill = FALSE, strip.white = FALSE, quiet = FALSE,
     blank.lines.skip = TRUE, multi.line = TRUE, comment.char = "#")
```

| | |
|---|---|
| `file` | le nom du fichier (entre `""`), éventuellement avec son chemin d'accès (le symbole \ est interdit et doit être remplacé par /, même sous Windows), ou un accès distant à un fichier de type URL (http://...) ; si `file=""`, les données sont entrées au clavier (l'entrée étant terminée par une ligne blanche) |
| `what` | indique le(s) mode(s) des données lues (numérique par défaut) |
| `nmax` | le nombre de données à lire, ou, si `what` est une liste, le nombre de lignes lues (par défaut, `scan` lit jusqu'à la fin du fichier) |
| `n` | le nombre de données à lire (par défaut, pas de limite) |
| `sep` | le séparateur de champ dans le fichier |
| `quote` | les caractères utilisés pour citer les variables de mode caractère |
| `dec` | le caractère utilisé pour les décimales |
| `skip` | le nombre de lignes à sauter avant de commencer la lecture des données |
| `nlines` | le nombre de lignes à lire |
| `na.string` | indique la valeur des données manquantes (sera converti en `NA`) |
| `flush` | si `TRUE`, `scan` va à la ligne suivante une fois que le nombre de colonnes est atteint (permet d'ajouter des commentaires dans le fichier de données) |
| `fill` | si `TRUE` et que les lignes n'ont pas tous le même nombre de variables, des "blancs" sont ajoutés |
| `strip.white` | (conditionnel à `sep`) si `TRUE`, efface les espaces (= blancs) avant et après les variables de mode character |
| `quiet` | si `FALSE`, `scan` affiche une ligne indiquant quels champs ont été lus |
| `blank.lines.skip` | si `TRUE`, ignore les lignes "blanches" |
| `multi.line` | si `what` est une liste, précise si les variables du même individu sont sur une seule ligne dans le fichier (`FALSE`) |
| `comment.char` | un caractère qui définit des commentaires dans le fichier de données, les lignes commençant par ce caractère sont ignorées |

La fonction `read.fwf` sert à lire dans un fichier où les données sont dans un format à largeur fixée (*fixed width format*) :

```
read.fwf(file, widths, sep="\t", as.is = FALSE,
         skip = 0, row.names, col.names, n = -1)
```

Les options sont les mêmes que pour `read.table()` sauf `widths` qui spécifie la largeur des champs. Par exemple, si on a un fichier nommé data.txt dont le contenu est indiqué ci-contre, on pourra lire les données avec la commande suivante :

```
A1.501.2
A1.551.3
B1.601.4
B1.651.5
C1.701.6
C1.751.7
```

```
> mydata <- read.fwf("data.txt", widths=c(1, 4, 3))
> mydata
  V1   V2  V3
1  A 1.50 1.2
2  A 1.55 1.3
3  B 1.60 1.4
4  B 1.65 1.5
```

```
5  C 1.70 1.6
6  C 1.75 1.7
```

## 3.3 Enregistrer les données

La fonction `write.table` écrit dans un fichier un objet, typiquement un data.frame mais cela peut très bien être un autre type d'objet (vecteur, matrice, ...). Les arguments et options sont :

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

| | |
|---|---|
| `x` | le nom de l'objet à écrire |
| `file` | le nom du fichier (par défaut l'objet est affiché à l'écran) |
| `append` | si `TRUE` ajoute les données sans effacer celles éventuellement existantes dans le fichier |
| `quote` | une variable logique ou un vecteur numérique : si `TRUE` les variables de mode caractère et les facteurs sont écrits entre `""`, sinon le vecteur indique les numéros des variables à écrire entre `""` (dans les deux cas les noms des variables sont écrits entre `""` mais pas si `quote = FALSE`) |
| `sep` | le séparateur de champ dans le fichier |
| `eol` | le caractère imprimé à la fin de chaque ligne (`"\n"` correspond à un retour-charriot) |
| `na` | indique le caractère utilisé pour les données manquantes |
| `dec` | le caractère utilisé pour les décimales |
| `row.names` | une variable logique indiquant si les noms des lignes doivent être écrits dans le fichier |
| `col.names` | idem pour les noms des colonnes |
| `qmethod` | spécifie, si `quote=TRUE`, comment sont traitées les guillemets doubles `"` incluses dans les variables de mode caractère : si `"escape"` (ou `"e"`, le défaut) chaque `"` est remplacée par `\"`, si `"d"` chaque `"` est remplacée par `""` |

Pour écrire de façon plus simple un objet dans un fichier, on peut utiliser la commande `write(x, file="data.txt")` où `x` est le nom de l'objet (qui peut être un vecteur, une matrice ou un array). Il y a deux options : `nc` (ou `ncol`) qui définit le nombre de colonnes dans le fichier (par défaut `nc=1` si `x` est de mode caractère, `nc=5` pour les autres modes), et `append` (un logique) pour ajouter les données sans effacer celles éventuellement déjà existantes dans le fichier (`TRUE`) ou les effacer si le fichier existe déjà (`FALSE`, le défaut).

Pour enregistrer des objets, cette fois de n'importe quel type, on utilisera la commande `save (x, y, z, file="xyz.RData")`. Pour faciliter l'échange de fichiers entre machines et systèmes d'exploitation, on peut utiliser l'option `ascii=TRUE`. Les données (qui sont alors nommées *workspace* dans le jargon de R) peuvent ultérieurement être chargées en mémoire avec `load("xyz.RData")`. La fonction `save.image` est un raccourci pour `save(list=ls (all=TRUE), file=".RData")`.

## 3.4 Générer des données

### *3.4.1 Séquences régulières*

Une séquence régulière de nombres entiers, par exemple de 1 à 30, peut être générée par :

```
> x <- 1:30
```

On a ainsi un vecteur `x` avec 30 éléments. Cet opérateur '`:`' est prioritaire sur les opérations arithmétiques au sein d'une expression :

```
> 1:10-1
 [1] 0 1 2 3 4 5 6 7 8 9
> 1:(10-1)
[1] 1 2 3 4 5 6 7 8 9
```

La fonction `seq` peut générer des séquences de nombres réels de la manière suivante :

```
> seq(1, 5, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

où le premier nombre indique le début de la séquence, le second la fin, et le troisième l'incrément utilisé dans la progression de la séquence. On peut aussi utiliser :

```
> seq(length=9, from=1, to=5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

On peut aussi taper directement les valeurs désirées en utilisant la fonction `c` :

```
> c(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Il est aussi possible si l'on veut taper des données au clavier d'utiliser la fonction `scan` avec tout simplement les options par défaut :

```
> z <- scan()
1: 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
10:
Read 9 items
> z
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

La fonction `rep` crée un vecteur qui aura tous ses éléments identiques :

```
> rep(1, 30)
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

La fonction `sequence` va créer une suite de séquences de nombres entiers qui chacune se termine par les nombres donnés comme arguments à cette fonction :

```
> sequence(4:5)
[1] 1 2 3 4 1 2 3 4 5
> sequence(c(10,5))
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5
```

La fonction `gl` (*generate levels*) est très utile car elle génère des séries régulières dans un facteur. Cette fonction s'utilise ainsi `gl(k, n)` où `k` est le nombre de niveaux (ou classes) du facteur, et `n` est le nombre de réplications pour chaque niveau. Deux options peuvent être utilisées : `length` pour spécifier le nombre de données produites, et `labels` pour indiquer les noms des niveaux du facteur. Exemples :

```
> gl(3, 5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> gl(3, 5, length=30)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> gl(2, 6, label=c("Male", "Female"))
 [1] Male   Male   Male   Male   Male   Male
 [7] Female Female Female Female Female Female
Levels:  Male Female
> gl(2, 10)
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
Levels:  1 2
```

```
> gl(2, 1, length=20)
 [1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels:  1 2
> gl(2, 2, length=20)
 [1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2
Levels:  1 2
```

Enfin, `expand.grid()` sert à créer un data.frame avec toutes les combinaisons des vecteurs ou facteurs donnés comme arguments :

```
> expand.grid(h=c(60,80), w=c(100, 300), sex=c("Male", "Female"))
   h   w    sex
1 60 100   Male
2 80 100   Male
3 60 300   Male
4 80 300   Male
5 60 100 Female
6 80 100 Female
7 60 300 Female
8 80 300 Female
```

#### *3.4.2 Séquences aléatoires*

Il est utile en statistique de pouvoir générer des données aléatoires, et R peut le faire pour un grand nombre de fonctions de densité de probabilité. Ces fonctions sont de la forme `r`*`func`*`(n, p1, p2, ...)`, où *`func`* indique la loi de probabilité, `n` le nombre de données générées et `p1`, `p2`, ... sont les valeurs des paramètres de la loi. Le tableau suivant donne les détails pour chaque loi, et les éventuelles valeurs par défaut (si aucune valeur par défaut n'est indiquée, c'est que le paramètre doit être spécifié).

| **loi** | **fonction** |
|---|---|
| Gauss (normale) | `rnorm(n, mean=0, sd=1)` |
| exponentielle | `rexp(n, rate=1)` |
| gamma | `rgamma(n, shape, scale=1)` |
| Poisson | `rpois(n, lambda)` |
| Weibull | `rweibull(n, shape, scale=1)` |
| Cauchy | `rcauchy(n, location=0, scale=1)` |
| beta | `rbeta(n, shape1, shape2)` |
| 'Student' ($t$) | `rt(n, df)` |
| Fisher–Snedecor ($F$) | `rf(n, df1, df2)` |
| Pearson ($\chi^2$) | `rchisq(n, df)` |
| binomiale | `rbinom(n, size, prob)` |
| géométrique | `rgeom(n, prob)` |
| hypergéométrique | `rhyper(nn, m, n, k)` |
| logistique | `rlogis(n, location=0, scale=1)` |
| lognormale | `rlnorm(n, meanlog=0, sdlog=1)` |
| binomiale négative | `rnbinom(n, size, prob)` |
| uniforme | `runif(n, min=0, max=1)` |
| statistiques de Wilcoxon | `rwilcox(nn, m, n)`, `rsignrank(nn, n)` |

Toutes ces fonctions peuvent être utilisées en remplaçant la lettre `r` par `d`, `p` ou `q` pour obtenir, dans l'ordre, la densité de probabilité (`dfunc(x, ...)`), la densité de probabilité cumulée (`pfunc(x, ...)`), et la valeur de quantile (`qfunc(p, ...)`, avec $0 < p < 1$).

## 3.5 Manipuler les objets

### *3.5.1 Création d'objets*

On a vu différentes façons de créer des objets en utilisant l'opérateur assigner ; le mode et le type de l'objet ainsi créé sont généralement déterminés de façon implicite. Il est possible de créer un objet en précisant de façon explicite son mode, sa longueur, son type, etc. Cette approche est intéressante dans l'idée de manipuler les objets. On peut, par exemple, créer un vecteur 'vide' puis modifier successivement ses éléments, ce qui est beaucoup plus efficace que de rassembler ces éléments avec `c()`. On utilisera alors l'indexation comme on le verra plus loin (p. 22).

Il peut être aussi extrêment pratique de créer des objets à partir d'autres objets. Par exemple, si l'on veut ajuster une série de modèles, il sera commode de mettre les formules correspondantes dans une liste puis d'extraire successivement chaque élément de celle-ci qui sera ensuite inséré dans la fonction `lm`.

À ce point de notre apprentissage de R, l'intérêt d'aborder les fonctionnalités qui suivent n'est pas seulement pratique mais aussi didactique. La construction explicite d'objets permet de mieux comprendre leur structure et d'approfondir certaines notions vues précédemment.

**Vecteur.** La fonction `vector`, qui a deux arguments `mode` et `length`, va servir à créer un vecteur dont la valeur des éléments sera fonction du mode spécifié : 0 si numérique, `FALSE` si logique, ou `""` si caractère. Les fonctions suivantes ont exactement le même effet et ont pour seul argument la longueur du vecteur créé : `numeric()`, `logical()`, et `character()`.

**Facteur.** Un facteur inclue non seulement les valeurs de la variable catégorique correspondante mais aussi les différents niveaux possibles de cette variable (même ceux qui ne sont pas représentés dans les données). La fonction `factor` crée un facteur avec les options suivantes :

```
factor(x, levels = sort(unique(x), na.last = TRUE),
       labels = levels, exclude = NA, ordered = is.ordered(x))
```

`levels` spécifie quels sont les niveaux possibles du facteur (par défaut les valeurs uniques du vecteur `x`), `labels` définit les noms des niveaux, `exclude` les valeurs de `x` à ne pas inclure dans les niveaux, et `ordered` est un argument logique spécifiant si les niveaux du facteur sont ordonnés. Rappelons que `x` est de mode numérique ou caractère. En guise d'exemples :

```
> factor(1:3)
[1] 1 2 3
Levels:  1 2 3
> factor(1:3, levels=1:5)
[1] 1 2 3
Levels:  1 2 3 4 5
> factor(1:3, labels=c("A", "B", "C"))
[1] A B C
Levels:  A B C
> factor(1:5, exclude=4)
[1] 1  2  3  NA 5
Levels:  1 2 3 5
```

La fonction `levels` sert à extraire les niveaux possibles d'un facteur :

```
> ff <- factor(c(2, 4), levels=2:5)
> ff
[1] 2 4
Levels:  2 3 4 5
> levels(ff)
[1] "2" "3" "4" "5"
```

**Matrice.** Une matrice est en fait un vecteur qui possède un argument supplémentaire (dim) qui est lui-même un vecteur numérique de longueur 2 et qui définit les nombres de lignes et de colonnes de la matrice. Une matrice peut être créée avec la fonction `matrix` :

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)
```

L'option `byrow` indique si les valeurs données par `data` doivent remplir successivement les colonnes (le défaut) ou les lignes (si `TRUE`). L'option `dimnames` permet de donner des noms aux lignes et colonnes.

```
> matrix(data=5, nr=2, nc=2)
     [,1] [,2]
[1,]    5    5
[2,]    5    5
> matrix(1:6, 2, 3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:6, 2, 3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Une autre façon de créer une matrice est de donner les valeurs voulues à l'attribut dim d'un vecteur (attribut qui est initialement `NULL`) :

```
> x <- 1:15
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> dim(x)
NULL
> dim(x) <- c(5, 3)
> x
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

**Data frame.** On a vu qu'un data.frame est créé de façon implicite par la fonction `read.table` ; on peut également créer un data.frame avec la fonction `data.frame`. Les vecteurs inclus dans le data.frame doivent être de même longueur, ou si un de ces éléments est plus court il est alors "recyclé" un nombre entier de fois :

```
> x <- 1:4; n <- 10; M <- c(10, 35); y <- 2:4
> data.frame(x, n)
  x  n
1 1 10
2 2 10
3 3 10
4 4 10
> data.frame(x, M)
  x  M
1 1 10
2 2 35
3 3 10
4 4 35
> data.frame(x, y)
Error in data.frame(x, y) :
    arguments imply differing number of rows: 4, 3
```

Si un facteur est inclus dans le data.frame, il doit être de même longueur que le(s) vecteur(s). Il est possible de changer les noms des colonnes avec `data.frame(A1=x, A2=n)`. On peut aussi donner des noms aux lignes avec l'option `row.names` qui doit, bien sûr, être un vecteur de mode caractère et de longueur égale au nombre de lignes du data.frame. Enfin, notons que les data.frames ont un attribut dim de la même façon que les matrices.

**Liste.** Une liste est créée de la même façon qu'un data.frame avec la fonction `list`. Il n'y a aucune contrainte sur les objets qui y sont inclus. À la différence de `data.frame()`, les noms des objets ne sont pas repris par défaut ; en reprenant les vecteurs `x` et `y` de l'exemple précédant :

```
> L1 <- list(x, y); L2 <- list(A=x, B=y)
> L1
[[1]]
[1] 1 2 3 4

[[2]]
[1] 2 3 4

> L2
$A
[1] 1 2 3 4

$B
[1] 2 3 4

> names(L1)
NULL
> names(L2)
[1] "A" "B"
```

**Série temporelle.** La fonction `ts` va créer un objet de classe `"ts"` à partir d'un vecteur (série temporelle simple) ou d'une matrice (série temporelle multiple), et des options qui caractérisent la série. Les options, avec les valeurs par défaut, sont :

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class, names)
```

| | |
|---|---|
| `data` | un vecteur ou une matrice |
| `start` | le temps de la 1ère observation, soit un nombre, ou soit un vecteur de deux entiers (*cf.* ex. ci-dessous) |
| `end` | le temps de la dernière observation spécifié de la même façon que `start` |
| `frequency` | nombre d'observations par unité de temps |
| `deltat` | la fraction de la période d'échantillonnage entre observations successives (ex. `1/12` pour des données mensuelles) ; seulement un de `frequency` ou `deltat` doit être précisé |
| `ts.eps` | tolérance pour la comparaison de séries. Les fréquences sont considérées égales si leur différence est inférieure à `ts.eps` |
| `class` | classe à donner à l'objet ; le défaut est `"ts"` pour une série simple, et `c("mts", "ts")` pour une série multiple |
| `names` | un vecteur de mode caractère avec les noms des séries individuelles dans le cas d'une série multiple ; par défaut les noms des colonnes de `data`, ou `Series 1`, `Series 2`, ... |

Quelques exemples de création de séries temporelles avec `ts()` :

```
> ts(1:10, start = 1959)
Time Series:
Start = 1959
End = 1968
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
> ts(1:47, frequency = 12, start = c(1959, 2))
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1959       1   2   3   4   5   6   7   8   9  10  11
1960  12  13  14  15  16  17  18  19  20  21  22  23
1961  24  25  26  27  28  29  30  31  32  33  34  35
1962  36  37  38  39  40  41  42  43  44  45  46  47
> ts(1:10, frequency = 4, start = c(1959, 2))
     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
> ts(matrix(rpois(36, 5), 12, 3), start=c(1961, 1), frequency=12)
         Series 1 Series 2 Series 3
Jan 1961        8        5        4
Feb 1961        6        6        9
Mar 1961        2        3        3
Apr 1961        8        5        4
May 1961        4        9        3
Jun 1961        4        6       13
Jul 1961        4        2        6
Aug 1961       11        6        4
Sep 1961        6        5        7
Oct 1961        6        5        7
Nov 1961        5        5        7
```

```
Dec 1961        8        5        2
```

**Expression.** Les objets de mode expression ont un rôle fondamental dans R. Une expression est une suite de caractères qui ont un sens pour R. Toutes les commandes valides sont des expressions. Lorsque la commande est tapée directement au clavier, elle est alors *évaluée* par R qui l'exécute si elle est valide. Dans bien des circonstances, il est utile de construire une expression sans l'évaluer : c'est le rôle de la fonction `expression`. On pourra, bien sûr, évaluer l'expression ultérieurement avec `eval()`.

```
> x <- 3; y <- 2.5; z <- 1
> exp1 <- expression(x / (y + exp(z)))
> exp1
expression(x/(y + exp(z)))
> eval(exp1)
[1] 0.5749019
```

Les expressions servent aussi, entre autres, à inclure des équations sur les graphiques (p. 32). Une expression peut être créée à partir d'une variable de mode caractère. Certaines fonctions utilisent des expressions en tant qu'argument, par exemple `D()` qui calcule des dérivées partielles :

```
> D(exp1, "x")
1/(y + exp(z))
> D(exp1, "y")
-x/(y + exp(z))^2
> D(exp1, "z")
-x * exp(z)/(y + exp(z))^2
```

#### *3.5.2 Conversion d'objets*

Le lecteur aura sûrement réalisé que les différences entre certains objets sont parfois minces ; il est donc logique de pouvoir convertir un objet en un autre en changeant certains de ces attributs. Une telle conversion sera effectuée avec une fonction du genre `as.`*`something`*. R (version 1.5.1) comporte, dans le package `base`, 77 de ces fonctions, aussi nous ne rentrerons pas dans les détails ici.

Le résultat d'une conversion dépend bien sûr des attributs de l'objet converti. En général, la conversion suit des règles intuitives. Pour les conversions de modes, le tableau suivant résume la situation.

| Conversion en | Fonction | Règles |
|---|---|---|
| numérique | `as.numeric` | `FALSE` → 0 |
| | | `TRUE` → 1 |
| | | `"1"`, `"2"`, ... → 1, 2, ... |
| | | `"A"`, ... → `NA` |
| logique | `as.logical` | 0 → `FALSE` |
| | | autres nombres → `TRUE` |
| | | `"FALSE"`, `"F"` → `FALSE` |
| | | `"TRUE"`, `"T"` → `TRUE` |
| | | autres caractères → `NA` |
| caractère | `as.character` | 1, 2, ... → `"1"`, `"2"`, ... |
| | | `FALSE` → `"FALSE"` |
| | | `TRUE` → `"TRUE"` |

Il existe des fonctions pour convertir les types d'objets (`as.matrix`, `as.data.frame`, `as.ts`, `as.expression`, ...). Ces fonctions vont agir sur des attributs autres que le mode pour la conversion. Là encore les résultats sont généralement intuitifs. Une situation fréquemment rencontrée est la conversion de facteur en vecteur numérique. Dans ce cas, R convertit avec le codage numérique des niveaux du facteur :

```
> fac <- factor(c(1, 10))
> fac
[1] 1 10
Levels:  1 10
> as.numeric(fac)
[1] 1 2
```

Pour convertir un facteur en conservant les niveaux tels qu'ils sont spécifiés, on convertira d'abord en caractère puis en numérique.

```
> as.numeric(as.character(fac))
[1] 1 10
```

Cette procédure est très utile si, dans un fichier, une variable numérique contient (pour une raison ou une autre) également des valeurs non-numériques. On a vu que `read.table()` dans ce genre de situation va, par défaut, lire cette colonne comme un facteur.

### *3.5.3 Les opérateurs*

On a vu précédemment qu'il y a trois types d'opérateurs dans R[10]. En voici la liste.

| **Opérateurs** | | | | | |
|---|---|---|---|---|---|
| Arithmétique | | Comparaison | | Logique | |
| `+` | addition | `<` | inférieur à | `! x` | NON logique |
| `-` | soustraction | `>` | supérieur à | `x & y` | ET logique |
| `*` | multiplication | `<=` | inférieur ou égal à | `x && y` | idem |
| `/` | division | `>=` | supérieur ou égal à | `x \| y` | OU logique |
| `^` | puissance | `==` | égal | `x \|\| y` | idem |
| `%%` | modulo | `!=` | différent | `xor(x, y)` | OU exclusif |
| `%/%` | division entière | | | | |

Les opérateurs arithmétiques ou de comparaison agissent sur deux éléments (`x + y`, `a < b`). Les opérateurs arithmétiques agissent sur les variables de mode numérique ou complexe, mais aussi sur celles de mode logique ; dans ce dernier cas, les valeurs logiques sont converties en valeurs numériques. Les opérateurs de comparaison peuvent s'appliquer à n'importe quel mode : ils retournent une ou plusieurs valeurs logiques.

Les opérateurs logiques s'appliquent à un (`!`) ou deux objets de mode logique et retournent une (ou plusieurs) valeurs logiques. Les opérateurs "ET" et "OU" existent sous deux formes : la forme simple opére sur chaque élément des objets et retourne autant de valeurs logiques que de comparaisons effectuées ; la forme double opére sur le premier élément des objets.

On utilisera l'opérateur "ET" pour spécifier une inégalité du type $0 < x < 1$ qui sera codée ainsi : `0 < x & x < 1`. L'expression `0 < x < 1` est valide mais ne donnera pas le résultat escompté : les deux opérateurs de cette expression étant identiques, ils seront exécutés successivement de la gauche vers la droite. L'opération `0 < x` sera d'abord réalisée retournant une valeur

[10] Les caractères suivants sont en fait aussi des opérateurs pour R : `$`, `[`, `[[`, `:`, `?`, `<-`.

logique qui sera ensuite comparée à 1 (`TRUE` ou `FALSE < 1`) : dans ce cas la valeur logique sera convertie implicitement en numérique (`1` ou `0 < 1`).

Les opérateurs de comparaison opèrent sur *chaque* élément des deux objets qui sont comparés (en recyclant éventuellement les valeurs si l'un est plus court), et retournent donc un objet de même taille. Pour effectuer une comparaison "globale" de deux objets, il faut utiliser la fonction `identical`:

```
> x <- 1:3; y <- 1:3
> x == y
[1] TRUE TRUE TRUE
> identical(x, y)
[1] TRUE
```

#### *3.5.4 Accéder aux valeurs d'un objet : le système d'indexation*

L'indexation est un moyen efficace et flexible d'accéder de façon sélective aux éléments d'un objet ; elle peut être *numérique* ou *logique*. Pour accéder à, par exemple, la 3ème valeur d'un vecteur `x`, on tape `x[3]`. Si `x` est une matrice ou un data.frame, on accédera à la valeur de la i$^{ème}$ ligne et j$^{ème}$ colonne par `x[i, j]`. Pour changer toutes les valeurs de la 3$^{ème}$ colonne, on peut taper :

```
> x[, 3] <- 10.2
```

Ce système d'indexation se généralise facilement pour les array, on aura alors autant d'indices que l'array a de dimensions (par exemple pour une array à trois dimensions : `x[i, j, k]`, `x[, , 3]`, …). Il faut retenir que l'indexation se fait à l'aide de crochets, les parenthèses étant réservées pour les arguments d'une fonction :

```
> x(1)
Error: couldn't find function "x"
```

L'indexation peut aussi être utilisée pour supprimer une (des) ligne(s) ou colonne(s). Par exemple, `x[-1, ]` supprimera la 1ère ligne, ou `x[-c(1, 15), ]` fera de même avec les 1ère et 15ème lignes.

Pour les vecteurs, matrices et array il est possible d'accéder aux valeurs de ces éléments à l'aide d'une expression de comparaison en guise d'indice :

```
> x <- 1:10
> x[x >= 5] <- 20
> x
 [1]  1  2  3  4 20 20 20 20 20 20
> x[x == 1] <- 25
> x
 [1] 25  2  3  4 20 20 20 20 20 20
```

Une utilisation pratique de cette indexation logique est, par exemple, la possibilité de sélectionner les éléments pairs d'une variable entière :

```
> x <- rpois(40, lambda=5)
> x
 [1]  5  9  4  7  7  6  4  5 11  3  5  7  1  5  3  9  2  2  5  2
[21]  4  6  6  5  4  5  3  4  3  3  3  7  7  3  8  1  4  2  1  4
> x[x %% 2 == 0]
 [1] 4 6 4 2 2 2 4 6 6 4 4 8 4 2 4
```

Ce système d'indexation utilise donc des valeurs logiques retournées dans ce cas par les opérateurs de comparaison. Ces valeurs logiques peuvent être calculées au préalable, elles seront éventuellement recyclées :

```
> x <- 1:40
> s <- c(FALSE, TRUE)
> x[s]
 [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

L'indexation logique peut également être utilisée avec des data.frames, mais avec la difficulté que les différentes colonnes du data.frame peuvent être de modes différents.

Pour les listes, l'accès aux différents éléments (qui peuvent être n'importe quel objet) se fait avec des crochets doubles, par exemple `my.list[[3]]` pour accéder au 3ème objet de la liste nommée `my.list`. Le résultat pourra ensuite être indexé de la même façon que l'on a vu précédemment pour les vecteurs, matrices, etc. Si ce 3ème objet en question est un vecteur, ses valeurs pourront être modifiées avec `my.list[[3]][i]`, s'il s'agit d'un array à trois dimensions avec `my.list[[3]][i, j, k]`, etc.

#### *3.5.5 Accéder aux valeurs d'un objet avec les noms*

On a vu à plusieurs reprises le concept de *nom* apparaître. Les noms sont des attributs dont il existe plusieurs sortes (*names*, *colnames*, *rownames*, *dimnames*). On va se limiter ici à des notions très simples sur ces noms, en particulier pour accéder aux éléments d'un objet.

Si les éléments d'un objet ont des noms, ils peuvent être extraits en les utilisant en guise d'indices. De cette façon les attributs de l'objet d'origine sont conservés. Par exemple, si un data.frame `DF` comporte les variables `x`, `y`, et `z`, la commande `DF["x"]` donnera un data.frame avec juste `x` ; `DF[c("x", "y")]` donnera un data.frame avec les deux variables correspondantes. Ce système marche aussi avec une liste si ses éléments ont des noms.

Comme on le constate, l'index ainsi utilisé est un vecteur de mode caractère. Comme pour les vecteurs logiques ou numériques vus précédemment, ce vecteur peut être établi au préalable et ensuite inséré pour l'extraction.

Pour extraire un vecteur ou un facteur d'un data.frame on utilisera le symbole `$` (par ex. `DF$x`). Cette procédure marche également avec les listes.

#### *3.5.6 L'éditeur de données*

Il est possible d'utiliser un éditeur graphique de style tableur pour éditer un objet contenant des données. Par exemple, si on a une matrice `X`, la commande `data.entry(X)` ouvrira l'éditeur graphique et l'on pourra modifier les valeurs en cliquant sur les cases correspondantes ou encore ajouter des colonnes ou des lignes.

La fonction `data.entry` modifie directement l'objet passé en argument sans avoir à assigner son résultat. Par contre la fonction `de` retourne une liste composée des objets passés en arguments et éventuellement modifiés. Ce résultat est affiché à l'écran par défaut mais, comme pour la plupart des fonctions, peut être assigné dans un objet.

Les détails de l'utilisation de cet éditeur de données dépendent du système d'exploitation.

#### *3.5.7 Calcul arithmétique et fonctions simples*

Il existe de nombreuses fonctions dans R pour manipuler les données. La plus simple, on l'a vue plus haut, est `c` qui concatène les objets énumérés entre parenthèses. Par exemple :

```
> c(1:5, seq(10, 11, 0.2))
 [1]  1.0  2.0  3.0  4.0  5.0 10.0 10.2 10.4 10.6 10.8 11.0
```

Les vecteurs peuvent être manipulés selon des expressions arithmétiques classiques :

```
> x <- 1:4
> y <- rep(1, 4)
> z <- x + y
> z
[1] 2 3 4 5
```

Des vecteurs de longueurs différentes peuvent être additionnés, dans ce cas le vecteur le plus court est recyclé. Exemples :

```
> x <- 1:4
> y <- 1:2
> z <- x + y
> z
[1] 2 4 4 6
> x <- 1:3
> y <- 1:2
> z <- x + y
Warning message:
longer object length
 is not a multiple of shorter object length in: x + y
> z
[1] 2 4 4
```

On notera que R a retourné un message d'avertissement et non pas un message d'erreur, l'opération a donc été effectuée. Si l'on veut ajouter (ou multiplier) la même valeur à tous les éléments d'un vecteur :

```
> x <- 1:4
> a <- 10
> z <- a * x
> z
[1] 10 20 30 40
```

Les fonctions disponibles avec R sont trop nombeuses pour être énumérées ici. On trouve toutes les fonctions mathématiques de base (`log`, `exp`, `log10`, `log2`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `abs`, `sqrt`, …), des fonctions spéciales (`gamma`, `digamma`, `beta`, `besselI`, …), ainsi que diverses fonctions utiles en statistiques. Quelques-unes sont détaillées dans le tableau qui suit.

| | |
|---|---|
| `sum(x)` | somme des éléments de `x` |
| `prod(x)` | produit des éléments de `x` |
| `max(x)` | maximum des éléments de `x` |
| `min(x)` | minimum des éléments de `x` |
| `which.max(x)` | retourne l'indice du maximum des éléments de `x` |
| `which.min(x)` | retourne l'indice du minimum des éléments de `x` |
| `range(x)` | idem que `c(min(x), max(x))` |
| `length(x)` | nombre d'éléments dans `x` |
| `mean(x)` | moyenne des éléments de `x` |
| `median(x)` | médianne des éléments de `x` |
| `var(x)` ou `cov(x)` | variance des éléments de `x` (calculée sur $n-1$); si `x` est une matrice ou un data.frame, la matrice de variance-covariance est alors calculée |
| `cor(x)` | matrice de corrélation si `x` est une matrice ou un data.frame (1 si `x` est un vecteur) |
| `var(x, y)` ou `cov(x, y)` | covariance entre `x` et `y`, ou entre les colonnes de `x` et de `y` si ce sont des matrices ou des data.frames |
| `cor(x, y)` | corrélation linéaire entre `x` et `y`, ou matrice de corrélations si ce sont des matrices ou des data.frames |

Ces fonctions retournent une valeur simple (donc un vecteur de longueur 1), sauf `range()` qui retourne un vecteur de longueur 2, et `var()`, `cov()` et `cor()` qui peuvent retourner une matrice. Les fonctions suivantes retournent des résultats plus complexes.

| | |
|---|---|
| `round(x, n)` | arrondit les éléments de `x` à `n` chiffres après la virgule |
| `rev(x)` | inverse l'ordre des éléments de `x` |
| `sort(x)` | trie les éléments de `x` dans l'ordre ascendant ; pour trier dans l'ordre descendant : `rev(sort(x))` |
| `rank(x)` | rangs des éléments de `x` |
| `log(x, base)` | calcule le logarithme à base `"base"` de `x` |
| `scale(x)` | si `x` est une matrice, centre et réduit les données ; pour centrer uniquement ajouter l'option `center=FALSE`, pour réduire uniquement `scale=FALSE` (par défaut `center=TRUE, scale=TRUE`) |
| `pmin(x,y,...)` | un vecteur dont le $i^{ème}$ élément est le minimum entre `x[i]`, `y[i]`, … |
| `pmax(x,y,...)` | idem pour le maximum |
| `cumsum(x)` | un vecteur dont le $i^{ème}$ élément est la somme de `x[1]` à `x[i]` |
| `cumprod(x)` | idem pour le produit |
| `cummin(x)` | idem pour le minimum |
| `cummax(x)` | idem pour le maximum |
| `match(x, y)` | retourne un vecteur de même longueur que `x` contenant les éléments de `x` qui sont dans `y` (`NA` sinon) |
| `which(x == a)` | retourne un vecteur des indices de `x` pour lesquels l'opération de comparaison est vraie (`TRUE`), dans cet exemple les valeurs de `i` telles que `x[i] == a` (l'argument de cette fonction doit être une variable de mode logique) |
| `choose(n, k)` | calcule les combinaisons de $k$ événements parmi $n$ répétitions = $n!/[(n-k)!k!]$ |
| `na.omit(x)` | supprime les observations avec données manquantes (`NA`) (supprime la ligne correspondante si `x` est une matrice ou un data.frame) |
| `na.fail(x)` | retourne un message d'erreur si `x` contient au moins un `NA` |
| `unique(x)` | si `x` est un vecteur ou un data.frame, retourne un objet similaire mais avec les éléments dupliqués supprimés |
| `table(x)` | retourne un tableau des effectifs des différentes valeurs de `x` (typiquement pour des entiers ou des facteurs) |
| `subset(x, ...)` | retourne une sélection de `x` en fonction de critères (`...`, typiquement des comparaisons : `x$V1 < 10`) ; si `x` est un data.frame, l'option `select` permet de préciser les variables à sélectionner (ou à éliminer à l'aide du signe `-`) |
| `sample(x, size)` | ré-échantillonne aléatoirement et sans remise `size` éléments dans le vecteur `x`, pour ré-échantillonner avec remise on ajoute l'option `replace = TRUE` |

### *3.5.8 Calcul matriciel*

R offre des facilités pour le calcul et la manipulation de matrices. Les fonctions `rbind()` et `cbind()` juxtaposent des matrices en conservant les lignes ou les colonnes, respectivement :

```
> m1 <- matrix(1, nr = 2, nc = 2)
> m2 <- matrix(2, nr = 2, nc = 2)
> rbind(m1, m2)
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    2    2
[4,]    2    2
> cbind(m1, m2)
     [,1] [,2] [,3] [,4]
[1,]    1    1    2    2
[2,]    1    1    2    2
```

L'opérateur pour le produit de deux matrices est '%*%'. Par exemple, en reprenant les deux matrices `m1` et `m2` ci-dessus :

```
> rbind(m1, m2) %*% cbind(m1, m2)
     [,1] [,2] [,3] [,4]
[1,]    2    2    4    4
[2,]    2    2    4    4
[3,]    4    4    8    8
[4,]    4    4    8    8
> cbind(m1, m2) %*% rbind(m1, m2)
     [,1] [,2]
[1,]   10   10
[2,]   10   10
```

La transposition d'une matrice se fait avec la fonction `t` ; cette fonction marche aussi avec un data.frame.

La fonction `diag` sert à extraire, modifier la diagonale d'une matrice, ou encore à construire une matrice diagonale.

```
> diag(m1)
[1] 1 1
> diag(rbind(m1, m2) %*% cbind(m1, m2))
[1] 2 2 8 8
> diag(m1) <- 10
> m1
     [,1] [,2]
[1,]   10    1
[2,]    1   10
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
> v <- c(10, 20, 30)
> diag(v)
     [,1] [,2] [,3]
[1,]   10    0    0
[2,]    0   20    0
[3,]    0    0   30
> diag(2.1, nr = 3, nc = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.1  0.0  0.0    0    0
[2,]  0.0  2.1  0.0    0    0
[3,]  0.0  0.0  2.1    0    0
```

R a également des fonctions spéciales pour le calcul matriciel. Citons `solve()` pour l'inversion d'une matrice, `qr()` pour la décomposition, `eigen()` pour le calcul des valeurs et vecteurs propres, et `svd()` pour la décomposition en valeurs singulières.

## 4 Les graphiques avec R

R offre une variété de graphiques remarquable. Pour avoir une petite idée des possibilités offertes, il suffit de taper la commande `demo(graphics)`. Il n'est pas possible ici de détailler

toutes les possibilités ainsi offertes, en particulier chaque fonction graphique a beaucoup d'options qui rendent la production de graphiques extrêment flexible et l'utilisation d'un logiciel de dessin presqu'inutile.

Le fonctionnement des fonctions graphiques dévie substantiellement du schéma dressé au début de ce document. Notamment, le résultat d'une fonction graphique ne peut pas être assigné à un objet[11] mais est envoyé à un *dispositif graphique* (*graphical device*). Un dispositif graphique est matérialisé par une fenêtre graphique ou un fichier.

Il existe deux sortes de fonctions graphiques : *principales* qui créent un nouveau graphe, et *secondaires* qui ajoutent des éléments à un graphe déjà existant. Les graphes sont produits en fonction de *paramètres graphiques* qui sont définis par défaut et peuvent être modifiés avec la fonction `par`.

Nous allons dans un premier temps voir comment gérer les graphiques, ensuite nous détaillerons les fonctions et paramètres graphiques. Nous verrons un exemple concret de l'utilisation de ces fonctionnalités pour la production de graphes. Enfin, nous verrons les packages grid et lattice dont le fonctionnement est différent de celui résumé ci-dessus.

## 4.1 Gestion des graphiques

### *4.1.1 Ouvrir plusieurs dispositifs graphiques*

Lorsqu'une fonction graphique est exécutée, si aucun dispositif graphique n'est alors ouvert, R ouvrira une fenêtre graphique et y affichera le graphe. Un dispositif graphique peut être ouvert avec une fonction appropriée. La liste des dispositifs graphiques disponibles dépend du système d'exploitation. Les fenêtres graphiques sont nommées `X11` sous Unix/Linux et `windows` sous Windows. Dans tous les cas, on peut ouvrir une fenêtre avec la commande `x11()` qui marche même sous Windows grâce à un alias vers la commande `windows()`. Un dispositif graphique de type fichier sera ouvert avec une fonction qui dépend du format : `postscript()`, `pdf()`, `png()`, ... Pour connaître la liste des dispositifs disponibles pour votre installation, tapez `?device`.

Le dernier dispositif ouvert devient le dispositif graphique actif sur lequel seront affichés les graphes suivants. La fonction `dev.list()` affiche la liste des dispositifs ouverts :

```
> x11(); x11(); pdf()
> dev.list()
X11 X11 pdf
  2   3   4
```

Les chiffres qui s'affichent correspondent aux numéros des dispositifs qui doivent être utilisés si l'on veut changer le dispositif actif. Pour connaître le dispositif actif :

```
> dev.cur()
pdf
  4
```

et pour changer le dispositif actif :

```
> dev.set(3)
X11
  3
```

La fonction `dev.off()` ferme un dispositif graphique : par défaut le dispositif actif est fermé sinon c'est celui dont le numéro est donné comme argument à la fonction. R affiche le numéro du dispositif actif :

[11] Il y a quelques exceptions notables : `hist()` et `barplot()` produisent également des résultats numériques sous forme de liste ou de matrice.

```
> dev.off(2)
X11
  3
> dev.off()
pdf
  4
```

Deux spécificités de la version Windows de R sont à signaler : la fonction `win.metafile` qui accède à un fichier au format Windows Metafile, et un menu “History” affiché lorsque la fenêtre graphique est sélectionnée qui permet d’enregistrer tous les graphes produits au cours d’une session (par défaut l’enregistrement n’est pas activé, l’utilisateur l’active en cliquant sur “Recording” dans ce menu).

#### *4.1.2 Partitionner un graphique*

La fonction `split.screen` partitionne le graphique actif. Par exemple :

```
> split.screen(c(1, 2))
```

va diviser le graphique en deux parties qu’on sélectionnera avec `screen(1)` ou `screen(2)`; `erase.screen()` efface le graphe dernièrement dessiné. Une partie peut être elle-même divisée avec `split.screen()` donnant la possibilité de faire des arrangements complexes.

Ces fonctions sont incompatibles avec d’autres (tel `layout()` ou `coplot()`) et ne doivent pas être utilisées avec des dispositifs graphiques multiples. Leur utilisation doit donc être limitée par exemple pour l’exploration visuelle de données.

La fonction `layout` partitionne le graphique actif en plusieurs parties sur lesquelles sont affichés les graphes successivement. Cette fonction a pour argument principal une matrice avec des valeurs entières qui indiquent les numéros des sous-fenêtres. Par exemple, si l’on veut diviser la fenêtre en quatre parties égales :

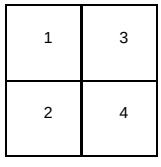
```
> layout(matrix(1:4, 2, 2))
```

On pourra bien sûr créer cette matrice au préalable ce qui permettra de mieux voir comment est divisé le graphique :

```
> mat <- matrix(1:4, 2, 2)
> mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> layout(mat)
```

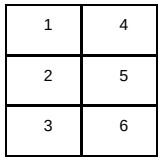
Pour visualiser concrètement la partition créée, on utilisera la fonction `layout.show` avec en argument le nombre de sous-fenêtres (ici 4). Avec cet exemple on aura :

```
> layout.show(4)
```

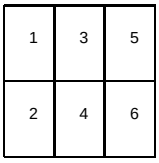



Les exemples qui suivent montrent certaines des possibilités ainsi offertes.

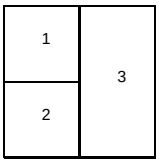
```
> layout(matrix(1:6, 3, 2))
> layout.show(6)
```



```
> layout(matrix(1:6, 2, 3))
> layout.show(6)
```



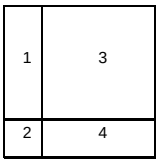

```
> m <- matrix(c(1:3, 3), 2, 2)
> layout(m)
> layout.show(3)
```



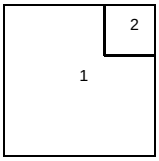

Dans tous ces exemples, nous n'avons pas utilisé l'option `byrow` de `matrix()`, les sous-fenêtres sont donc numérotées par colonne ; il suffit bien sûr de spécifier `matrix(..., byrow = TRUE)` pour que les sous-fenêtres soient numérotées par ligne. On peut aussi donner les numéros dans la matrice dans l'ordre que l'on veut avec, par exemple, `matrix(c(2, 1, 4, 3), 2, 2)`.

Par défaut, `layout()` va partitionner le graphique avec des hauteurs et largeurs régulières : ceci peut être modifié avec les options `widths` et `heights`. Ces dimensions sont données relativement[12]. Exemples :

```
> m <- matrix(1:4, 2, 2)
> layout(m, widths=c(1, 3),
         heights=c(3, 1))
> layout.show(4)
```

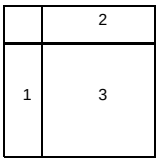



```
> m <- matrix(c(1,1,2,1),2,2)
> layout(m, widths=c(2, 1),
         heights=c(1, 2))
> layout.show(2)
```



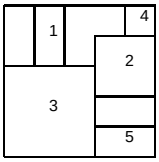

Enfin, les numéros dans la matrice peuvent inclure des 0 donnant la possibilité de construire des partitions complexes (voire ésotériques).

```
> m <- matrix(0:3, 2, 2)
> layout(m, c(1, 3), c(1, 3))
> layout.show(3)
```




[12] Elles peuvent aussi être données en centimètres, *cf.* `?layout`.

```
> m <- matrix(scan(), 5, 5)
1: 0 0 3 3 3 1 1 3 3 3
11: 0 0 3 3 3 0 2 2 0 5
21: 4 2 2 0 5
26:
Read 25 items
> layout(m)
> layout.show(5)
```




## 4.2 Les fonctions graphiques

Voici un aperçu des fonctions graphiques de R.

| | |
|---|---|
| `plot(x)` | graphe des valeurs de `x` (sur l'axe des *y*) ordonnées sur l'axe des *x* |
| `plot(x, y)` | graphe bivarié de `x` (sur l'axe des *x*) et `y` (sur l'axe des *y*) |
| `sunflowerplot(x, y)` | idem que `plot()` mais les points superposés sont dessinés sous forme de fleurs dont le nombre de pétales représente le nombre de points |
| `piechart(x)` | graphe en 'camembert' |
| `boxplot(x)` | graphe 'boites et moustaches' |
| `stripplot(x)` | graphe des valeurs de `x` sur une ligne (une alternative à `boxplot()` pour des petits échantillons) |
| `coplot(x~y \| z)` | graphe bivarié de `x` et `y` pour chaque valeur ou intervalle de valeurs de `z` |
| `interaction.plot (f1, f2, y)` | si `f1` et `f2` sont des facteurs, graphe des moyennes de `y` (sur l'axe des *y*) en fonction des valeurs de `f1` (sur l'axe des *x*) et de `f2` (différentes courbes) ; l'option `fun` permet de choisir la statistique résumée de `y` (par défaut `fun=mean`) |
| `matplot(x,y)` | graphe bivarié de la $1^{\text{ère}}$ colonne de `x` contre la $1^{\text{ère}}$ de `y`, la $2^{\text{ème}}$ de `x` contre la $2^{\text{ème}}$ de `y`, etc. |
| `dotplot(x)` | si `x` est un data.frame, dessine un graphe de Cleveland (graphes superposés ligne par ligne et colonne par colonne) |
| `fourfoldplot(x)` | visualise, avec des quarts de cercles, l'association entre deux variables dichotomiques pour différentes populations (`x` doit être un array avec `dim=c(2, 2, k)` ou une matrice avec `dim=c(2, 2)` si $k = 1$) |
| `assocplot(x)` | graphe de Cohen–Friendly indiquant les déviations de l'hypothèse d'indépendance des lignes et des colonnes dans un tableau de contingence à deux dimensions |
| `mosaicplot(x)` | graphe en 'mosaïque' des résidus d'une régression log-linéaire sur une table de contingence |
| `pairs(x)` | si `x` est une matrice ou un data.frame, dessine tous les graphes bivariés entre les colonnes de `x` |
| `plot.ts(x)` | si `x` est un objet de classe `"ts"`, graphe de `x` en fonction du temps, `x` peut être multivarié mais les séries doivent avoir les mêmes fréquence et dates |
| `ts.plot(x)` | idem mais si `x` est multivarié les séries peuvent avoir des dates différentes et doivent avoir la même fréquence |
| `hist(x)` | histogramme des fréquences de `x` |
| `barplot(x)` | histogramme des valeurs de `x` |
| `qqnorm(x)` | quantiles de `x` en fonction des valeurs attendues selon une loi normale |
| `qqplot(x, y)` | quantiles de `y` en fonction des quantiles de `x` |
| `contour(x, y, z)` | courbes de niveau (les données sont interpolées pour tracer les courbes), `x` et `y` doivent être des vecteurs et `z` une matrice telle que `dim(z)=c(length(x), length(y))` (`x` et `y` peuvent être omis) |
| `filled.contour (x, y, z)` | idem mais les aires entre les contours sont colorées, et une légende des couleurs est également dessinée |
| `image(x, y, z)` | idem mais en couleur (les données sont tracées) |
| `persp(x, y, z)` | idem mais en 3-D (les données sont tracées) |

| | |
|---|---|
| `stars(x)` | si `x` est une matrice ou un data.frame, dessine un graphe en segments ou en étoile où chaque ligne de `x` est représentée par une étoile et les colonnes par les longueurs des branches |
| `symbols(x, y, ...)` | dessine aux coordonnées données par `x` et `y` des symboles (cercles, carrés, rectangles, étoiles, thermomètres ou "boxplots") dont les tailles, couleurs ... sont spécifiées par des arguments supplémentaires |
| `termplot(mod.obj)` | graphe des effets (partiels) d'un modèle de régression (`mod.obj`) |

Pour chaque fonction, les options peuvent être trouvées via l'aide-en-ligne de R. Certaines de ces options sont identiques pour plusieurs fonctions graphiques ; voici les principales (avec leurs éventuelles valeurs par défaut) :

| | |
|---|---|
| `add=FALSE` | si `TRUE` superpose le graphe au graphe existant (s'il y en a un) |
| `axes=TRUE` | si `FALSE` ne trace pas les axes ni le cadre |
| `type="p"` | le type de graphe qui sera dessiné, `"p"` : points, `"l"` : lignes, `"b"` : points connectés par des lignes, `"o"` : idem mais les lignes recouvrent les points, `"h"` : lignes verticales, `"s"` : escaliers, les données étant représentées par le sommet des lignes verticales, `"S"` : idem mais les données étant représentées par le bas des lignes verticales |
| `xlim=, ylim=` | fixe les limites inférieures et supérieures des axes, par exemple avec `xlim=c(1, 10)` ou `xlim=range(x)` |
| `xlab=, ylab=` | annotations des axes, doivent être des variables de mode caractère |
| `main=` | titre principal, doit être une variable de mode caractère |
| `sub=` | sous-titre (écrit dans une police plus petite) |

### 4.3 Les fonctions graphiques secondaires

Il y a dans R un ensemble de fonctions graphiques qui ont une action sur un graphe déjà existant (ces fonctions sont appelées *low-level plotting commands* dans le jargon de R, alors que les fonctions précédentes sont nommées *high-level plotting commands*). Voici les principales :

| | |
|---|---|
| `points(x, y)` | ajoute des points (l'option `type=` peut être utilisée) |
| `lines(x, y)` | idem mais avec des lignes |
| `text(x, y, labels, ...)` | ajoute le texte spécifié par `labels` au coordonnées (`x`,`y`) ; un usage typique sera : `plot(x, y, type="n"); text(x, y, names)` |
| `mtext(text, side=3, line=0, ...)` | ajoute le texte spécifié par `text` dans la marge spécifiée par `side` (*cf.* `axis()` plus bas) ; `line` spécifie la ligne à partir du cadre de traçage |
| `segments(x0, y0, x1, y1)` | trace des lignes des points (`x0`,`y0`) aux points (`x1`,`y1`) |
| `arrows(x0, y0, x1, y1, angle=30, code=2)` | idem avec des flèches aux points (`x0`,`y0`) si `code=2`, aux points (`x1`,`y1`) si `code=1`, ou aux deux si `code=3` ; `angle` contrôle l'angle de la pointe par rapport à l'axe |
| `abline(a,b)` | trace une ligne de pente `b` et ordonnée à l'origine `a` |
| `abline(h=y)` | trace une ligne horizontale sur l'ordonnée `y` |
| `abline(v=x)` | trace une ligne verticale sur l'abcisse `x` |
| `abline(lm.obj)` | trace la droite de régression donnée par `lm.obj` (*cf.* section 5) |
| `rect(x1, y1, x2, y2)` | trace un rectangle délimité à gauche par `x1`, à droite par `x2`, en bas par `y1` et en haut par `y2` |
| `polygon(x, y)` | trace un polygone reliant les points dont les coordonnées sont données par `x` et `y` |
| `legend(x, y, legend)` | ajoute la légende au point de coordonnées (`x`,`y`) avec les symboles donnés par `legend` |

| | |
|---|---|
| `title()` | ajoute un titre et optionnellement un sous-titre |
| `axis(side, vect)` | ajoute un axe en bas (`side=1`), à gauche (2), en haut (3) ou à droite (4) ; `vect` (optionnel) indique les abcisses (ou ordonnées) où les graduations seront tracées |
| `rug(x)` | dessine les données `x` sur l'axe des $x$ sous forme de petits traits verticaux |
| `locator(n, type="n", ...)` | retourne les coordonnées $(x,y)$ après que l'utilisateur ait cliqué `n` fois sur le graphe avec la souris ; également trace des symboles (`type="p"`) ou des lignes (`type="l"`) en fonction de paramètres graphiques optionnels (`...`) ; par défaut ne trace rien (`type="n"`) |

À noter la possibilité d'ajouter des expressions mathématiques sur un graphe à l'aide de `text(x, y, expression(...))`, où la fonction `expression` transforme son argument en équation mathématique. Par exemple,

```
> text(x, y, expression(p == over(1, 1+eˆ-(beta*x+alpha))))
```

va afficher, sur le graphe, l'équation suivante au point de coordonnées $(x,y)$ :

$$\mathrm{p} = \frac{1}{1+\mathrm{e}^{-(\beta \mathrm{x}+\alpha)}}$$

Pour inclure dans une expression une variable numérique on utilisera les fonctions `substitute` et `as.expression` ; par exemple pour inclure une valeur de $R^2$ (précédemment calculée et stockée dans un objet nommé `Rsquared`) :

```
> text(x, y, as.expression(substitute(Rˆ2==r, list(r=Rsquared))))
```

qui affichera sur le graphe au point de coordonnées $(x,y)$ :

$$\mathrm{R}^2 = 0.9856298$$

Pour ne conserver que trois chiffres après la virgule on modifiera le code comme suit :

```
> text(x, y, as.expression(substitute(Rˆ2==r,
+                          list(r=round(Rsquared, 3)))))
```

qui affichera :

$$\mathrm{R}^2 = 0.986$$

Enfin, pour obtenir le R en italique :

```
> text(x, y, as.expression(substitute(italic(R)ˆ2==r,
+                          list(r=round(Rsquared, 3)))))
```

$$R^2 = 0.986$$

### 4.4 Les paramètres graphiques

En plus des fonctions graphiques secondaires, la présentation des graphiques peut être améliorée grâce aux paramètres graphiques. Ceux-ci s'utilisent soit comme des options des fonctions graphiques principales ou secondaires (mais cela ne marche pas pour tous), soit à l'aide de la fonction `par` qui permet d'enregistrer les changements des paramètres graphiques de façon permanente, c'est-à-dire que les graphes suivants seront dessinés en fonction des nouveaux paramètres spécifiés par l'utilisateur. Par exemple, l'instruction suivante :

```
> par(bg="yellow")
```

fera que tous les graphes seront dessinés avec un fond jaune. Il y a 68 paramètres graphiques, dont certains ont des rôles proches. La liste détaillée peut être obtenue avec `?par` ; je me limite ici à ceux qui sont les plus couramment utilisés.

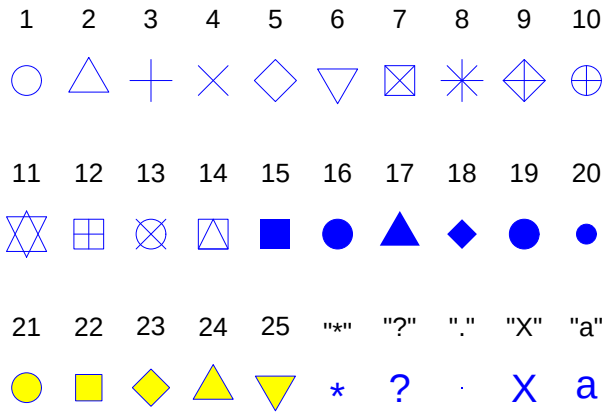



FIG. 2 – Les symboles pour tracer des points avec R (`pch=1:25`). Les couleurs ont été obtenues avec les options `col="blue", bg="yellow"`, la seconde option n'a un effet que pour les symboles 21 à 25. N'importe quel caractère peut être utilisé (`pch="*", "?", ".", ...`).

| | |
|---|---|
| `adj` | contrôle la justification du texte (`0` à gauche, `0.5` centré, `1` à droite) |
| `bg` | spécifie la couleur de l'arrière-plan (ex. : `bg="red"`, `bg="blue"`, ... la liste des 657 couleurs disponibles est affichée avec `colors()`) |
| `bty` | contrôle comment le cadre est tracé, valeurs permises : `"o"`, `"l"`, `"7"`, `"c"`, `"u"` ou `"]"` (le cadre ressemblant au caractère correspondant) ; `bty="n"` supprime le cadre |
| `cex` | une valeur qui contrôle la taille des caractères et des symboles par rapport au défaut ; les paramètres suivants ont le même contrôle pour les nombres sur les axes, `cex.axis`, les annotations des axes, `cex.lab`, le titre, `cex.main`, le sous-titre, `cex.sub` |
| `col` | contrôle la couleur des symboles ; comme pour `cex` il y a : `col.axis`, `col.lab`, `col.main`, `col.sub` |
| `font` | un entier qui contrôle le style du texte (`1` : normal, `2` : italique, `3` : gras, `4` : gras italique) ; comme pour `cex` il y a : `font.axis`, `font.lab`, `font.main`, `font.sub` |
| `las` | un entier qui contrôle comment sont disposées les annotations des axes (`0` : parallèles aux axes, `1` : horizontales, `2` : perpendiculaires aux axes, `3` : verticales) |
| `lty` | contrôle le type de ligne tracée, peut être un entier (`1` : continue, `2` : tirets, `3` : points, `4` : points et tirets alternés, `5` : tirets longs, `6` : tirets courts et longs alternés), ou ensemble de 8 caractères maximum (entre `"0"` et `"9"`) qui spécifie alternativement la longueur, en points ou pixels, des éléments tracés et des blancs, par exemple `lty="44"` aura le même effet que `lty=2` |
| `lwd` | une valeur numérique qui contrôle la largeur des lignes |
| `mar` | un vecteur de 4 valeurs numériques qui contrôle l'espace entre les axes et le bord de la figure de la forme `c(bas, gauche, haut, droit)`, les valeurs par défaut sont `c(5.1, 4.1, 4.1, 2.1)` |
| `mfcol` | un vecteur de forme `c(nr,nc)` qui partitionne la fenêtre graphique en une matrice de `nr` lignes et `nc` colonnes, les graphes sont ensuite dessinés en colonne (*cf.* section 4.1.2) |
| `mfrow` | idem mais les graphes sont ensuite dessinés en ligne (*cf.* section 4.1.2) |
| `pch` | contrôle le type de symbole, soit un entier entre 1 et 25, soit n'importe quel caractère entre guillements (FIG. 2) |
| `ps` | un entier qui contrôle la taille en points du texte et des symboles |
| `pty` | un caractère qui spécifie la forme du graphe, `"s"` : carrée, `"m"` : maximale |
| `tck` | une valeur qui spécifie la longueur des graduations sur les axes en fraction du plus petit de la largeur ou de la hauteur du graphe ; si `tck=1` une grille est tracée |
| `tcl` | une valeur qui spécifie la longueur des graduations sur les axes en fraction de la hauteur d'une ligne de texte (défaut `tcl=-0.5`) |
| `xaxt` | si `xaxt="n"` l'axe des *x* est défini mais pas tracé (utile avec `axis(side=1, ...)`) |
| `yaxt` | si `yaxt="n"` l'axe des *y* est défini mais pas tracé (utile avec `axis(side=2, ...)`) |

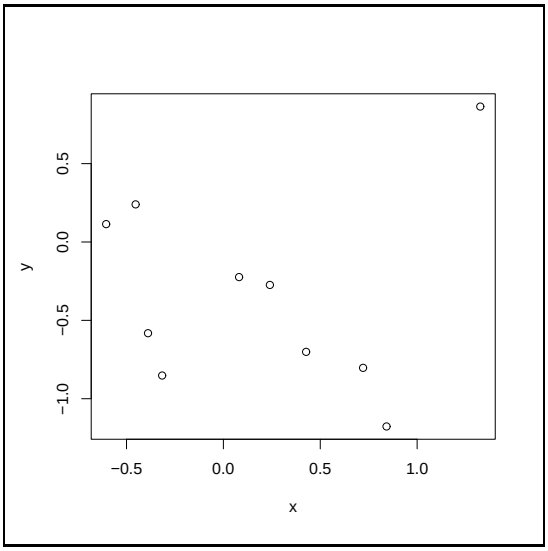



FIG. 3 – La fonction `plot` utilisée sans options.

### 4.5 Un exemple concret

Afin d'illustrer l'utilisation des fonctionnalités graphiques de R, considérons un cas concret et simple d'un graphe bivarié de 10 paires de valeurs aléatoires. Ces valeurs ont été générées avec :

```
> x <- rnorm(10)
> y <- rnorm(10)
```

Le graphe voulu sera obtenu avec `plot()` ; on tapera la commande :

```
> plot(x, y)
```

et le graphique sera dessiné sur le graphique actif. Le résultat est représenté FIG. 3. Par défaut, R dessine les graphiques de façon "intelligente" : l'espacement entre les graduations sur les axes, la disposition des annotations, etc. sont calculés afin que le graphique obtenu soit le plus intelligible possible.

L'utilisateur peut toutefois vouloir changer l'allure du graphe, par exemple, pour conformer ses figures avec un style éditorial prédéfini ou les personnaliser pour un séminaire. La façon la plus simple de changer la présentation d'un graphe est d'ajouter des options qui modifieront les arguments par défaut. Dans notre cas, on peut modifier de façon appréciable notre figure de la façon suivante :

```
plot(x, y, xlab="Ten random values", ylab="Ten other values",
     xlim=c(-2, 2), ylim=c(-2, 2), pch=22, col="red",
     bg="yellow", bty="l", tcl=0.4,
     main="How to customize a plot with R", las=1, cex=1.5)
```

Le résultat est la FIG. 4. Voyons en détail chacune des options utilisée. D'abord `xlab` et `ylab` vont changer les annotations sur les axes qui, par défaut, étaient les noms des variables. Ensuite, `xlim` et `ylim` nous permettent de définir les limites sur les deux axes[13]. Le paramètre graphique `pch` a été ici utilisé comme option : `pch=22` spécifie un carré dont la couleur du contour et celle de l'intérieur peuvent être différentes et qui sont données, respectivement, par `col` et `bg`. On se reportera au tableau sur les paramètres graphiques pour comprendre les modifications apportées par `bty`, `tcl`, `las` et `cex`. Enfin, un titre a été ajouté par l'option `main`.

[13] Par défaut, R ajoute 4% de part et d'autre des limites des axes. Ce comportement peut être supprimé en mettant les paramètres graphiques `xaxs="i"` et `yaxs="i"` (ceux-ci peuvent être passés comme options à `plot()`).

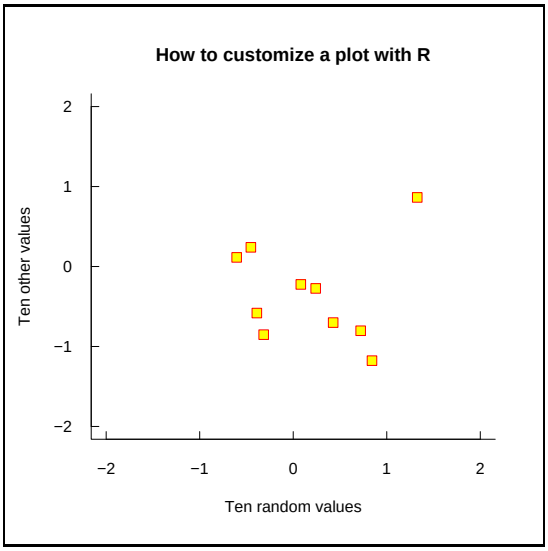



FIG. 4 – La fonction `plot` utilisée avec options.

Les paramètres graphiques et les fonctions graphiques secondaires permettent d'aller plus loin dans la présentation d'un graphe. Comme vu précédemment, certains paramètres graphiques ne peuvent pas être passés comme arguments dans une fonction comme `plot`. On va maintenant modifier certains de ces paramètres avec `par()`, il est donc nécessaire cette fois de taper plusieurs commandes. Quand on change les paramètres graphiques, il est utile de sauver les valeurs initiales de ces paramètres au préalable afin de pouvoir les rétablir par la suite. Voici les commandes pour obtenir la FIG. 5.

```
opar <- par()
par(bg="lightyellow", col.axis="blue", mar=c(4, 4, 2.5, 0.25))
plot(x, y, xlab="Ten random values", ylab="Ten other values",
     xlim=c(-2, 2), ylim=c(-2, 2), pch=22, col="red", bg="yellow",
     bty="l", tcl=-.25, las=1, cex=1.5)
title("How to customize a plot with R (bis)", font.main=3, adj=1)
par(opar)
```

Détaillons les actions provoquées par ces commandes. Tout d'abord, les paramètres graphiques par défaut sont sauvés dans une liste qui est nommée, par exemple, `opar`. Trois paramètres vont être modifiés ensuite : `bg` pour la couleur de l'arrière-plan, `col.axis` pour la couleur des chiffres sur les axes et `mar` pour les dimensions des marges autour du cadre de traçage. Le graphe est tracé de façon presque similaire que pour la FIG. 4. On voit que la modification des marges a permis d'utiliser de l'espace libre autour du cadre de traçage. Le titre est ajouté cette fois avec la fonction graphique secondaire `title` ce qui permet de passer certains paramètres en arguments sans altérer le reste du graphique. Enfin, les paramètres graphiques initiaux sont restaurés avec la dernière commande.

Maintenant, le contrôle total ! Sur la FIG. 5 R détermine encore certaines choses comme le nombre de graduations sur les axes ou l'espace entre le titre et le cadre de traçage. Nous allons maintenant contrôler totalement la présentation du graphique. L'approche utilisée ici est de tracer le graphe "à blanc" avec `plot(..., type="n")`, puis d'ajouter les points, les axes, les annotations, etc, avec des fonctions graphiques secondaires. On se permettra aussi quelques fantaisies, comme de changer la couleur de fond du cadre de traçage. Les commandes suivent, et le graphe produit est la FIG. 6.

```
opar <- par()
```

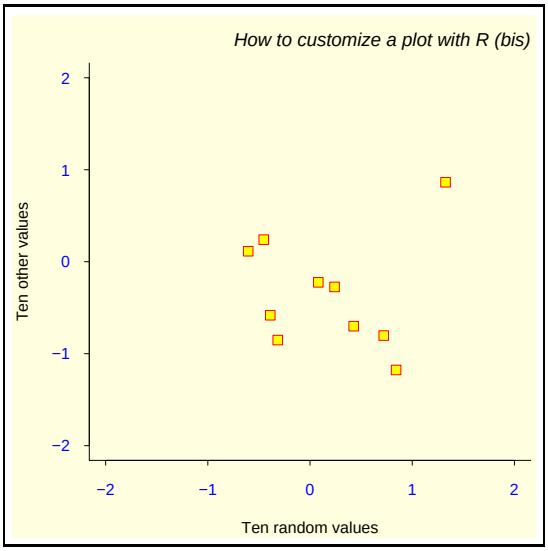



FIG. 5 – Les fonctions `par`, `plot` et `title`.

```
par(bg="lightgray", mar=c(2.5, 1.5, 2.5, 0.25))
plot(x, y, type="n", xlab="", ylab="", xlim=c(-2, 2),
     ylim=c(-2, 2), xaxt="n", yaxt="n")
rect(-3, -3, 3, 3, col="cornsilk")
points(x, y, pch=10, col="red", cex=2)
axis(side=1, c(-2, 0, 2), tcl=-0.2, labels=FALSE)
axis(side=2, -1:1, tcl=-0.2, labels=FALSE)
title("How to customize a plot with R (ter)",
      font.main=4, adj=1, cex.main=1)
mtext("Ten random values", side=1, line=1, at=1, cex=0.9, font=3)
mtext("Ten other values", line=0.5, at=-1.8, cex=0.9, font=3)
mtext(c(-2, 0, 2), side=1, las=1, at=c(-2, 0, 2), line=0.3,
      col="blue", cex=0.9)
mtext(-1:1, side=2, las=1, at=-1:1, line=0.2, col="blue", cex=0.9)
par(opar)
```

Comme précédemment, les paramètres graphiques par défaut sont enregistrés et la couleur de l'arrière-plan est changé ainsi que les marges. Le graphe est ensuite dessiné avec `type="n"` pour ne pas tracer les points, `xlab="", ylab=""` pour ne pas marquer les noms des axes et `xaxt="n", yaxt="n"` pour ne pas tracer les axes. Le résultat est de tracer uniquement le cadre de traçage et de définir les axes en fonction de `xlim` et `ylim`. Notez qu'on aurait pu utiliser l'option `axes=FALSE` mais dans ce cas ni les axes ni le cadre n'auraient été tracés.

Les éléments sont ensuite ajoutés dans le cadre ainsi défini avec des fonctions graphiques secondaires. Avant d'ajouter les points, on va changer la couleur dans le cadre avec `rect()` : les dimensions du rectangle sont choisies afin de dépasser largement celles du cadre.

Les points sont tracés avec `points()`; on a cette fois changé de symbole. Les axes sont ajoutés avec `axis()` : le vecteur qui est passé en second argument donne les coordonnées des graduations qui doivent être tracées. L'option `labels=FALSE` spécifie qu'aucune annotation n'est ajoutée avec les graduations. Cette option accepte aussi un vecteur de mode caractère, par exemple `labels=c("A", "B", "C")`.

Le titre est ajouté avec `title()`, mais on a changé légèrement la police. Les annotations des axes sont mises avec `mtext()` (*marginal text*). Le premier argument de cette fonction est

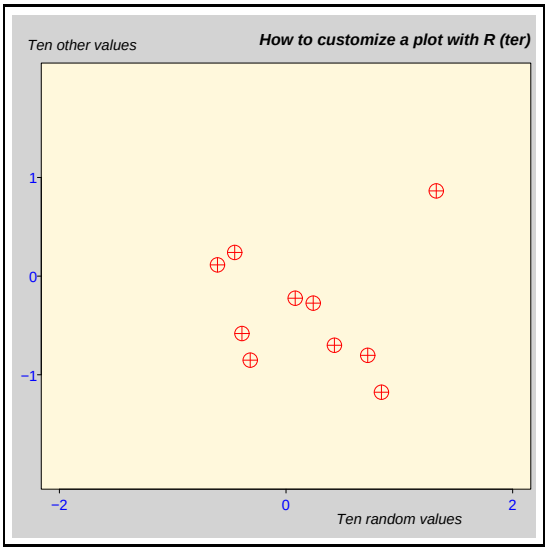



FIG. 6 – Un graphe fait “sur mesure”.

un vecteur de mode caractère qui donne le texte à afficher. L’option `line` indique la distance à partir du cadre de traçage (par défaut `line=0`), et `at` la coordonnée. Le second appel à `mtext()` utilise la valeur par défaut de `side` (3). Les deux autres appels de `mtext()` passent un vecteur numérique en premier argument : celui-ci sera converti en mode caractère.

### 4.6 Les packages grid et lattice

Les packages grid et lattice représentent l’implémentation dans R des graphiques de type Trellis de S-PLUS. Trellis est une approche pour la visualisation de données multivariées particulièrement appropriée pour l’exploration de relations ou d’interactions entre variables[14].

L’idée principale derrière lattice (tout comme Trellis) est celle des graphes multiples conditionnés : un graphe bivarié entre deux variables sera découpé en plusieurs graphes en fonction des valeurs d’une troisième variable. La fonction `coplot` utilise une approche similaire, mais lattice offre des fonctionnalités plus vastes que cette fonction.

Les graphes produits par lattice ou grid ne peuvent pas être combinés ou mélangés avec ceux produits par les fonctions graphiques vues précédemment, car ces packages utilisent un nouveau mode graphique[15]. Ce nouveau mode possède son propre système de paramètres graphiques qui sont distincts de ceux vus plus haut. On peut par contre utiliser les deux modes graphiques dans la même session sur le même dispositif graphique.

D’un point de vue pratique, grid contient les fonctions nécessaires au mode graphique, alors que les fonctions graphiques plus susceptibles d’être utilisées couramment sont dans lattice.

La plupart des fonctions de lattice prennent pour argument principal une formule, par exemple `y ~ x`[16]. La formule `y ~ x | z` signifie que le graphe de `y` en fonction de `x` sera dessiné en plusieurs sous-graphes en fonction des valeurs de `z`.

Le tableau ci-dessous indique les principales fonctions de lattice. La formule donnée en argument est la formule type nécessaire, mais toutes ces fonctions acceptent une formule conditionnelle (`y ~ x | z`) comme argument principal ; dans ce cas un graphe multiple, en fonction des

[14] http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/index.html

[15] Ce mode graphique devrait palier certaines faiblesses de l’ancien, comme le manque d’interactivité directe avec les graphiques.

[16] `plot()` accepte également une formule en argument principal : si `x` et `y` sont deux vecteurs de même longueur, `plot(y ~ x)` et `plot(x, y)` donneront des graphiques identiques.

valeurs de z, est dessiné comme on le verra dans les exemples ci-dessous.

| | |
|---|---|
| `barchart(y ~ x)` | histogramme des valeurs de y en fonction de celles de x |
| `bwplot(y ~ x)` | graphe 'boites et moustaches' |
| `densityplot(~ x)` | graphe de fonctions de densité |
| `dotplot(y ~ x)` | graphe de Cleveland (graphes superposés ligne par ligne et colonne par colonne) |
| `histogram(~ x)` | histogrammes des fréquences de x |
| `qqmath(~ x)` | quantiles de x en fonction des valeurs attendues selon une distribution théorique |
| `stripplot(y ~ x)` | graphe unidimensionnel, x doit être numérique, y peut être un facteur |
| `qq(y ~ x)` | quantiles pour comparer deux distributions, x doit être numérique, y peut être numérique, caractère ou facteur mais doit avoir deux 'niveaux' |
| `xyplot(y ~ x)` | graphes bivariés (avec de nombreuses fonctionnalités) |
| `levelplot(z ~ x*y)` | graphe en couleur des valeurs de z aux coordonnées fournies par x et y (x, y et z sont tous de même longueur) |
| `splom(~ x)` | matrice de graphes bivariés |
| `parallel(~ x)` | graphe de coordonnées parallèles |

Certaines fonctions de lattice ont le même nom que des fonctions graphiques du package base. Ces dernières sont "masqués" lorsque lattice est chargé en mémoire.

Voyons maintenant quelques exemples afin d'illustrer quelques aspects de lattice. Il faut au préalable charger le package en mémoire avec la commande `library(lattice)` afin d'accéder aux fonctions.

D'abord, les graphes de fonctions de densité. Un tel graphe peut être dessiné simplement avec `densityplot(~ x)` qui tracera une courbe de densité empirique ainsi que les points correspondants aux observations sur l'axe des *x* (comme `rug()`). Notre exemple sera un peu plus compliqué avec la superposition, sur chaque graphe, des courbes de densité empirique et de densité estimée avec une loi normale. Il nous faut à cette fin utiliser l'argument `panel` qui définit ce qui doit être tracé dans chaque graphe. Les commandes sont :

```
n <- seq(5, 45, 5)
x <- rnorm(sum(n))
y <- factor(rep(n, n), labels=paste("n =", n))
densityplot(~ x | y,
            panel = function(x, ...) {
                panel.densityplot(x, col="DarkOliveGreen", ...)
                panel.mathdensity(dmath=dnorm,
                                  args=list(mean=mean(x), sd=sd(x)),
                                  col="darkblue")
            })
```

Les trois premières lignes génèrent un échantillon de variables normales que l'on divise en sous-échantillons d'effectif égal à 5, 10, 15, ... et 45. Ensuite vient l'appel de `densityplot()` qui produit un graphe par sous-échantillon. `panel` prend pour argument une fonction. Dans notre exemple, nous avons défini une fonction qui fait appel à deux fonctions prédéfinies dans lattice : `panel.densityplot` qui trace la fonction de densité empirique et `panel.mathdensity` qui trace la fonction de densité estimée avec une loi normale. La fonction `panel.densityplot` est appellée par défaut si aucun argument n'est donné à `panel` : la commande `densityplot(~ x | y)` aurait donné le même graphe que sur la FIG. 7 mais sans les courbes bleues.

Les exemples suivants utilisent des données disponibles dans R : les localisations de 1000 séismes près des îles Fidji et des données biométriques sur des fleurs de trois espèces d'iris.

La FIG. 8 représente la localisation géographique des séismes en fonction de la profondeur. Les commandes nécessaires pour ce graphe sont :

```
data(quakes)
```

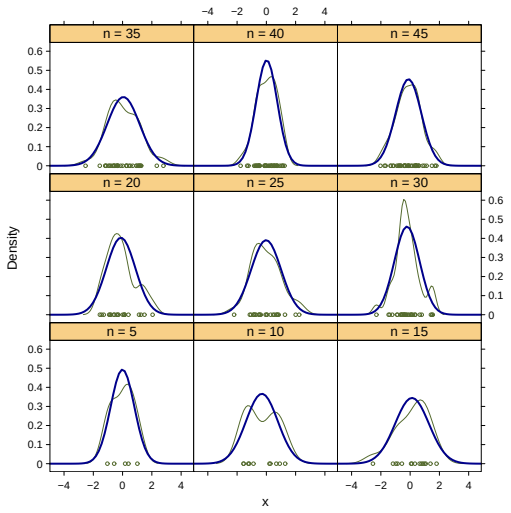



FIG. 7 – La fonction `densityplot`.

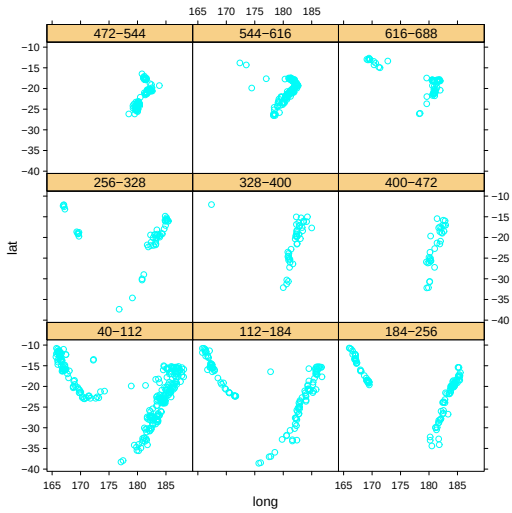



FIG. 8 – La fonction `xyplot` avec les données “quakes”.

```
mini <- min(quakes$depth)
maxi <- max(quakes$depth)
int <- ceiling((maxi - mini)/9)
inf <- seq(mini, maxi, int)
quakes$depth.cat <- factor(floor(((quakes$depth - mini) / int)),
                           labels=paste(inf, inf + int, sep="-"))
xyplot(lat ~ long | depth.cat, data = quakes)
```

La première commande charge le jeu de données `quakes` en mémoire. Les cinq commandes suivantes créent un facteur en divisant la profondeur (variable `depth`) en neuf intervalles d’étendues égales : les niveaux de ce facteur sont nommés avec les bornes inférieures et supérieures de ces intervalles. Il suffit ensuite d’appeller la fonction `xyplot` avec la formule appropriée et un argument `data` qui indique où `xyplot` doit chercher les variables[17].

[17] `plot()` ne peut pas prendre d’argument `data`, la localisation des variables doit être donnée explicitement, par

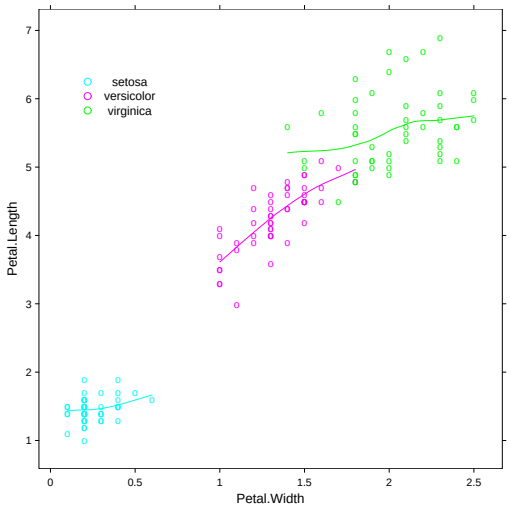



FIG. 9 – La fonction `xyplot` avec les données “iris”.

Avec les données `iris`, le chevauchement entre les différentes espèces est suffisament faible pour les représenter ensemble sur la même figure (FIG. 9). Les commandes correspondantes sont :

```
data(iris)
xyplot(
  Petal.Length ~ Petal.Width, data = iris, groups=Species,
  panel = panel.superpose,
  type = c("p", "smooth"), span=.75,
  key = list(x=0.15, y=0.85,
    points=list(col=trellis.settings[["superpose.symbol"]]$col[1:3],
                pch = 1),
    text = list(levels(iris$Species)))
)
```

L’appel de la fonction `xyplot` est ici un peu plus complexe que dans l’exemple précédent et utilise plusieurs options que nous allons détailler. L’option `groups`, comme son nom l’indique, définit des groupes qui seront utilisés par les autres options. On a déjà vu l’option `panel` qui définit comment les différents groupes vont être représentés sur la graphe : on utilise ici une fonction pré-définie `panel.superpose` afin de superposer les groupes sur le même graphe. Aucune option n’étant passée à `panel.superpose`, les couleurs par défaut seront utilisées pour distinguer les groupes. L’option `type`, comme dans `plot()`, précise le type de traçage, sauf qu’ici on peut donner plusieurs arguments sous forme d’un vecteur : `"p"` pour tracer les points et `"smooth"` pour tracer une courbe de “lissage” dont le degré de lissage est donné par `span`. L’option `key` ajoute la légende au graphe ; sa syntaxe est assez compliquée mais ceci devrait être simplifier dans les futures versions de lattice pour arriver à quelque chose similaire à la fonction `legend` des graphiques standards. `key` prend comme argument une liste : `x` et `y` indiquent l’emplacement de la légende (si ces coordonnées sont omises, la légende est placée en dehors du cadre) ; `points` spécifie le type de symbole dessiné dans la légende qu’il est nécessaire d’extraire dans les définitions par défaut (d’où une expression un peu compliquée) ; et `text` donne le texte de la légende qui est bien sûr ici les noms d’espèces.

Nous allons voir maintenant la fonction `splom` avec les mêmes données sur les iris. Les commandes suivantes ont servi à produire la FIG. 10 :

exemple `plot(quakes$long ~ quakes$lat)`.

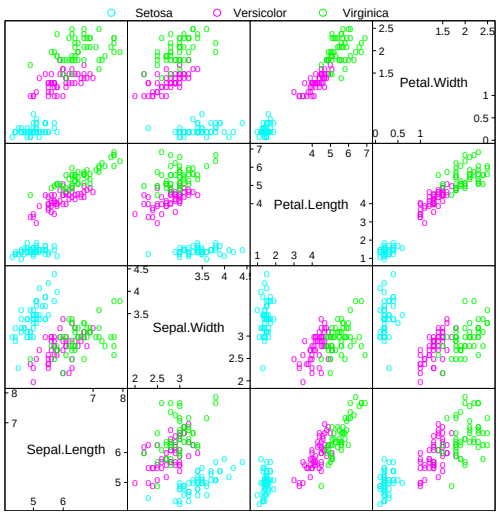



FIG. 10 – La fonction `splom` avec les données “iris” (1).

```
splom(
  ~iris[1:4], groups = Species, data = iris, xlab = "",
  panel = panel.superpose,
  key = list(columns = 3,
    points = list(col=trellis.settings[["superpose.symbol"]]$col[1:3],
                  pch = 1),
    text = list(c("Setosa", "Versicolor", "Virginica")))
)
```

L’argument principal est cette fois une matrice (les quatre premières colonnes d’`iris`). Le résultat est l’ensemble des graphes bivariés possibles entre les variables de la matrice, tout comme la fonction standard `pairs`. Par défaut, `splom` ajoute le texte “Scatter Plot Matrix” sous l’axe des *x* : pour l’éviter on a précisé `xlab=""`. Le reste des options est similaire à l’exemple précédent, sauf qu’on a précisé `columns = 3` pour `key` afin que la légende soit disposée sur trois colonnes.

La FIG. 10 aurait pu être faite avec `pairs()`, mais cette fonction ne peut pas produire des graphes conditionnés comme sur la FIG. 11. Le code utilisé est relativement simple :

```
splom(~iris[1:3] | Species, data = iris, pscales = 0,
      varnames = c("Sepal\nLength", "Sepal\nWidth", "Petal\nLength"))
```

Les sous-graphes étant assez petits, on a ajouté deux options pour améliorer la lisibilité de la figure : `pscales = 0` supprime les graduations des axes (tous les sous-graphes sont à la même échelle), et on a redéfini les noms des variables pour les faire tenir sur deux lignes (`"\n"` code pour un saut de ligne dans une chaine de caractères).

Le dernier exemple utilise la méthode des coordonnées parallèles pour l’analyse exploratoire de données multivariées. Les variables sont alignées sur un axe (par exemple sur l’axe des *y*) et les valeurs observées sont représentées sur l’autre axe (les variables étant mises à la même échelle, par exemple en les réduisant). Les valeurs correspondant au même individu sont reliées par une ligne. Avec les données `iris` on obtient la FIG. 12 avec le code suivant :

```
parallel(~iris[, 1:4] | Species, data = iris, layout = c(3, 1))
```

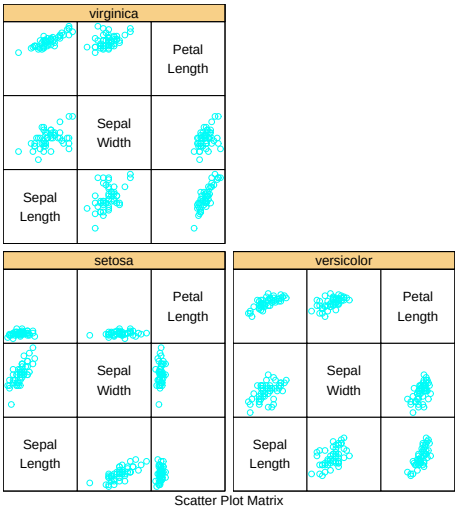



FIG. 11 – La fonction `splom` avec les données “iris” (2).

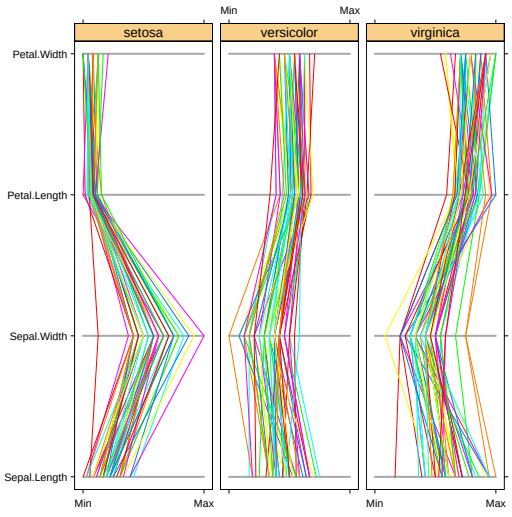



FIG. 12 – La fonction `parallel` avec les données “iris”.

# 5 Les analyses statistiques avec R

Encore plus que pour les graphiques, il est impossible ici d'aller dans les détails sur les possibilités offertes par R pour les analyses statistiques. Mon but est ici de donner des points de repères afin de se faire une idée sur les caractéristiques de R pour conduire des analyses de données.

À l'exception des fonctions dans les packages grid et lattice, toutes les fonctions que nous avons vues jusqu'à maintenant sont contenues dans le package base. Certaines fonctions pour l'analyse des données sont dans base mais la grande majorité des méthodes statistiques dans R sont distribuées sous forme de *package*. Certains de ces packages sont installés avec base, d'autres sont *recommandés* car ils couvrent un éventail de méthodes couramment utilisées, et enfin de nombreux autres packages sont *contribués* et doivent être installés par l'utilisateur.

On commencera par un exemple simple, qui ne nécessite aucun package autre que base, afin de présenter l'approche générale pour analyser des données avec R. Puis on détaillera certaines notions qui sont utiles en général quelque soit le type d'analyse que l'on veut conduire tel les *formules* et les *fonctions génériques*. Ensuite, on dressera une vue d'ensemble sur les packages.

## 5.1 Un exemple simple d'analyse de variance

Il y a trois fonctions statistiques principales dans le package base : `lm`, `glm` et `aov` pour, respectivement, les modèles linéaires, les modèles linéaires généralisés et les analyses de variance. On peut aussi mentionner `loglin` pour les modèles log-linéaires mais cette fonction prend un tableau de contingence comme argument principal au lieu d'une formule[18]. Pour nous essayer à l'analyse de variance, prenons un jeu de données disponible dans R : `InsectSprays`. Six insecticides ont été testés en culture, la réponse observée étant le nombre d'insectes. Chaque insecticide ayant été testé 12 fois, on a donc 72 observations. Laissons de côté l'exploration graphique de ces données pour se consacrer à une simple analyse de variance de la réponse en fonction de l'insecticide. Après avoir chargé les données en mémoire à l'aide de la fonction `data`, l'analyse sera faite avec la fonction `aov` (après transformation de la réponse) :

```
> data(InsectSprays)
> aov.spray <- aov(sqrt(count) ~ spray, data = InsectSprays)
```

L'argument principal (et obligatoire) d'`aov()` est une formule qui précise la réponse à gauche du signe ~ et le prédicteur à droite. L'option `data = InsectSprays` précise que les variables doivent être prises dans le data.frame `InsectSprays`. Cette syntaxe est équivalente à :

```
> aov.spray <- aov(sqrt(InsectSprays$count) ~ InsectSprays$spray)
```

ou encore (si l'on connait les numéros de colonne des variables) :

```
> aov.spray <- aov(sqrt(InsectSprays[, 1]) ~ InsectSprays[, 2])
```

On préferera la première syntaxe qui est plus claire.

Les résultats ne sont pas affichés car ceux-ci sont copiés dans un objet nommé `aov.spray`. On utilisera ensuite certaines fonctions pour extraire les résultats désirés, par exemple `print()` pour afficher un bref résumé de l'analyse (essentiellement les paramètres estimés) et `summary()` pour afficher plus de détails (dont les tests statistiques) :

```
> aov.spray
Call:
   aov(formula = sqrt(count) ~ spray, data = InsectSprays)

Terms:
```

[18] Le package MASS a la fonction `loglm` qui permet de faire passer des formules comme argument à `loglin`.

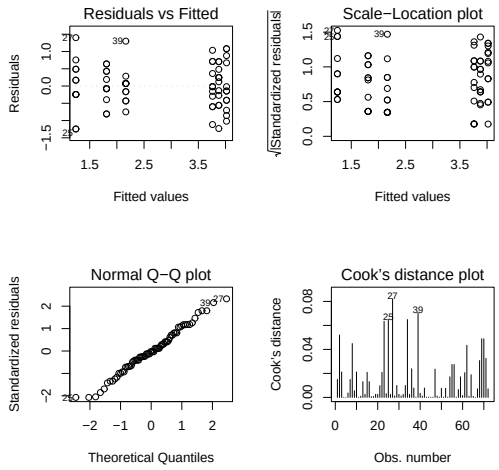



FIG. 13 – Représentation graphique des résultats de la fonction `aov` avec `plot()`.

```
                  spray Residuals
Sum of Squares  88.43787  26.05798
Deg. of Freedom        5        66

Residual standard error: 0.6283453
Estimated effects may be unbalanced
> summary(aov.spray)
            Df Sum Sq Mean Sq F value    Pr(>F)
spray        5 88.438  17.688  44.799 < 2.2e-16 ***
Residuals   66 26.058   0.395
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Rappelons que de taper le nom de l'objet en guise de commande équivaut à la commande `print(aov.spray)`. Une représentation graphique des résultats peut être obtenue avec `plot()` ou `termplot()`. Avant de taper `plot(aov.spray)`, on divisera le graphique en quatre afin que les quatre graphes diagnostiques soient dessinés sur le même graphe. Les commandes sont :

```
> opar <- par()
> par(mfcol = c(2, 2))
> plot(aov.spray)
> par(opar)
> termplot(aov.spray, se=TRUE, partial.resid=TRUE, rug=TRUE)
```

et les graphes obtenus sont représentés FIG. 13 et FIG. 14.

## 5.2 Les formules

Les formules sont un élément-clef des analyses statistiques avec R : la notation utilisée est la même pour (presque) toutes les fonctions. Une formule est typiquement de la forme `y ~ model` où `y` est la réponse analysée et `model` est un ensemble de termes pour lesquels les paramètres sont estimés. Ces termes sont séparés par des symboles arithmétiques mais qui ont ici une signification particulière.

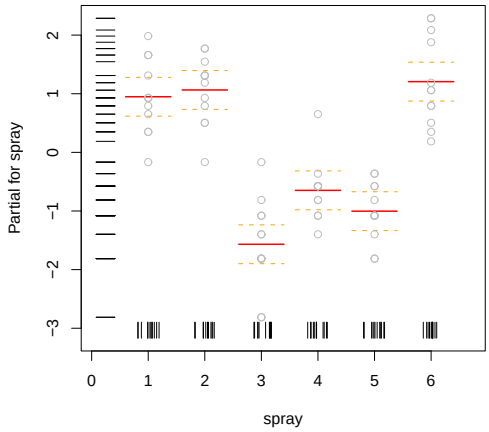



FIG. 14 – Représentation graphique des résultats de la fonction `aov` avec `termplot()`.

| | |
|---|---|
| `a+b` | effets additifs de `a` et de `b` |
| `X` | si `X` est une matrice, ceci équivaut à un effet additif de toutes ses colonnes, c'est-à-dire `X[,1]+X[,2]+...+X[,ncol(X)]`; certaines de ces colonnes peuvent être sélectionnées avec l'indexation numérique (ex. : `X[,2:4]`) |
| `a:b` | effet interactif entre `a` et `b` |
| `a*b` | effets additifs et interactifs (identique à `a+b+a:b`) |
| `poly(a, n)` | polynome de `a` jusqu'au degré *n* |
| `^n` | inclue toutes les interactions jusqu'au niveau *n*, c'est-à-dire `(a+b+c)^2` est identique à `a+b+c+a:b+a:c+b:c` |
| `b %in% a` | les effets de `b` sont hiérarchiquement inclus dans `a` (identique à `a+a:b` ou `a/b`) |
| `a-b` | supprime l'effet de `b`, par exemple : `(a+b+c)^2-a:b` est identique à `a+b+c+a:c+b:c` |
| `-1` | `y~x-1` force la régression à passer par l'origine (idem pour `y~x+0` ou `0+y~x`) |
| `1` | `y~1` ajuste un modèle sans effets (juste l'"intercept") |
| `offset(...)` | ajoute un effet au modèle sans estimer de paramètre (par ex., `offset(3*x)`) |

On voit que les opérateurs arithmétiques de R ont dans une formule un sens différent de celui qu'ils ont dans une expression classique. Par exemple, la formule `y~x1+x2` définira le modèle $y = \beta_1 x_1 + \beta_2 x_2 + \alpha$, et non pas (si l'opérateur `+` avait sa fonction habituelle) $y = \beta(x_1 + x_2) + \alpha$. Pour inclure des opérations arithmétiques dans une formule, on utilisera la fonction `I()` : la formule `y~I(x1+x2)` définira alors le modèle $y = \beta(x_1 + x_2) + \alpha$. De même, pour définir le modèle $y = \beta_1 x + \beta_2 x^2 + \alpha$ on utilisera la formule `y ~ poly(x, 2)` (et non pas `y ~ x + x^2`).

Pour les analyses de variance, `aov()` accepte une syntaxe particulière pour spécifier les effets aléatoires. Par exemple, `y ~ a + Error(b)` signifie effets additifs d'un terme fixe (`a`) et d'un terme aléatoire (`b`).

## 5.3 Les fonctions génériques

On se souvient que les fonctions de R agissent en fonction des attributs des objets éventuellement passés en arguments. Les objets qui contiennent les résultats d'une analyse ont, quant à eux, un attribut particulier nommé la *classe* qui contient la signature de la fonction qui a fait l'analyse. Les fonctions qui serviront ensuite à extraire des informations de l'objet-résultat agiront spécifiquement en fonction de la classe de l'objet. Ces fonctions sont dites *génériques*.

Par exemple, la fonction la plus utilisée pour extraire des résultats d'analyse est `summary` qui permet d'afficher les résultats détaillés. Selon que l'objet qui est passé en argument est de classe `"lm"` (modèle linéaire) ou `"aov"` (analyse de variance), il est clair que les informations à afficher ne seront pas les mêmes. L'avantage des fonctions génériques est d'avoir une syntaxe unique pour toutes les analyses.

Un objet qui contient les résultats d'une analyse est généralement une liste dont l'affichage est déterminée par un attribut classe. On a déjà vu cette notion que les fonctions de R agissent spécifiquement en fonction de la nature des objets qui sont donnés en arguments. C'est un caractère général de R[19]. Le tableau suivant donne les principales fonctions génériques qui permettent d'extraire des informations d'un objet qui résulte d'une analyse. L'usage typique de ces fonctions étant :

```
> mod <- lm(y ~ x)
> df.residual(mod)
[1] 8
```

| | |
|---|---|
| `print` | retourne un résumé succint |
| `summary` | retourne un résumé détaillé |
| `df.residual` | retourne le nombre de degrés de liberté résiduel |
| `coef` | retourne les coefficients estimés (avec parfois leurs erreurs-standards) |
| `residuals` | retourne les résidus |
| `deviance` | retourne la déviance |
| `fitted` | retourne les valeurs ajustées par le modèle |
| `logLik` | calcule le logarithme de la vraisemblance et le nombre de paramètre d'un modèle |
| `AIC` | calcule le critère d'information d'Akaike ou AIC (dépend de `logLik()`) |

Une fonction comme `aov` ou `lm` produit donc une liste dont les différents éléments correspondent aux résultats de l'analyse. Si l'on reprend l'exemple de l'analyse de variance sur les données `InsectSprays`, on peut regarder la structure de l'objet créé par `aov()` :

```
> str(aov.spray, max.level = -1)
List of 13
 - attr(*, "class")= chr [1:2] "aov" "lm"
```

Une autre façon de regarder cette structure est d'afficher les noms des éléments de l'objet :

```
> names(aov.spray)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "contrasts"
[10] "xlevels"       "call"          "terms"
[13] "model"
```

Les éléments puivent ensuite être extraits comme vu précédemment :

[19] Il y a plus de 100 fonctions génériques dans R.

```
> aov.spray$coefficients
(Intercept)      sprayB      sprayC      sprayD
  3.7606784   0.1159530  -2.5158217  -1.5963245
     sprayE      sprayF
 -1.9512174   0.2579388
```

`summary()` crée également une liste, qui dans le cas d'`aov()` se limite à un tableau de tests :

```
> str(summary(aov.spray))
List of 1
 $ :Classes anova  and 'data.frame':    2 obs. of  5 variables:
  ..$ Df     : num [1:2] 5 66
  ..$ Sum Sq : num [1:2] 88.4 26.1
  ..$ Mean Sq: num [1:2] 17.688  0.395
  ..$ F value: num [1:2] 44.8   NA
  ..$ Pr(>F) : num [1:2] 0 NA
 - attr(*, "class")= chr [1:2] "summary.aov" "listof"
> names(summary(aov.spray))
NULL
```

Les fonctions génériques sont aussi appelées des *méthodes*. De façon schématique, elles sont contruites comme *`method.foo`*, où *`foo`* désigne la fonction d'analyse. Dans le cas de `summary`, on peut afficher les fonctions qui appliquent cette méthode :

```
> apropos("^summary")
 [1] "summary"               "summary.aov"
 [3] "summary.aovlist"       "summary.connection"
 [5] "summary.data.frame"    "summary.default"
 [7] "summary.factor"        "summary.glm"
 [9] "summary.glm.null"      "summary.infl"
[11] "summary.lm"            "summary.lm.null"
[13] "summary.manova"        "summary.matrix"
[15] "summary.mlm"           "summary.packageStatus"
[17] "summary.POSIXct"       "summary.POSIXlt"
[19] "summary.table"
```

On peut visualiser les particularités de cette méthode dans le cas de la régression linéaire par rapport à l'analyse de variance avec un petit exemple simulé :

```
> x <- y <- rnorm(5);
> mod <- lm(y ~ x)
> names(mod)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
> names(summary(mod))
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "sigma"         "df"
 [7] "r.squared"     "adj.r.squared" "fstatistic"
[10] "cov.unscaled"
```

Les objets produits par `aov()`, `lm()`, `summary()`, ... sont des listes mais ils ne sont pas affichés comme les listes que nous avons vues dans la paragraphe relatif à ce type d'objet. En effet,

il existe des méthodes `print` (en rappelant que de taper le nom en guise de commande équivaut à utiliser `print()`):

```
> apropos("^print")
 [1] "print.pairwise.htest"   "print.power.htest"
 [3] "print"                  "print.anova"
 [5] "print.aov"              "print.aovlist"
 [7] "print.atomic"           "print.by"
 [9] "print.coefmat"          "print.connection"
[11] "print.data.frame"       "print.default"
[13] "print.density"          "print.difftime"
[15] "print.dummy.coef"       "print.dummy.coef.list"
[17] "print.factor"           "print.family"
[19] "print.formula"          "print.ftable"
[21] "print.glm"              "print.glm.null"
[23] "print.hsearch"          "print.htest"
[25] "print.infl"             "print.integrate"
[27] "print.libraryIQR"       "print.listof"
[29] "print.lm"               "print.lm.null"
[31] "print.logLik"           "print.matrix"
[33] "print.mtable"           "print.noquote"
[35] "print.octmode"          "print.ordered"
[37] "print.packageIQR"       "print.packageStatus"
[39] "print.POSIXct"          "print.POSIXlt"
[41] "print.recordedplot"     "print.rle"
[43] "print.SavedPlots"       "print.simple.list"
[45] "print.socket"           "print.summary.aov"
[47] "print.summary.aovlist"  "print.summary.glm"
[49] "print.summary.glm.null" "print.summary.lm"
[51] "print.summary.lm.null"  "print.summary.manova"
[53] "print.summary.table"    "print.table"
[55] "print.tables.aov"       "print.terms"
[57] "print.ts"               "print.xtabs"
```

Toutes ces méthodes `print` permettent bien évidemment un affichage adapté à chaque analyse.

Le tableau suivant indique certaines fonctions génériques qui font des analyses supplémentaires à partir d'un objet qui résulte d'une analyse faite au préalable, l'argument principal étant cet objet, mais dans certains un argument supplémentaire est nécessaire comme pour `predict` ou `update`.

| | |
|---|---|
| `add1` | teste successivement tous les termes qui peuvent être ajoutés à un modèle |
| `drop1` | teste successivement tous les termes qui peuvent être enlevés d'un modèle |
| `step` | sélectionne un modèle par AIC (fait appel à `add1` et `drop1`) |
| `anova` | calcule une table d'analyse de variance ou de déviance pour un ou plusieurs modèles |
| `predict` | calcule les valeurs prédites pour de nouvelles données à partir d'un modèle |
| `update` | ré-ajuste un modèle avec une nouvelle formule ou de nouvelles données |

Il y a également diverses fonctions utilitaires qui extraient des informations d'un objet modèle ou d'une formule, comme `alias()` qui trouve les termes linéairement dépendants dans un modèle linéaire spécifié par une formule.

Enfin, il y a bien sûr les fonctions graphiques comme `plot` qui affiche divers diagnostiques ou `termplot` (*cf.* l'exemple ci-dessus), cette dernière fonction n'est pas vraiment générique mais fait appel à `predict()`.

### 5.4 Les packages

Le tableau suivant liste les packages distribués avec le package base.

| Package | Description |
|---|---|
| ctest | tests classiques (Fisher, 'Student', Wilcoxon, Pearson, Bartlett, Kolmogorov-Smirnov, ...) |
| eda | méthodes décrites dans "Exploratory Data Analysis" de Tukey (seulement ajustement robuste et lissage) |
| lqs | régression "résistante" et estimation de covariance |
| methods | définition des méthodes et classes pour les objets R ainsi que des utilitaires pour la programmation |
| modreg | régression "moderne" (lissage et ajustement local) |
| mva | analyses multivariées |
| nls | régression non-linéaire |
| splines | représentations polynomiales |
| stepfun | analyse de fonctions de distributions empiriques |
| tcltk | fonctions pour utiliser les éléments de l'interface graphique de Tcl/Tk |
| tools | utilitaires pour le développement de package et l'administration |
| ts | analyse de séries temporelles |

À l'exception de ctest qui est chargé au démarrage de R, chaque package est utilisable après l'avoir chargé en mémoire :

```
> library(eda)
```

La liste des fonctions d'un package peut être affichée avec :

```
> library(help=eda)
```

ou en parcourant l'aide au format html. Les informations relatives à chaque fonction peuvent être accédées comme vu précédemment (p. 7).

De nombreux packages *contribués* allongent la liste des analyses possibles avec R. Ils sont distribués séparément, et doivent être installés et chargés en mémoire sous R. Une liste complète de ces packages contribués, accompagnée d'une description, se trouve sur le site Web du CRAN[20]. Certains de ces packages sont regroupés parmi les packages *recommandés* car ils couvrent des méthodes souvent utilsées en analyse des données. (Sous Windows, ces packages recommandés sont distribués avec l'installation de base dans le fichier SetupR.exe.) Ces packages recommandés sont décrits dans le tableau ci-dessous.

[20] http://cran.r-project.org/src/contrib/PACKAGES.html

| Package | Description |
|---|---|
| boot | méthodes de ré-échantillonnage et de bootstrap |
| class | méthodes de classification |
| cluster | méthodes d'aggrégation |
| foreign | fonctions pour importer des données enregistrés sous divers formats (S3, Stata, SAS, Minitab, SPSS, Epi Info) |
| KernSmooth | méthodes pour le calcul de fonctions de densité (y compris bivariées) |
| MASS | contient de nombreuses fonctions, utilitaires et jeux de données accompagnant le livre "Modern Applied Statistics with S-PLUS" par Venables & Ripley |
| mgcv | modèles additifs généralisés |
| nlme | modèles linéaires ou non-linéaires à effets mixtes |
| nnet | réseaux neuroniques et modèles log-linéaires multinomiaux |
| rpart | méthodes de partitionnement récursif |
| spatial | analyses spatiales ("kriging", covariance spatiale, ...) |
| survival | analyses de survie |

La procédure pour installer un package dépend du système d'exploitation et si vous avez installé R à partir des sources ou des exécutables pré-compilés. Dans ce dernier cas, il est recommandé d'utiliser les packages pré-compilés disponibles sur le site du CRAN. Sous Windows, l'exécutable Rgui.exe a un menu "Packages" qui permet d'installer un ou plusieurs packages via internet à partir du site Web de CRAN ou des fichiers '.zip' sur le disque local.

Si l'on a compilé R, un package pourra être installé à partir de ses sources qui sont distribuées sous forme de fichiers '.tar.gz'. Par exemple, si l'on veut installer le package gee, on téléchargera dans un permier temps le fichier gee_4.13-6.tar.gz (le numéro 4.13-6 désigne la version du package ; en général une seule version est disponible sur CRAN). On tapera ensuite à partir du système (et non pas de R) la commande :

```
R INSTALL gee_4.13-6.tar.gz
```

Il y a plusieurs fonctions utiles pour gérer les packages comme `installed.packages()`, `CRAN.packages()` ou `download.packages()`. Il est utile également de taper régulièrement la commande :

```
> update.packages()
```

qui vérifie les versions des packages installés en comparaison à celles disponibles sur CRAN (cette commande peut être appelée du menu "Packages" sous Windows). L'utilisateur peut ensuite mettre à jour les packages qui ont des versions plus récentes que celles installées sur son système.

## 6 Programmer avec R en pratique

Maintenant que nous avons fait un tour d'ensemble des fonctionnalités de R, revenons au langage et à la programmation. Nous allons voir des idées très simples susceptibles d'être mises en pratique aisément.

### 6.1 Boucles et vectorisation

Le point fort de R par rapport à un logiciel à menus déroulants est dans la possibilité de programmer, de façon simple, une suite d'analyses qui seront exécutées successivement. Cette possibilité est propre à tout langage informatique, mais R possède des particularités qui rendent la programmation accessible à des non-spécialistes.

Comme les autres langages, R possède des *structures de contrôle* qui ne sont pas sans rappeler celles du langage C. Supposons qu'on a un vecteur `x`, et pour les éléments de `x` qui ont la valeur `b`, on va donner la valeur 0 à une autre variable `y`, sinon 1. On crée d'abord un vecteur `y` de même longueur que `x` :

```
y <- numeric(length(x))
for (i in 1:length(x)) if (x[i] == b) y[i] <- 0 else y[i] <- 1
```

On peut faire exécuter plusieurs instructions si elles sont encadrées dans des accolades :

```
for (i in 1:length(x)) {
    y[i] <- 0
    ...
}

if (x[i] == b) {
    y[i] <- 0
    ...
}
```

Une autre situation possible est de vouloir faire exécuter une instruction tant qu'une condition est vraie :

```
while (myfun > minimum) {
    ...
}
```

Les boucles et structures de contrôle peuvent cependant être évitées dans la plupart des situations et ce grâce à une caractéristique du langage R : la *vectorisation*. La structure vectorielle rend les boucles implicites dans les expressions et nous en avons vu plein de cas. Considérons l'addition de deux vecteurs :

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

Cette addition pourrait être écrite avec une boucle comme cela se fait dans la plupart de langages :

```
> z <- numeric(length(x))
> for (i in 1:length(z)) z[i] <- x[i] + y[i]
```

Dans ce cas il est nécessaire de créer le vecteur `z` au préalable à cause de l'utilisation de l'indexation. On réalise que cette boucle explicite ne fonctionnera que si `x` et `y` sont de même longueur : elle devra être modifiée si cela n'est pas le cas, alors que la première expression marchera quelque soit la situation.

Les exécutions conditionnelles (`if ... else`) peuvent être évitées avec l'indexation logique ; en reprenant l'exemple plus haut :

```
> y[x == b] <- 0
> y[x != b] <- 1
```

Il y a également les fonctions du type "`apply`" qui évitent d'écrire des boucles. `apply()` agit sur les lignes et/ou les colonnes d'une matrice, sa syntaxe est `apply(X, MARGIN, FUN, ...)`, où `X` est la matrice, `MARGIN` indique si l'action doit être appliquée sur les lignes (`1`), les colonnes (`2`) ou les deux (`c(1, 2)`), `FUN` est la fonction (ou l'opérateur mais dans ce cas il doit être spécifié entre guillemets doubles) qui sera utilisée, et `...` sont d'éventuels arguments supplémentaires pour `FUN`. Un exemple simple suit.

```
> x <- rnorm(10, -5, 0.1)
> y <- rnorm(10, 5, 2)
> X <- cbind(x, y)   # les colonnes de X gardent les noms "x" et "y"
> apply(X, 2, mean)
        x         y
-4.975132  4.932979
> apply(X, 2, sd)
        x         y
0.0755153 2.1388071
```

`lapply()` va agir sur une liste : la syntaxe est similaire à celle d'`apply` et le résultat retourné est une liste.

```
> forms <- list(y ~ x, y ~ poly(x, 2))
> lapply(forms, lm)
[[1]]

Call:
FUN(formula = X[[1]])

Coefficients:
(Intercept)           x
     31.683        5.377


[[2]]

Call:
FUN(formula = X[[2]])

Coefficients:
(Intercept)  poly(x, 2)1  poly(x, 2)2
     4.9330       1.2181      -0.6037
```

`sapply()` est une variante plus flexible de `lapply()` qui peut prendre un vecteur ou une matrice en argument principal, et retourne ses résultats sous une forme plus conviviale, en général sous forme de tableau.

### 6.2 Écrire un programme en R

Typiquement, un programme en R sera écrit dans un fichier sauvé au format ASCII et avec l'extension '.R'. La situation typique où un programme se révèle utile est lorsque l'on veut exécuter plusieurs fois une tâche identique. Dans notre premier exemple, on veut tracer le même graphe pour trois espèces d'oiseaux différentes, les données se trouvant dans trois fichiers distincts. Nous allons procéder pas-à-pas en voyant différentes façons de construire un programme pour ce problème très simple.

D'abord, construisons notre programme de la façon la plus intuitive en faisant exécuter successivement les différentes commandes désirées, en prenant soin au préalable de partitionner le graphique.

```
layout(matrix(1:3, 3, 1))              # partitionne le graphique
data <- read.table("Swal.dat")         # lit les données
plot(data$V1, data$V2, type="l")
```

```
title("swallow")                          # ajoute le titre
data <- read.table("Wren.dat")
plot(data$V1, data$V2, type="l")
title("wren")
data <- read.table("Dunn.dat")
plot(data$V1, data$V2, type="l")
title("dunnock")
```

Le caractère '#' sert à ajouter des commentaires dans le programme, R passe alors à la ligne suivante.

Le problème de ce premier programme est qu'il risque de s'allonger sérieusement si l'on veut ajouter d'autres espèces. De plus, certaines commandes sont répétées plusieurs fois, elles peuvent être regroupées et exécutées en modifiant les arguments qui changent. Les noms de fichier et d'espèce sont donc utilisés comme des variables. La stratégie utilisée ici est de mettre ces noms dans des vecteurs de mode caractère, et d'utiliser ensuite l'indexation pour accéder à leurs différentes valeurs.

```
layout(matrix(1:3, 3, 1))                 # partitionne le graphique
species <- c("swallow", "wren", "dunnock")
file <- c("Swal.dat" , "Wren.dat", "Dunn.dat")
for(i in 1:length(species)) {
    data <- read.table(file[i])           # lit les données
    plot(data$V1, data$V2, type="l")
    title(species[i])                     # ajoute le titre
}
```

On notera qu'il n'y a pas de guillemets autour de `file[i]` dans `read.table()` puisque cet argument est de mode caractère.

Notre programme est maintenant plus compact. Il est plus facile d'ajouter d'autres espèces car les deux vecteurs qui contiennent les noms d'espèces et de fichiers sont définis au début du programme.

Les programmes ci-dessus pourront marcher si les fichiers '.dat' sont placés dans le répertoire de travail de R, sinon il faut soit changer ce répertoire de travail, ou bien spécifier le chemin d'accés dans le programme (par exemple : `file <- "C:/data/Swal.dat"`). Si les instructions sont écrites dans un fichier Mybirds.R, on peut appeler le programme en tapant :

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

Comme pour toute lecture dans un fichier, il est nécessaire de préciser le chemin d'accès au fichier s'il n'est pas dans le répertoire de travail.

## 6.3 Écrire ses fonctions

On a vu que l'essentiel du travail de R se fait à l'aide de fonctions dont les arguments sont indiqués entre parenthèses. L'utilisateur peut écrire ses propres fonctions qui auront les mêmes propriétés que les autres fonctions de R.

Écrire ses propres fonctions permet une utilisation efficace, flexible et rationnelle de R. Reprenons l'exemple ci-dessus de la lecture de données dans un fichier suivi d'un graphe. Si l'on veut répéter cette opération quand on le veut, il peut être judicieux d'écrire une fonction :

```
myfun <- function(S, F)
{
    data <- read.table(F)
    plot(data$V1, data$V2, type="l")
```

```
    title(S)
}
```

Pour pouvoir être exécutée, cette fonction doit être chargée en mémoire ce qui peut se faire de plusieurs façons. On peut entrer les lignes de la fonction au clavier comme n'importe quelle commande, ou les 'copier/coller' à partir d'un éditeur. Si la fonction a été enregistrée dans un fichier ASCII, on peut la charger avec `source()` comme un autre programme. Si l'utilisateur veut que ses fonctions soient chargées au démarrage de R, il peut les enregistrer dans un workspace `.RData` qui sera chargé en mémoire s'il est localisé dans le répertoire de travail de démarrage. Une autre possibilté est de configurer le fichier '.Rprofile' ou 'Rprofile' (voir `?Startup` pour les détails). Enfin, il est possible de créer un package mais ceci ne sera pas abordé ici (on se reportera au manuel "Writing R Extensions").

On pourra par la suite, par une seule commande, lire les données et dessiner le graphe, par exemple `myfun("swallow", "Swal.dat")`. Nous arrivons donc à une troisième version de notre programme :

```
layout(matrix(1:3, 3, 1))
myfun("swallow", "Swal.dat")
myfun("wren", "Wrenn.dat")
myfun("dunnock", "Dunn.dat")
```

On peut également utiliser `sapply()` aboutissant à une quatrième version du programme :

```
layout(matrix(1:3, 3, 1))
species <- c("swallow", "wren", "dunnock")
file <- c("Swal.dat" , "Wren.dat", "Dunn.dat")
sapply(species, myfun, file)
```

Avec R, il n'est pas nécessaire de déclarer les variables qui sont utilisées dans une fonction (au contraire des langages comme C ou Fortran). Quand une fonction est exécutée, R utilise une règle nommée "étendue lexiquale" (*lexical scoping*) pour décider si un objet désigne une variable locale à la fonction ou un objet global. Pour comprendre ce mécanisme, considérons la fonction très simple ci-dessous :

```
> foo <- function() print(x)
> x <- 1
> foo()
[1] 1
```

Le nom `x` n'a pas été utilisée au sein de `foo()`, R va donc chercher dans l'environnement *immédiatement* supérieur si un objet nommé `x` existe et affichera sa valeur (sinon un message d'erreur est affiché et l'exécution est terminée).

Si l'on utilise `x` comme nom d'objet au sein de notre fonction, la valeur de `x` dans l'environnement global n'est pas modifiée.

```
> x <- 1
> foo2 <- function() { x <- 2; print(x) }
> foo2()
[1] 2
> x
[1] 1
```

Cette fois `print()` a utilisé l'objet `x` qui a été défini dans son environnement, c'est-à-dire celui de la fonction `foo2`.

Le mot "*immédiatement*" ci-dessus est important. Dans les deux exemples que nous venons de voir, il y a *deux* environnements : le global et celui de la fonction `foo` ou `foo2`. S'il y avait trois

ou plus environnements emboîtés, la recherche des objets se fait par “paliers” d’un environnement à l’environnement immédiatement supérieur, ainsi de suite jusqu’à l’environnement global.

Il y a deux façons de spécifier les arguments à une fonction : par leurs positions ou par leurs noms. Par exemple, considérons une fonction qui prendrait trois arguments :

```
foo <- function(arg1, arg2, arg3) {...}
```

On peut exécuter `foo()` sans utiliser les noms `arg1`, ..., si les objets correspondants sont placés dans l’ordre, par exemple : `foo(x, y, z)`. Par contre, l’ordre n’a pas d’importance si les noms des arguments sont utilisés, par exemple : `foo(arg3 = z, arg2 = y, arg1 = x)`. Une autre particularité des fonctions dans R est la possibilité d’utiliser des valeurs par défaut dans la définition. Par exemple :

```
foo <- function(arg1, arg2 = 5, arg3 = FALSE) {...}
```

Les deux commandes `foo(x)` et `foo(x, 5, FALSE)` auront exactement le même résultat. L’utilisation de valeurs par défaut dans la définition d’une fonction est bien sûr très pratique et ajoute à la flexibilité du système.

Un autre exemple de fonction n’est pas purement statistique mais illustre bien la grande flexibilité de R. Considérons que l’on veuille étudier le comportement d’un modèle non-linéaire : le modèle de Ricker défini par :

$$N_{t+1} = N_t \exp\left[r\left(1 - \frac{N_t}{K}\right)\right]$$

Ce modèle est très utilisé en dynamique des populations, en particulier de poissons. On voudra à l’aide d’une fonction simuler ce modèle en fonction du taux de croissance $r$ et de l’effectif initial de la population $N_0$ (la capacité du milieu $K$ est couramment prise égale à 1 et cette valeur sera prise par défaut) ; les résultats seront affichés sous forme de graphique montrant les changements d’effectifs au cours du temps. On ajoutera une option qui permettra de réduire l’affichage des résultats aux dernières générations (par défaut tous les résultats seront affichés). La fonction ci-dessous permet de faire cette analyse numérique du modèle de Ricker.

```
ricker <- function(nzero, r, K=1, time=100, from=0, to=time)
{
    N <- numeric(time+1)
    N[1] <- nzero
    for (i in 1:time) N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
    Time <- 0:time
    plot(Time, N, type="l", xlim=c(from, to))
}
```

Essayez vous-mêmes avec :

```
> layout(matrix(1:3, 3, 1))
> ricker(0.1, 1); title("r = 1")
> ricker(0.1, 2); title("r = 2")
> ricker(0.1, 3); title("r = 3")
```

## 7 Littérature sur R

**Manuels.** Plusieurs manuels sont distribués avec R dans le répertoire R_HOME/doc/manual/ (R_HOME désignant le chemin où R est installé) :

- “An Introduction to R” [R-intro.pdf],
- “R Installation and Administration” [R-admin.pdf],

– “R Data Import/Export” [R-data.pdf],
– “Writing R Extensions” [R-exts.pdf],
– “R Language Definition” [R-lang.pdf].

Les fichiers correspondants peuvent être dans divers formats (pdf, html, texi, . . .) en fonction du type d’installation.

**FAQ.** R est également distribué avec un FAQ (*Frequently Asked Questions*) localisé dans le répertoire R_HOME/doc/html/. Une version de ce R-FAQ est régulièrement mise à jour sur le site Web du CRAN : http://cran.r-project.org/doc/FAQ/R-FAQ.html.

**Ressources en-ligne.** Le site Web du CRAN ainsi que la home-page de R accueille plusieurs documents et ressources bibliographiques ainsi que des liens vers d’autres sites. On peut y trouver une liste de publications (livres et articles) liées à R ou aux méthodes statistiques[21], et des documents et manuels écrits par des utilisateurs de R[22].

**Listes de discussion.** Il existe trois listes de discussion électroniques sur R ; pour s’inscrire, envoyer un message ou consulter les archives voir : http://www.R-project.org/mail.html.

La liste de discussion générale ‘r-help’ est une source intéressante d’information pour les utilisateurs (les deux autres listes sont consacrées aux annonces de nouvelles versions, nouveaux packages, . . ., et aux développeurs). De nombreux utilisateurs ont envoyé sur ‘r-help’ des fonctions ou des programmes qui peuvent donc être trouvés dans les archives. Il est donc important si l’on a un problème avec R de procéder dans l’ordre avant d’envoyer un message à ‘r-help’ et de :

1. consulter attentivement l’aide-en-ligne (éventuellement avec le moteur de recherche),
2. consulter le R-FAQ,
3. chercher dans les archives de ‘r-help’ à l’adresse ci-dessus ou en consultant un des moteurs de recherche mis en place sur certains sites Web [23].

**R News.** La revue électronique *R News* a pour but de combler l’espace entre les listes de discussion électroniques et les publications scientifiques traditionnelles. Le premier numéro a été publié en janvier 2001 et le rythme de sortie est de trois numéros par an. Kurt Hornik et Friedrich Leisch sont les éditeurs[24].

**Citer R dans une publication.** Enfin, si vous mentionnez R dans une publication, il faut citer l’article original :

> Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5 : 299–314.

[21] http://www.R-project.org/doc/bib/R-publications.html
[22] http://cran.r-project.org/other-docs.html
[23] Les adresses de ces sites sont répertoriées sur celui du CRAN à http://cran.r-project.org/search.html
[24] http://cran.r-project.org/doc/Rnews/