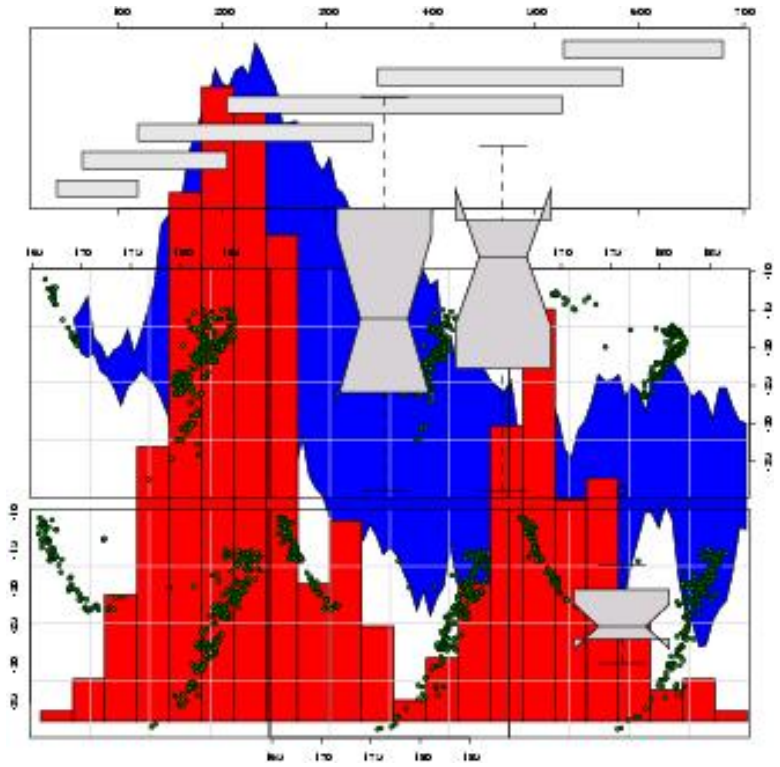
INITIATION A L'ENVIRONNEMENT R

UNIVERSITE PAUL SABATIER TOULOUSE III

Jérôme HUILLET



10 Septembre 2002

Table des matières

Première partie

Documentation

Chapitre 1

Présentation

1.1 Présentation de R

1.1.1 Généralités

R est un système qui est communément appelé langage et logiciel, il permet de réaliser des analyses statistiques.
Plus particulièrement, il comporte des moyens qui rendent possibles la manipulation des données, les calculs et les représentations graphiques. **R** a aussi la possibilité d'exécuter des programmes stockés dans des fichiers textes.

En effet **R** possède :

- un système efficace de manipulation et de stockage des données
- différents opérateurs pour le calcul sur tableaux, en particulier les matrices
- un grand nombre d'outils pour l'analyse des données et les méthodes statistiques
- des moyens graphiques pour visualiser les analyses
- un langage de programmation simple et performant comportant : conditions, boucles, moyens d'entrées sorties, possibilité de définir des fonctions récursives.

La conception de **R** a été fortement influencée par deux langages :

- **S** qui est un langage développé par les *AT&T Bell Laboratories* et plus particulièrement par Rick Becker, John Chambers et Allan Wilks. **S** est un langage de haut niveau et est un environnement pour l'analyse des données et les représentations graphiques.
  S est utilisable à travers le logiciel **S-Plus** qui est commercialisé par la société *Insightful* (http ://www.splus.com/). **S-Plus** est un des logiciels de statistiques les plus populaires et il s'est imposé comme une référence dans le milieu statistique.
- **Scheme** de Sussman (http ://www.schemers.org/) est un langage fonctionnel, le principe fondamental de ce langage est la *récursivité*.

R, le langage obtenu est très semblable à **S**, l'exécution et la sémantique sont dérivées de **Scheme**.

Le noyau de **R** est écrit en langage machine interprété qui a une syntaxe similaire au langage **C**, mais qui est réellement un langage de programmation avec des capacités identique au langage **Scheme**.
La plupart des fonctions accessibles, par l'utilisateur dans **R**, sont écrites en **R**. (Le système est lui-même écrit en **R**). Pour les tâches intensives les langages **C**, **C++** et **Fortran** ont été utilisés et liés pour une meilleure efficacité.
R permet aux utilisateurs d'accroître les possibilités du logiciel en créant de nouvelles fonctions. Les utilisateurs expérimentés peuvent écrire du code en **C** pour manipuler directement des objets **R**.

R comporte un grand nombre de procédures statistiques.
Parmi elles, nous avons : les modèles linéaires, les modèles linéaires généralisés, la régression non-linéaire, les séries chronologiques, les tests paramétriques et non paramétriques classiques, ...
Il y a également un grand nombre de fonctions fournissant un environnement graphique flexible afin de visualiser et créer divers genres de présentations de données.

Les utilisateurs pensent souvent que **R** est un système de statistique. Les concepteurs et développeurs préfèrent dire que c'est un environnement dans lequel des techniques statistiques sont exécutées. **R** peut étendre ses fonctions par l'intermédiaire de modules. Il existe à l'heure actuelle douze modules fournis lorsque **R** est distribué (sous *Unix*) et d'autres sont disponibles par l'intermédiaire du **CRAN**. Ils sont disponibles pour des buts spécifiques et présentent une large gamme de statistiques modernes. (analyse descriptive des données multidimensionnelles, arbres de régression et de classification, graphiques en trois dimensions, etc...)

R est développé pour pouvoir être utilisé avec les systèmes d'exploitation *Unix*, *GNU/Linux*, *Windows* et *MacOS*.

R possède un site officiel à l'adresse http ://www.R-project.org/, c'est un logiciel libre qui est distribué sous les termes de la **« GNU Public Licence »** (règle du copyleft) et il fait partie intégrante du **projet GNU**.

1.1.2 Les auteurs

R a été initialement créé par Robert Gentleman et Ross Ihaka du département de statistique de l'Université d'Auckland en Nouvelle Zélande.
Depuis 1997, il s'est formé une équipe (la « R Core Team ») qui développe **R**, elle est constituée de :

```
Douglas Bates        <bates@stat.wisc.edu>
John Chambers        <jmc@research.bell-labs.com>
Peter Dalgaard       <p.dalgaard@kubism.ku.dk>
Robert Gentleman     <rgentlem@jimmy.dfci.harvard.edu>
Kurt Hornik          <Kurt.Hornik@ci.tuwien.ac.at>
Stefano Iacus        <stefano.iacus@unimi.it>
Ross Ihaka           <ihaka@stat.auckland.ac.nz>
```

```
Friedrich Leisch     <Friedrich.Leisch@ci.tuwien.ac.at>
Thomas Lumley        <tlumley@u.washington.edu>
Martin Maechler      <maechler@stat.math.ethz.ch>
Guido Masarotto      <guido@hal.stat.unipd.it>
Paul Murrell         <paul@stat.auckland.ac.nz>
Brian Ripley         <ripley@stats.ox.ac.uk>
Duncan Temple Lang   <duncan@research.bell-labs.com>
Luke Tierney         <luke@stat.umn.edu>

plus Heiner Schwarte <h.schwarte@bluewin.ch> jusqu'à octobre 1999.
```

1.1.3 Le CRAN

Le « Comprehensive R Archive Network »(**CRAN**) est un ensemble de sites qui fournit ce qui est nécessaire à la distribution de **R**, ses extensions, sa documentation, ses fichiers sources et ses fichiers binaires.
Le **CRAN** est similaire à CPAN pour le langage **Perl** ou CTAN pour **TEX/LATEX**.

Le site maître du CRAN est situé en Autriche à Vienne, nous pouvons y accéder par l'URL :

```
http://cran.r-project.org/
```

Les sites miroirs sont à l'heure actuelle :

```
http://cran.at.r-project.org/
                  (TU Wien, Austria)
http://cran.au.r-project.org/
                  (PlanetMirror, Australia)
http://cran.br.r-project.org/
                  (Universidade Federal de Paraná, Brazil)
http://cran.ch.r-project.org/
                  (ETH Zürich, Switzerland)
http://cran.de.r-project.org/
                  (APP, Germany)
http://cran.dk.r-project.org/
                  (SunSITE, Denmark)
http://cran.hu.r-project.org/
                  (Semmelweis U, Hungary)
http://cran.uk.r-project.org/
                  (U of Bristol, United Kingdom)
http://cran.us.r-project.org/
                  (U of Wisconsin, USA)
http://cran.za.r-project.org/
                  (Rhodes U, South Africa)
```

Pour télécharger les applications du projet **R**, il est conseillé d'accéder au site miroir le plus proche géographiquement de l'endroit de votre connexion.
Après une lecture des adresses des différents sites miroirs, nous remarquons que **R** comporte une forte répartition européenne.

1.1.4 Le point fort de R

Ce logiciel étant de domaine public son point fort est représenté par le développement d'applications, de modules qui sont mis à la disposition de tous les utilisateurs et développeurs.
Son réseau international de développement est en perpétuelle évolution.
L'intérêt majeur de **R** est qu'il est ouvert à tous. De ce fait, tout le monde peut « apporter sa pierre à l'édifice ».
Cela laisse envisager de phénoménales extensions au système. Le potentiel de **R** semble donc énorme.

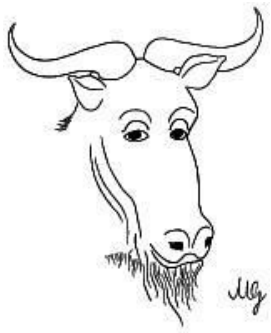
1.2 La philosophie des logiciels libres : le projet GNU

R est souvent présenté comme un clone de **S**, en effet **R** étant développé par le **« GNU-project »**, il est défini par certains comme **« GNU-S »**.
Pour cela, il nous semble intéressant de présenter **le projet GNU** ainsi que **la philosophie des logiciels libres**.

1.2.1 Le projet

Le projet GNU a été lancé en 1984 afin de développer un système d'exploitation complet, semblable à Unix et qui soit un logiciel libre : **le système GNU** (On le prononce « gnou »avec un « G »audible). Des variantes du système d'exploitation GNU, basées sur le noyau *« Linux »*, sont largement utilisées à présent bien que ces systèmes soient communément appelés par le terme *« Linux »*. Ils le seraient plus exactement par *« GNU/Linux »*.
Les logiciels libres disponibles complètent le système.

1.2.2 La philosophie



L'expression « **Logiciel libre** » fait référence à la liberté et non pas au prix. Pour comprendre le concept, vous devez penser à la « **liberté d'expression** », pas à « l'entrée libre ».

L'expression « Logiciel libre »fait référence à la liberté pour les utilisateurs d'exécuter, de copier, de distribuer, d'étudier, de modifier et d'améliorer le logiciel.

Plus précisément, elle fait référence à quatre types de liberté pour l'utilisateur du logiciel :

- La liberté d'exécuter le programme, pour tous les usages *(liberté 0)*.
- La liberté d'étudier le fonctionnement du programme, et de l'adapter à vos besoins *(liberté 1)*. Pour ceci l'accès au code source est une condition requise.
- La liberté de redistribuer des copies, donc d'aider votre voisin, *(liberté 2)*.
- La liberté d'améliorer le programme et de publier vos améliorations, pour en faire profiter toute la communauté *(liberté 3)*. Pour ceci l'accès au code source est une condition requise.

Un programme est un logiciel libre si les utilisateurs ont toutes ces libertés. Ainsi, il est libre de redistribuer des copies, avec ou sans modification, gratuitement ou non, à tout le monde, partout. Etre libre de faire ceci signifie entre autre que vous n'avez pas à demander ou à payer pour en avoir la permission.

Vous devez aussi avoir **la liberté de faire des modifications** et de les utiliser à titre personnel dans votre travail ou vos loisirs, sans en mentionner l'existence. Si vous publiez vos modifications, vous n'êtes pas obligé de prévenir quelqu'un de particulier ou de le faire d'une manière particulière.

La liberté d'utiliser un programme est la liberté pour tout type de personne ou d'organisation de l'utiliser pour tout type de système informatique, pour tout type de tâche et sans être obligé de communiquer ultérieurement avec le développeur ou tout autre entité spécifique.

La liberté de redistribuer des copies doit inclure les formes binaires ou exécutables du programme (tout comme le code source) à la fois pour les versions modifiées ou non modifiées du programme.

Pour avoir la liberté d'effectuer des modifications et de publier des versions améliorées, vous devez avoir l'accès au code source du programme. Par conséquent, l'accessibilité du code source est une condition requise pour un logiciel libre.

Pour que ces libertés soient réelles, elles doivent être irrévocables tant que vous n'avez rien fait de mal. Si le développeur du logiciel a le droit de révoquer la licence sans que vous n'ayez fait quoi que ce soit pour le justifier, le logiciel n'est pas libre. Cependant, certains types de règles sur la manière de distribuer le logiciel libre sont acceptables tant que ces règles ne rentrent pas en conflit avec les libertés fondamentales.
Par exemple, le « copyleft »(pour résumer très simplement) est une règle qui établit que lorsque vous redistribuez les programmes, vous ne pouvez ajouter de restrictions pour retirer les libertés fondamentales au public. Cette règle ne rentre pas en conflit avec les libertés fondamentales ; en fait, elle les protège.

Ainsi, vous pouvez avoir à payer pour obtenir une copie d'un logiciel du projet GNU ou vous pouvez l'obtenir gratuitement. Mais indifféremment de la manière dont vous vous l'êtes procuré, vous avez toujours la liberté de copier et de modifier un logiciel et même d'en vendre des copies.
« Logiciel libre »ne signifie pas « non commercial ». Un logiciel libre doit être disponible pour un usage commercial. Le développement commercial de logiciel libre n'est plus l'exception.

Les règles sur la manière d'empaqueter une version modifiée sont acceptables si elles n'entravent pas votre liberté de la publier. Les règles disant « si vous publiez le programme par ce moyen, vous devez le faire par ce moyen aussi »sont acceptables aux mêmes conditions (notez que de telles règles doivent vous laisser le choix de publier ou non le programme).
Dans le projet GNU, il est utilisé le **« copyleft »** pour protéger ces libertés. Mais des logiciels libres « non-copyleftés »c'est-à-dire qui ne sont pas protégés par la règle existent aussi. Nous croyons qu'il y a de bonnes raisons qui font qu'il est mieux d'utiliser le « copyleft », mais si votre programme est libre « non-copylefté », nous pouvons tout de même l'utiliser.

Quand vous parlez des logiciels libres, il est préférable de ne pas utiliser de termes comme « donner »ou « gratuit », car ils laissent supposer que la finalité des logiciels libres est la gratuité et non la liberté.

Pour résumer, un logiciel libre est un logiciel qui est fourni avec l'autorisation pour quiconque de l'utiliser, de le copier, et de le distribuer, soit sous une forme conforme à l'original, soit avec des modifications, ou encore gratuitement ou contre un certain montant. Ceci signifie en particulier que son code source doit être disponible. « S'il n'y a pas de sources, ce n'est pas du logiciel. »

Pour en savoir plus sur **le projet et la philosophie GNU**, il est possible de consulter le site officiel (http ://www.gnu.org/) pour lequel l'ensemble des informations précédentes ont été tirées.

Chapitre 2

Installation du logiciel

2.1 Systèmes d'exploitations et R

R a été développé pour les systèmes d'exploitations *Unix*, *Windows* et *MacOS*. Les fichiers disponibles sont précompilés pour certains systèmes (*Windows* et *MacOS*) alors qu'il faut les compiler pour d'autres (*Unix*).

2.2 Obtention de R

Les fichiers sources, binaires et de documentation de **R** peuvent être obtenus par le **CRAN** (Comprehensive R Archive Network), le réseau complet des archives de **R**.

Il est possible d'obtenir les fichiers sources de différentes façons :

- En chargeant et décompressant le fichier de **R** correspondant à la version la plus récente, **R-x.y.z.tgz**. (Actuellement le fichier `R-1.5.0.tgz`, version 1.5.0)

  La marche à suivre est :

  - Lancer un navigateur sur **Internet** (e.g. Netscape ou Internet Explorer)
  - Se connecter au site officiel de **R** (Adresse : http ://www.R-project.org/) ou de préférence sur le miroir **CRAN** le plus proche.
  - Aller sur la page correspondant au lien Download « **CRAN** »
  - Télécharger le fichier **R-x.y.z.tgz** présent sur la page chargée précédemment qui est une archive (tar) compressée avec gzip.
    Ce fichier correspond aux fichiers sources de R.
  - On décompresse et on désarchive le fichier R-x.y.z.tgz grâce à la commande :

    ```
    gzip -dc R-x.y.z.tgz | tar xvfo -
    ```

- Il est possible avec *rsync* de maintenir une version de **R** à jour.
  En utilisant *rsync*, la commande à effectuer est :

  ```
  rsync -rC rsync.r-project.org:: "module" R
  ```

  pour créer une copie de l'arbre des sources indiqué par *« module »* dans le sous-répertoire **R** du répertoire courant.
  (*« module »* correspond à une des quatre possibilités d'installation de fichiers sources de **R**)
  (*'r-release'* correspond à l'installation des fichiers sources de la version courante)
  (*'r-patched'* correspond aux fichiers pour la version corrigée)
  (*'r-devel'* correspond aux fichiers pour la version en cours de développement)
  (*'r-ng'* correspond aux fichiers pour la version future qui est encore instable)
  Les arbres de *rsync* sont créés directement et sont mis à jour d'heure en heure.

  Plus d'information sur *rsync* est disponible à l'adresse : http ://rsync.org/

2.3 Installation

2.3.1 Sous Unix

L'installation sous *Unix* se réalise sans difficultés particulières.
Il est nécessaire de posséder un **compilateur C** et un **compilateur FORTRAN** ou **f2c**. De plus, il est nécessaire de posséder la version 5 de **Perl** pour installer correctement les documentations. (Sinon, il est possible d'obtenir une version pdf du manuel de référence par l'intermédiaire du **CRAN**.)

La marche à suivre pour l'installation est la suivante :

- Lancer les commandes suivantes :

  - `./configure` (Contrôle, préparation de l'environnement)

  - `make` (Compilation et liaison des fichiers sources)

  - `make check` (Contrôle, test de l'installation)

  - `make install` (Réalise l'installation proprement dîte, positionnement des fichiers par exemple)
    C'est généralement, l'utilisateur « root », l'administrateur système qui installe les logiciels et de ce fait, il installe les fichiers correspondants à l'installation dans le répertoire de son choix.
    La commande effectuée précédemment crée de nouveaux répertoires au niveau du chemin '/usr/local' par défaut, si le « root »ne souhaite pas implanter ces nouveaux fichiers à cet endroit (pour la gestion de l'espace disque), il utilise alors les commandes suivantes :

    * `./configure --prefix=/$HOME/chemin/acces`
      Spécification du chemin d'accès pour la configuration

    * `make prefix=/$HOME/chemin/acces install`
      Spécification du chemin d'accès pour l'installation

- Enfin, pour installer les manuels de **R**, on peut utiliser une ou plusieurs commandes selon les versions :

  - `make install-dvi`

  - `make install-info`

  - `make install-pdf`

2.3.2 Sous Windows

Le répertoire 'bin/windows' du site **CRAN** comporte les fichiers binaires de la distribution de base ainsi qu'un grand nombre de modules supplémentaires. Ceci peut fonctionner sous *Windows 95, 98, ME, NT4, 2000* et *XP*.
Le système de fichiers doit accepter les noms de fichiers longs.
En utilisant, `'SetupR.exe'` ou `'miniR.exe'`, il faut cliquer doublement sur l'icône et suivre les instructions. Si on installe **R** de cette façon, on peut le désinstaller par le *menu Start*.

La version *Windows* de **R** a été créée par Robert Gentleman, et est maintenant développée et maintenue par Guido Masarotto et Brian D. Ripley.

Regarder le document *« R for Windows FAQ »* pour plus de détails.

2.3.3 Sous Macintosh

Le répertoire 'bin/macos' du site **CRAN** comporte les archives bin-hexed ('hqx') and stuffit ('sit') pour la distribution de base et un grand nombre de modules qui fonctionnent sous les systèmes allant de *MacOS 8.6* à *MacOS 9.1* ou sous les divers *MacOS X*.
La version *Macintosh* de **R** et les fichiers binaires sont maintenus par Stefano Iacus.
Les fichiers binaires de la distribution de base pour *MacOS X* sont disponibles dans le répertoire 'bin/macosx' du site **CRAN** grâce à Jan de Leeuw.

Regarder le document *« R for Macintosh FAQ »* pour plus de détails.

2.4 Modules supplémentaires et manipulation

L'installation, la mise à jour et la suppression des modules supplémentaires sont détaillées pour *Unix*, en ce qui concerne les systèmes d'exploitation *Windows* et *MacOS*, il faut consulter les documents *« R Windows FAQ »* et *« R Macintosh FAQ »* accessibles par le **CRAN**.

2.4.1 Les modules supplémentaires

La distribution courante de **R** sous *Unix* est actuellement fournie avec des modules supplémentaires qui s'installent en même temps que le logiciel.

Modules	Description
ctest	Ensemble de tests classiques : Chi-squared, Fisher, Student, ...
eda	Analyse exploratoire des données
lqs	Régression robuste et estimation de la covariance
methods	Des méthodes explicitement définies et des outils de programmation
modreg	Méthodes modernes de régression : lissage et méthodes locales
mva	Analyse multivariée
nls	Régression non linéaire
splines	Splines
stepfun	Traitement des fonctions en escalier, y compris les répartition cumulées
tcltk	Interface pour le logiciel Tcl/Tk
tools	Outils pour le développement des modules et l'administration
ts	Séries chronologiques

De plus, un très grand nombre d'autres modules sont disponibles par l'intermédiaire du **CRAN** par internet.

2.4.2 Installation

Les modules supplémentaires du **CRAN** sont compressés et archivés, ils sont nommés « mod_version.tar.gz ».
Si **tar** et **gzip** sont disponibles sur le système utilisé, on effectue la commande *Unix* :

```
R CMD INSTALL /chemin/de/"mod_version.tar.gz"
```

Cette commande est possible pour l'utilisateur « root », un utilisateur quelconque effectuera plutôt la commande :

```
R CMD INSTALL -l "lib" /chemin/de/"mod_version.tar.gz"
```

Ceci pour définir le chemin d'accès au répertoire où sera installé le module correspondant.
La variable d'environnement `R_LIBS` permet de spécifier les différents chemins d'accès aux modules.

Pour spécifier cette variable, on effectue une commande particulière suivant le type de **Shell** utilisé :

- Pour le **C_Shell** et ses dérivés :

```
setenv R_LIBS /chemin/acces/modules
```

- Pour le **Shell de Bourne** et ses dérivés (**Korn Shell,...**) :

```
R_LIBS="/chemin/acces/modules"; export R_LIBS
```

Il est aussi possible d'installer et de mettre à jour les modules à partir de **R**.
(Fonctions : `options(), install.packages(), update.packages()`...)
Pour plus d'information sur ce mode de fonctionnement, consulter le manuel *« R Installation and Administration »* ou la page internet du **CRAN** correspondant aux « packages »(http ://cran.r-project.org/)

2.4.3 Utilisation

- Pour afficher l'ensemble des modules installés dans l'environnement on utilise la commande :

```
library()
```

Exemple de sortie :

```
Packages in library '/$HOME/R/lib':

pls                     Partial Least Squares Regression

Packages in library '/usr/local/lib/R/library':

KernSmooth              Functions for kernel smoothing for Wand &
                        Jones (1995)
MASS                    Main Library of Venables and Ripley's MASS
base                    The R base package
boot                    Bootstrap R (S-Plus) Functions (Canty)
class                   Functions for classification
cluster                 Functions for clustering (by Rousseeuw et al.)
combinat                combinatorics utilities
ctest                   Classical Tests
...
```

- On charge un module installé pour pouvoir l'utiliser avec :

```
library(mod)
```

Exemple :

```
> library(multidim)
```

- Pour obtenir l'ensemble des fonctions d'un module :

```
library(help = mod)
help(package = mod)
```

Exemple :

```
> help(package=foreign)
foreign          Read data stored by Minitab, S, SAS, SPSS, Stata, ...

Description:

Package: foreign
Priority: recommended
Version: 0.5-5
Date: 2002-05-17
Title: Read data stored by Minitab, S, SAS, SPSS, Stata, ...
Depends: R (>= 1.2.0)
Maintainer: R-core <R-core@r-project.org>
Author: Thomas Lumley <thomas@biostat.washington.edu>, Saikat DebRoy
        <saikat@stat.wisc.edu>, Douglas M. Bates <bates@stat.wisc.edu>
        and Duncan Murdoch <murdoch@stats.uwo.ca>
Description: Functions for reading and writing data stored by
        statistical packages such as Minitab, S, SAS, SPSS, Stata, ...
License: GPL version 2 or later
Built: R 1.5.0; mips-sgi-irix6.5; Thu May 30 16:35:19 MET 2002

Index:

lookup.xport             Lookup information on a SAS XPORT format
                         library
S3 read functions        Read an S3 Binary File
read.dta                 Read Stata binary files
read.epiinfo             Read Epi Info data files
read.mtp                 Read a Minitab Portable Worksheet
read.spss                Read an SPSS data file
read.ssd                 obtain a data frame from a SAS permanent
                         dataset, via read.xport
read.xport               Read a SAS XPORT format library
write.dta                Write files in Stata binary format
```

- Pour enlever un module qui a été préalablement chargé :

```
detach("package:mod")
```

Exemple :

```
detach("package:multidim")
```

2.4.4 Suppression

Les modules installés peuvent être supprimés par les commandes :

```
R CMD REMOVE mod_1...mod_n
```

ou

```
R CMD REMOVE -l "lib" mod_1...mod_n
```

Comme pour l'installation et la mise à jour, la suppresion peut être réalisée à partir de **R** (fonction : `remove.packages()`)

2.5 R et Emacs

Il existe un mode spécifique à *Emacs* appelé **ESS** (« Emacs Speacks Statistics ») qui fournit une **« bonne »interface entre les programmes statistiques et les méthodes statistiques**.

Ce module de *Emacs* est réalisé pour fournir une assistance à la programmation statistique interactive et à l'analyse des données.

Le module interprète plusieurs langages :

- Les dialectes de **S** (**S 3/4**, **S-Plus 3.x/4.x/5.x** et **R**)
- Les dialectes **LispStat (XLispStat, ViSta)**
- **SAS**
- **Stata**
- Les dialectes **SPSS (SPSS, Fiasco)**
- **SCA**

Grâce à **ESS**, **R** est facile d'utilisation, **ESS** permet le retour de commande, le retour du curseur sur une commande et sa correction en cours d'élaboration.
De plus, **ESS** met en couleur certaines parties des commandes mises en œuvre (par exemple les paramètres passés entre guillemets sont en vert, le caractère d'affectation est en rouge, ...).
Il met aussi en forme automatiquement les lignes de code créées (Création d'espacements entre les mots par exemple). Il transforme le caractère d'affectation '_' en '<-' qui est plus lisible et plus facile à interpréter.
L'appel de l'aide de **R** provoque l'ouverture d'une nouvelle fenêtre *Emacs* qui comporte l'aide demandée.
Le bouton de tabulation permet de compléter le nom de fonctions ou d'objets en cours de constitution. Lorsque différentes solutions de complétions sont possibles, elles sont affichées dans une fenêtre annexe identiquement à l'aide de **R** et accessibles grâce à la souris.

Les dernières versions de **ESS** sont disponibles aux adresses :
http ://ess.stat.wisc.edu/pub/ESS/ ou ftp ://ess.stat.wisc.edu/pub/ESS/ ou par le **CRAN**.
La version HTML de la documentation se trouve à l'adresse :
http ://stat.ethz.ch/ESS/.
ESS est fourni avec des instructions d'installation détaillées.
Pour avoir de l'aide à propos de **ESS**, il faut envoyer un courrier électronique à ESS-help@stat.ethz.ch.

Chapitre 3

Notions élémentaires

R ayant été conçu par le « GNU-project »et étant un clone de **S**, un certain nombre de commandes de **R** sont très semblables aux commandes de **S** et des systèmes d'exploitations *Unix* et *GNU/Linux*.
Exemple : Pour afficher les fichiers du répertoire courant sous *Unix* nous utilisons '`ls`' et dans **R** pour les objets cela est '`ls()`'

3.1 Appel de R

On rentre dans **R** en tapant dans la fenêtre *Unix* : `R`
et on le quitte en tapant : `> quit()`, `> q()` ou `Control D`
Une question est posée lorsqu'on veut quitter.

```
Save workspace image? [y/n/c]:
```

- `y` permet de quitter et de sauvegarder le travail effectué
- `n` permet de quitter sans sauvegarder
- `c` permet d'annuler la fermeture de **R**

Le prompt de **R** est par défaut le caractère '`>`', signifie qu'il est en attente d'une commande.
Les objets courants sont sauvés dans le fichier '`.RData`'.

3.2 L'aide de R

Il est possible d'obtenir de l'aide sous différents formats (html et texte).
R nous propose une aide en ligne en tapant la commande :

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

L' aide pour une commande particulière est possible en tapant :

```
> ?nom-commande
```

Les commandes :

```
> help(nom-commande)
> ?"nom-commande"
> help("nom-commande")
```

donnent le même résultat avec la possibilité pour les deux dernières d'obtenir l'aide pour les caractères spéciaux.

Exemple :

```
> ?&
Error: syntax error
> ?"&"
Logic                   package:base                   R Documentation

Logical Operators
... (suite de l'aide)
```

De plus, il est possible de réaliser une recherche de fonctions à l'aide de mots-clés grâce à la fonction `apropos()`.

```
>  apropos(mean)
[1] "mean"           "mean.POSIXct"   "mean.POSIXlt"   "mean.data.frame"
[5] "mean.default"   "weighted.mean"
```

3.3 Commandes utiles

Il n'est pas nécessaire de sortir de **R** pour exécuter une commande *Unix* (Utilisation de la commande `system()`) :

```
> system("ls")
Docs     Instal   Mail     MySwork  R        Tex      nsmail
```

Il est aussi possible d'obtenir le nom du répertoire de travail courant avec la commande :

```
> getwd()
[1] "/$HOME/R"     (par exemple)
```

3.4 Syntaxe générale

- **R** fait la différence entre les majuscules et les minuscules. (Ainsi '`A`' et '`a`' représentent deux objets différents)
- On sépare les différentes commandes par des points virgules '`;`' ou par des sauts de ligne.

  ```
  > a <- 3; b <- 2
  ```

- Pour ajouter des commentaires, on utilise le symbole dièse '`#`'.

  ```
  > a # Donne la valeur de a
  ```

- Lorsqu'une commande n'est pas complète à la fin d'une ligne, **R** retourne un prompt différent '`+`' jusqu'à ce que la commande soit complétée.

  ```
  > 3-
  + 2
  [1] 1
  ```

3.5 L'opérateur d'affectation

Il est constitué de deux caractères '<' et '-' placés côte à côte et pointant sur l'objet recevant la valeur.

```
> x <- 10
```

Cet opérateur peut être orienté dans l'autre sens.

```
> 10 -> x
```

Le caractère *underscore* '_' est similaire à l'opérateur '<-', il est plus pratique d'utilisation mais il rend le code moins lisible.

```
> x_10
```

On peut aussi utiliser la fonction `assign()` pour réaliser une affectation.

```
> assign("x",10)
```

L'opérateur '<-' peut être vu comme un raccourci de cette fonction.

3.6 Manipulations simples

- **R** peut être utilisé pour réaliser des calculs simples.
  Si on tape :

  ```
  > 5+4
  ```

  R retourne :

  ```
  [1] 9
  ```

  Cette réponse signifie que le résultat est un vecteur dont la première coordonnée est '9'. (**R** ne connaît pas les scalaires, il considère ces derniers comme des vecteurs de longueur 1)
  Le résultat '9' est juste retourné à l'écran, il n'est pas stocké en mémoire.
- On peut créer ou écraser des objets grâce à l'opérateur d'affectation.

  ```
  > x <- 5+6
  ```
- Le contenu d'un objet est visible en tapant son nom :

  ```
  > x
  [1] 11
  ```

3.7 Sauvegarder l'espace de travail

La fonction `save.image()` permet de sauvegarder l'environnement courant de façon ponctuelle. Il n'est pas nécessaire d'attendre la fin de la session de travail pour sauvegarder le travail réalisé. Les différents objets sont stockés dans le fichier '`.RData`'. En ajoutant un nom de fichier à la commande précédente, on peut sauvegarder l'espace de travail sous un nom différent.
Exemple :

```
> save.image(''exercice1'')
```

3.8 Lancer R en « batch »

Il est quelquefois utile de lancer le logiciel en **mode « batch »**.
Ceci peut servir par exemple pour réaliser des simulations, les techniques de Bootstrap, les calculs avec répétitions, les calculs sollicitant un temps de traitement considérable.
Sous *Unix* la syntaxe est la suivante :

```
> R BATCH input output
```

où `input` représente le fichier d'entrée contenant les commandes à exécuter et `output` est le fichier de sortie avec les résultats obtenus.
Exemple :

```
> R BATCH commandes.R resultats
```

Une autre façon d'effectuer cette manipulation est possible en utilisant les symboles de redirection d'*Unix*. (`>`, `<`)
De plus des options sont possibles pour spécifier par exemple si la session effectuée doit être sauvegardée ou pas.
Syntaxe Unix :

```
> R [options] < input > output
```

Exemple :

```
> R --no-save < commandes.R > resultats
```

L'option `--save` permet de sauver les objets à la fin de la session. L'option `--no-save` ne réalise pas cette opération.
Avec la commande *Unix* `at`, on peut définir la date et l'heure de l'exécution. Il n'est pas nécessaire d'être connecté sur une machine, `at` sert souvent pour lancer des exécutions de programmes la nuit ou les week-end.

Chapitre 4

Les objets

On appelle *objets*, **les entités créées et manipulées** par **R**. (variables, tableaux de nombres, chaînes de caractères, fonctions ou structures construites à partir de ces composants)
Les noms des objets sont constitués de lettres, chiffres et points '.'. Un nom ne commence jamais par un chiffre.

4.1 Gestion des objets

Les commandes `objects()` et `ls()` sont identiques et permettent d'afficher les objets stockés en mémoire pour le répertoire courant.

```
>  objects()
[1] "V3"    "laus"    "n"    "somme"   "test"    "test.tr"    "x"    "y"
```

Tous les objets créés pendant une session **R** peuvent être sauvés dans un fichier pour une réutilisation ultérieure. A la fin de chaque session, **R** demande l'on désire sauvegarder tous les objets courants (*workspace*). Si on le fait, ces derniers seront enregistrés dans le fichier appelé '`.RData`' du répertoire courant.
Plus tard, quand on démarrera **R**, il rechargera l'espace de travail du répertoire courant ainsi que l'historique des commandes précédantes.
Avant de commencer à travailler avec **R**, il est donc conseillé de créer un répertoire de travail afin que l'ensemble des objets créés lors des sessions de travail soient stockés dans ce répertoire. Ainsi, différents répertoires peuvent être créés pour l'ensemble des différentes analyses à effectuer.

Des variantes de ces fonctions sont utilisées pour obtenir des résultats particuliers. L'option `pattern` permet d'afficher uniquement les objets contenant une chaîne de caractère particulière.

```
> ls(pattern="s")
[1] "laus"    "somme"  "test"    "test.tr"
```

Les options peuvent être écrites en abrégé ici `pattern` peut s'écrire `pa`.
Pour lister les objets qui commencent par une chaîne particulière, on utilise le symbole '`^`'. (Principe des expressions régulières)

```
> ls(pa="^s")
[1] "somme"
```

Des détails supplémentaires sur les objets en mémoire sont accessibles grâce à la fonction `ls.str()`.

```
> ls.str()
V3 :  chr [1:10] "A" "A" "A" "A" "N" "B" "B" "C" "C" "C"
n :  NULL
somme :  num 0
test : 'data.frame':    10 obs. of  3 variables:
 $ V1: int  5 20 25 15 300 140 550 25 60 650
 $ V2: num  0.1 1.1 1.7 0.9 20.7 10.1 38.2 2.3 4.8 45.8
 $ V3: Factor w/ 4 levels "A","B","C","N": 1 1 1 1 4 2 2 3 3 3
x :  num 12
y :  num 65
```

Pour effacer les objets, on utilise la fonction `rm()` ou `remove()`.

```
> rm(V3,somme)
> rm(list=ls())
> rm(list=ls(all=TRUE))
```

La première instruction efface les objets '`V3`' et '`somme`', la seconde efface tous les objets sauf ceux commençant par un '`.`' et la troisième efface l'ensemble des objets de l'environnement courant (`.GlobalEnv`) y compris ceux commençant par un '`.`'.

4.2 Typologie des objets

Les objets **R** se différencient par leur **mode**, qui décrit leur contenu, et leur **classe**, qui décrit leur structure.
Les différents types ou classes d'objets sont :

- Les vecteurs : `vector`
- Les matrices : `matrix`
- Les tableaux : `array`
- Les facteurs : `factor`
- Les listes : `list`
- Les structures de données : `data.frame`
- Les séries chronologiques : `ts`
- Les fonctions : `function`

Les différents modes des objets sont :

- Non défini : `NULL`
- Logique : `logical`
- Numérique : `numeric`
- Complexe : `complex`
- Caractère : `character`

Si tous les composants d'un objet sont de même mode et si l'objet utilise un de ces modes on dit alors que ces modes sont « atomiques ».

- Liste : `list`
- Structure de données : `data.frame`

Si les composants d'un objet peuvent eux-mêmes être des objets, (exemple : Les listes sont dîtes récursives plutôt qu'atomiques, car leurs composants peuvent eux-même être des listes) on dit alors que ces modes (`list` et `data.frame`) sont « récursifs ».

On nomme **mode** d'un objet, le type de base de ses éléments fondamentaux. C'est l'une des propriétés d'un objet. La **longueur** est l'autre propriété que possède tout objet. Les fonctions `mode()` et `length()` permettent d'acceder au mode et à la longueur de toute structure.
Le mode et la longueur sont des *attributs intrinsèques* des objets, car ils sont communs à tous les objets.

```
> test
    V1   V2 V3
1    5  0.1  A
2   20  1.1  A
3   25  1.7  A
4   15  0.9  A
5  300 20.7  N
6  140 10.1  B
7  550 38.2  B
8   25  2.3  C
9   60  4.8  C
10 650 45.8  C

> mode(test)
[1] "list"

> length(test)
[1] 3
```

La fonction `attributes()` donne la liste de tous les attributs non intrinsèques définis pour un objet.

```
> attributes(test)
$names
[1] "V1" "V2" "V3"

$row.names
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

$class
[1] "data.frame"
```

La fonction `attr()` permet de définir de nouveaux attributs à un objet atomique. (exemple : Permet de définir les dimensions d'une matrice)

```
> test_as.matrix(test)
> attr(test,"dim")_c(10,3) # affecte à l'objet 'test' l'attribut 'dim'
```

Il existe des fonctions liées aux classes d'objets.

- **Les fonctions de définitions**
  `vector(), matrix(), array(), factor(), ts(), data.frame()`

  ```
  > x_matrix(1:50,nrow=5)
  ```

- **Les fonctions de tests**
  `is.null, is.logical, is.numeric, is.complex, is.character`
  `is.vector(),is. matrix(), is.array(), is.factor(), is.ts(),`
  `is.data.frame()`

  ```
  > is.vector(x)
  [1] FALSE
  ```

- **Les fonctions de transformations**
  `as.null, as.logical, as.numeric, as.complex, as.character`
  `as.vector(),as. matrix(), as.array(), as.factor(), as.ts(),`
  `as.data.frame()`

  ```
  >as.data.frame(x)
    V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
  1  1  6 11 16 21 26 31 36 41  46
  2  2  7 12 17 22 27 32 37 42  47
  3  3  8 13 18 23 28 33 38 43  48
  4  4  9 14 19 24 29 34 39 44  49
  5  5 10 15 20 25 30 35 40 45  50
  ```

4.3 Opérateurs intervenant sur les objets

- **Les opérateurs arithmétiques**
  `+ - * / ^` (puissance) `%/%` (division entière) `%%` (modulo)
- **Les opérateurs logiques**
  `> < <= >= ==` (égal), `!=` (différent), `&` (et), | (ou),`!` (non), `xor` (ou exclusif)
- **Les opérateurs matriciels**
  `%*%` (produit de matrices) `%o%` (produit extérieur)

4.4 Les vecteurs

Un **vecteur** est un **objet fondamental** de **R**, c'est un **arrangement d'éléments** qui sont positionnés dans un certain ordre.

4.4.1 Création d'un vecteur

Il existe un grand nombre de fonctions pour créer des vecteurs avec **R**.

Les principales sont :

- La fonction `c()` prend un nombre arbitraire d'arguments et retourne un vecteur en concaténant ces arguments.

```
> c(2,10,5,37,4,8)
[1]  2 10  5 37  4  8
> c(1,2,c(5,10))
[1]  1  2  5 10
```

- La fonction `rep()` prend un ou plusieurs éléments et les crée 'n' fois, elle retourne donc un vecteur qui sera la répétition de ces éléments.

```
> rep(1,5)
[1] 1 1 1 1 1
> rep(c(1,2),3)
[1] 1 2 1 2 1 2
```

- L'opérateur ':' génère un vecteur qui représente une suite de nombres ordonnée par ordre croissant ou décroissant.

```
> 2:11
 [1]  2  3  4  5  6  7  8  9 10 11
> 5:-5
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

- La fonction `seq()` est assez similaire à l'opérateur ':' avec comme particularité supplémentaire la possibilité de donner un pas à la séquence de nombres générée.

```
> seq(2,11)
 [1]  2  3  4  5  6  7  8  9 10 11
> seq(1,25,by=2)
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25
```

D'autres méthodes existent pour créer des vecteurs notamment grâce aux fonctions `vector()`, `sequence()`, `scan()`...

4.4.2 Opérations sur les vecteurs

Les vecteurs peuvent être utilisés comme des **expressions arithmétiques**, alors les opérations sont effectuées élément par élément.
Les vecteurs utilisés dans une même expression doivent tous être de même longueur. Si ce n'est pas le cas, la valeur de l'expression est un vecteur de la même longueur que le plus grand vecteur utilisé dans l'expression.

Les vecteurs plus courts sont alors réutilisés autant de fois que nécessaire pour avoir la même taille que le plus grand.
Comme pour les **vecteurs numériques**, **R** permet de faire des calculs avec des quantités logiques. Les éléments des **vecteurs logiques** peuvent prendre uniquement deux valeurs représentées par `TRUE` et `FALSE`.

```
> v1 <- 1:4
> v2 <- c(5,6,1,2)
> v1+v2
[1] 6 8 4 6
> c(1,2)+c(4,5,6,7)
[1] 5 7 7 9
> v1<=v2
[1]  TRUE  TRUE FALSE FALSE
```

De plus, toutes les fonctions arithmétiques élémentaires sont disponibles. (`log()`, `exp()`, `sin()`, `cos()`, `tan()`, `sqrt()`)
Les fonctions `min()`, `max()`, `sum()` et `prod()` retournent respectivement le minimum, le maximum, la somme et le produit d'un vecteur.
Encore, **R** met à disposition des fonctions statistiques de base. (`mean()`, `var()`, ...)
Il est aussi possible de trier avec `sort()`.

```
> sqrt(v1)
[1] 1.000000 1.414214 1.732051 2.000000
> max(v2)
[1] 6
> mean(v2)
[1] 3.5
> sort(c(v1,v2))
[1] 1 1 2 2 3 4 5 6
```

Les **vecteurs caractères** sont aussi très utilisés, ils sont constitués d'une chaîne de caractères (suite de chiffres et de lettres encadrée par des guillemets).
La fonction `nchar()` donne le nombre de caractère des éléments d'un vecteur de caractère.

```
> nchar(c("R","est","un","logiciel","libre","."))
[1] 1 3 2 8 5 1
```

Les **vecteurs complexes** ont des éléments qui contiennent une partie réelle et une partie imaginaire, ils acceptent les opérations arithmétiques comme les vecteurs numériques.

```
> c(1i,6+3i)
[1] 0+1i 6+3i
> 1i + (6+3i)
[1] 6+4i
```

Les vecteurs comportent quelquefois des **valeurs manquantes**. Lorsque un élément n'est pas disponible au sens statistique du terme, une place dans le vecteur peut lui être réservée en employant la valeur particulière '`NA`'. Ceci sert quand on ne

connaît pas tous les éléments d'un vecteur.
Généralement, toute opération sur un '`NA`' donne un '`NA`'. Il existe une fonction `is.na()` donnant un vecteur logique de la même taille que le vecteur testé avec la valeur `TRUE` pour chaque élément correspondant à un '`NA`'.
De plus, il existe une seconde sorte de valeur manquante qui est produite par des calculs numériques : les valeurs appelées « Not A Number », '`NaN`'

```
> vec_c(1,3,NA,37,NA)
> is.na(vec)
[1] FALSE FALSE  TRUE FALSE  TRUE
> 0/0
[1] NaN
```

4.4.3 Indiçage des vecteurs

Une partie des éléments d'un vecteur peut être selectionnée en ajoutant après le nom du vecteur un *vecteur indice* entre crochets. Les *vecteurs indices* peuvent être de quatre types distincts :

- **Des vecteurs d'entiers positifs :** sont retenus alors les éléments du vecteur correspondant au vecteur indice.

  ```
  > ages_c(10,18,25,60)
  > ages[c(3,1)]
  [1] 25 10
  ```

- **Des vecteurs d'entiers négatifs :** sont retenus alors tous les éléments du vecteur sauf ceux correspondants au vecteur indice, en fait cela sert à spécifier les valeurs à écarter plutôt que celles à sélectionner.

  ```
  > ages[-3]
  [1] 10 18 60
  ```

- **Des vecteurs de valeurs logiques :** sont retenus alors les éléments du vecteur correspondant à la valeur `TRUE` du vecteur indice.

  ```
  > ages[c(F,T,F,T)]
  [1] 18 60
  > ages[ages>20]
  [1] 25 60
  ```

- **Des vecteurs de chaîne de caractères :** dans ce cas le vecteur doit avoir un attribut `names` (à chaque élément du vecteur doit être associé un nom).

  ```
  >  names(ages) <- c("elisabeth","christophe","ludivine","luc")
  > ages
   elisabeth christophe   ludivine         luc
          10         18         25          60
  > ages["ludivine"]
  ludivine
        25
  ```

4.5 Les matrices

Une **matrice** est un jeu de **données arrangées en lignes et en colonnes**. Les matrices comme les vecteurs sont de mode quelconque (réel, complexe,...), mais elles ne peuvent contenir des éléments de natures différentes.

4.5.1 Création d'une matrice

On utilise la fonction `matrix()` pour créer une matrice à partir d'un vecteur.

- On utilise l'argument `nrow` pour définir le nombre de lignes.
- On utilise l'argument `ncol` pour définir le nombre de colonnes.
- On utilise l'argument `byrow` pour définir la méthode de remplissage de la matrice (ligne par ligne ou colonne par colonne), par défaut `byrow` est à `FALSE`.

```
> vec <- 1:10
> mat1 <- matrix(vec,ncol=2)
> mat1
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> mat2_matrix(1:10,nrow=2,byrow=T)
> mat2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

4.5.2 Opérations sur les matrices

En plus des attributs `length` et `mode`, une matrice a un attribut *dimension* qui est un vecteur à deux éléments, le nombre de lignes et de colonnes.

```
> dim(mat)
[1] 5 2
```

Le *produit* de deux matrices s'écrit avec l'opérateur `%*%`. Le langage contrôle l'adéquation des dimensions :

```
> mat.res <-  mat1 %*% mat2
> mat.res
     [,1] [,2] [,3] [,4] [,5]
[1,]   37   44   51   58   65
[2,]   44   53   62   71   80
[3,]   51   62   73   84   95
[4,]   58   71   84   97  110
[5,]   65   80   95  110  125
```

La fonction `diag()` permet d'extraire la *diagonale* d'une matrice carrée ou de construire une matrice diagonale à partir d'un vecteur.

```
> diag(mat.res)
[1]  37  53  73  97 125
> diag(c(6,5,3))
     [,1] [,2] [,3]
[1,]    6    0    0
[2,]    0    5    0
[3,]    0    0    3
```

La fonction `t()` *transpose* une matrice.

```
> t(mat1)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

La fonction `solve()` permet d'*inverser* une matrice.

```
> mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> solve(mat)
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Les fonction `rbinb()` et `cbinb()` permettent de *concaténer* par ligne ou par colonne des vecteurs ou des matrices.

```
> vec1 <- c(10,2,5)
> vec2 <- c(25,8,7)
> mat <- rbind(vec1,vec2)
> mat
     [,1] [,2] [,3]
vec1   10    2    5
vec2   25    8    7
> vec3 <- c(51,37)
>  cbind(mat,vec3)
            vec3
vec1 10 2 5   51
vec2 25 8 7   37
```

Pour *diagonaliser* une matrice carrée, on utilise `eigen()`, qui fournit comme résultat une liste contenant deux composants : `$values` qui contient les valeurs propres, `$vectors` qui contient les vecteurs propres normés.

```
> a <- matrix(c(6,2,0,2,6,0,0,0,36),nrow=3)
```

```
> a
     [,1] [,2] [,3]
[1,]    6    2    0
[2,]    2    6    0
[3,]    0    0   36
> eigen(a)
$values
[1] 36  8  4

$vectors
     [,1]      [,2]       [,3]
[1,]    0 0.7071068  0.7071068
[2,]    0 0.7071068 -0.7071068
[3,]    1 0.0000000  0.0000000
```

La fonction `apply()` permet d'effectuer des fonctions qui opèrent ligne par ligne ou colonne par colonne.

```
> mat_matrix(1:12,ncol=4)
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(mat,1,sum)
[1] 22 26 30
> apply(mat,2,mean)
[1]  2  5  8 11
```

4.5.3 Indiçage de matrices

L'indiçage des matrices s'effectue de façon analogue à l'indiçage des vecteurs (utilisation de `[ ]`, les lignes et les colonnes sont séparées par une virgule ',')
On peut aussi utiliser la fonction `dimnames()` pour nommer les lignes et les colonnes.

```
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> mat[3,2]
[1] 6
> mat[,2]
[1] 4 5 6
> mat[-c(2,3),]
[1]  1  4  7 10
> dimnames(mat) <- list(c("r1","r2","r3"),c("c1","c2","c3","c4"))
```

```
> mat
   c1 c2 c3 c4
r1  1  4  7 10
r2  2  5  8 11
r3  3  6  9 12
> mat["r1",c("c1","c3")]
c1 c3
 1  7
```

4.6 Les facteurs

Un **facteur** est un objet vectoriel utilisé pour identifier les composants d'autres vecteurs ayant la même longueur.
Les facteurs sont utilisés pour décrire des catégories qui ont un nombre fini de modalités. (Par exemple : le sexe, la classe sociale, la couleur des yeux, ...)
Un facteur peut être considéré comme une variable qualitative en statistique.
R peut fournir des facteurs dont les modalités sont ordonnées ou non ordonnées.
Les facteurs sont surtout utilisés dans les formules pour **décrire des modèles**.
La fonction permettant de créer un facteur est `factor()`.
Exemple de création d'un facteur :

```
> sexe_factor(c(rep("M",10),rep("F",10)))
> sexe
 [1] M M M M M M M M M M F F F F F F F F F F
Levels:  F M
```

La fonction `levels()` est utilisée pour afficher les différentes modalités d'un facteur.

```
> levels(sexe)
[1] "F" "M"
```

4.7 Les listes

Une **liste** est un objet qui **contient un ensemble ordonné d'objets** qui sont appelés composants.
Il n'est pas nécessaire que les composants soient du même type ou du même mode, et, par exemple une liste peut être composée d'un vecteur numérique, d'une valeur logique, d'une matrice, d'un vecteur de complexes, d'un tableau de caractères, d'une fonction, etc.
Les listes sont des *objets récursifs*, c'est-à-dire que les composants d'une liste peuvent eux mêmes être des listes.

4.7.1 Création d'une liste

On utilise la fonction `list()` pour créer une liste et nommer ses composants.

```
> li.IUP <- list(nom.formation="IUP SID",c("Deug","Licence","Maitrise")
 ,nb.promo=3,nb.etud=80)
> li.IUP
$nom.formation
[1] "IUP SID"

[[2]]
[1] "Deug"     "Licence"  "Maitrise"

$nb.promo
[1] 3

$nb.etud
[1] 80
```

4.7.2 Opérations sur les listes

Il est possible de réaliser des *opérations de concaténation* sur les listes.

```
> nom.responsable <- "Mathieu"
> liste <- c(liste1,nom.responsable)
> liste
$nom.formation
[1] "IUP SID"

[[2]]
[1] "Deug"     "Licence"  "Maitrise"

$nb.promo
[1] 3

$nb.etud
[1] 80
```

```
[[5]]
[1] "Mathieu"
```

La longueur d'une liste est égale au nombre de ses composants.

```
> length(liste)
[1] 5
```

La fonction `names()` permet de *nommer les composants* et de visualiser leurs noms.

```
> names(liste)
[1] "nom.formation" ""              "nb.promo"      "nb.etud"
[5] ""
```

On peut ajouter de nouveaux composants.

```
> liste$etablissement <- "UPS"
```

4.7.3 Extraction des composants d'une liste

Il existe *deux moyens* pour extraire les composants d'une liste :

- Par l'utilisation de l'opérateur '`$`' si le composant de la liste à un nom

  ```
  > liste$nom.formation
  [1] "IUP SID"
  ```

- Par l'utilisation de `[[ ]]`

  ```
  > liste[[2]]
  [1] "Deug"     "Licence"  "Maitrise"
  ```

Par convention, on utilise '`$`' si le composant est nommé et `[[ ]]` dans les autres cas.
Si l'on utilise `[ ]`, le résultat est un objet qui spécifie ses composants.

```
> liste[3]
$nb.promo
[1] 3
```

4.8 Les structures de données

Une **structure de données** ou **« data frame »** est une **généralisation de matrice** qui autorise les colonnes à avoir des modes différents.
Un « data frame »est très utile en analyse de données où l'on a généralement une variable caractère 'individu' et plusieurs variables observées pouvant être soit qualitatives (numériques ou caractères) soit quantitatives ou même logiques.

4.8.1 Création d'une structure de données

Nous pouvons créer un « data frame »de *différentes façons* :

- Avec la fonction `data.frame()` à partir de vecteurs (numériques, caractères ou logiques), de facteurs, de matrices, de listes ou d'autres « data frames ».
  Les noms d'objets deviennent les noms des colonnes.
  Les noms des lignes sont déterminées par les noms des lignes de la matrice argument ou par les noms des lignes d'un « data frame »précédant ou par défaut le numéro de ligne.

```
> df <- data.frame(V1,V2,V3)
> df[1:5,]
   V1 V2 V3
1   1  1  A
2   3  1  A
3   5  4  A
4   7  4  A
5   9  4  N
```

- En contraignant un objet (liste, matrice) à devenir une structure de données avec la fonction `as.data.frame()`

```
> as.data.frame(diag(1:5))
  V1 V2 V3 V4 V5
1  1  0  0  0  0
2  0  2  0  0  0
3  0  0  3  0  0
4  0  0  0  4  0
5  0  0  0  0  5
```

- En lisant une structure de données stockée dans un fichier. (fonction `read.table()` par exemple)
  Cette méthode sera expliquée plus en détails dans le chapitre *« Importation et exportation de données »*.

4.8.2 Opérations sur les structures de données

La fonction `names()` et `row.names()` permettent de *visualiser* et de *spécifier* explicitement le nom des colonnes et celui des lignes.

```
> names(df)
[1] "V1" "V2" "V3"
> row.names(df)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Les fonctions `cbind()` et `rbind()` s'utilisent de la même façon que pour les autres objets (vecteurs, matrices), elles permettent de *combiner* en colonnes ou en lignes plusieurs « data frames »(par exemple pour ajouter de nouvelles variables ou de nouvelles données à une structure de données).

La fonction `summary()` permet d'obtenir un *résumé des données* du « data frame ».

```
> summary(df)
       V1              V2       V3
 Min.   : 1.0   Min.   :1.00   A:4
 1st Qu.: 5.5   1st Qu.:2.50   B:2
 Median :10.0   Median :4.00   C:3
 Mean   :10.0   Mean   :3.50   N:1
 3rd Qu.:14.5   3rd Qu.:4.75
 Max.   :19.0   Max.   :5.00
```

4.8.3 Extraction d'informations d'une structure de données

On extrait les informations d'un « data frame »soit de façon identique à une matrice soit de façon identique à une liste.

La notation '`$`' comme dans '`df$V3`', n'est pas toujours pratique pour accéder aux composants d'une liste ou d'un « data frame ».
Il est utile de rendre temporairement visibles ces derniers pour les utiliser plus aisément. (par exemple en tapant directement `> V3`)
La fonction permettant de réaliser cette opération est `attach()`, elle sert pour les listes, les « data frames »mais aussi les noms de répertoires.

```
> attach(df)
```

Maintenant, les variables du « data frame »sont manipulables par leurs propres noms si il n'existe pas de noms d'objets similaires.

```
> V2 # V2 est une colonne du data frame 'df'
 [1] 1 1 4 4 4 4 5 5 5 2
```

Cependant, les diverses modifications ne seront pas prises en compte par le « data frame ». Pour cela, il faut malgré tout utiliser le '`$`'.

Pour « détacher »un objet, on utilise la fonction `detach()`, cette fonction a pour effet de ne plus rendre visibles les différentes variables du « data frame ».

```
> detach(df)
```

La fonction `search()` permet de gérer le répertoire courant grâce à `attach()` et `detach()`.
En effet, elle affiche les éléments du répertoire courant ce qui est très utile pour garder la trace des listes, « data frames »et aussi des modules qui ont été attachés et détachés.

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

Après avoir attaché '`df`' et le module **ctest**, nous avons :

```
> search()
[1] ".GlobalEnv"    "df"         "package:ctest"         "Autoloads"
[5] "package:base"
```

Chapitre 5

Importation et exportation de données

Lire des données dans un système statistique pour les analyser et ensuite exporter les résultats dans un autre système pour pouvoir les commenter peut être une tâche plus longue et plus pénible que l'analyse statistique elle-même.
R comporte des moyens permettant l'**importation** et l'**exportation** des données. Des modules supplémentaires sont disponibles pour des travaux spécifiques concernant les fichiers particuliers.

5.1 Importation

Les jeux de données sont en général lus dans des fichiers externes plutôt qu'entrés manuellement durant une session **R**.

5.1.1 La fonction read.table()

La fonction `read.table()` permet de *lire des fichiers de données au format texte* (ASCII). Cette fonction est destinée à lire les tableaux de données, les données issues d'un tableur qui auraient été préalablement transformées en fichier texte, ...

La syntaxe de cette fonction avec la valeur par défaut de ses arguments est :

```
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 = "#")
```

La description de ses principaux arguments est :
- `file` : Le nom du fichier à lire entre guillemets (avec éventuellement son chemin d'accès)
- `header` : Une valeur logique (`TRUE` ou `FALSE`) indiquant si le fichier contient les noms de variable sur la première ligne du fichier
- `sep` : Le séparateur de champs entre guillemets
- `dec` : Le caractère utilisé pour les décimales entre guillemets

Exemple :

```
> consom <- read.table("consom3.dat",T)
> consom[1:5,]
   km   cs
1   5  0.1
2  20  1.1
3  25  1.7
4  15  0.9
5 300 20.7
```

L'objet obtenu est un « data frame ».

Il existe plusieurs variantes de cette fonction pour lesquelles la valeur par défaut des arguments change.
(`read.csv()`, `read.csv2()`, `read.delim()` et `read.delim2()`)

5.1.2 La fonction read.fwf()

Pour cette fonction, les attributs sont similaires à ceux de `read.table()` avec en plus l'attribut `widths` qui *spécifie la largeur des champs.*

La syntaxe de cette fonction avec la valeur par défaut de ses arguments est :

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

Exemple :

```
Sous Unix :
Fichier : "test.txt"
sid1g2
sid2g6
sid1g15
sid3g18
sid3g5

Dans R :
> test <- read.fwf("test.txt", c(3,1,1,2))
> test
   V1 V2 V3 V4
1 sid  1  g  2
2 sid  2  g  6
3 sid  1  g 15
4 sid  3  g 18
5 sid  3  g  5
```

5.1.3 La fonction scan()

La fonction `scan()` est plus souple que `read.table()` et comporte plus d'arguments.
L'argument `what` permet de décrire le mode des variables qui vont être lues dans le fichier. Ces modes peuvent être numériques, caractères ou complexes et sont respectivement spécifiés par `0, "", 0i`.

La syntaxe de cette fonction avec la valeur par défaut de ses arguments est :

```
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 = "")
```

Exemple :

```
Sous Unix :
Fichier : "test.dat"
  5  0.1 A
 20  1.1 A
 25  1.7 A
 15  0.9 A
300 20.7 N
...

Dans R :
> scan("test.dat",list(0,0,""))
Read 10 records
[[1]]
 [1]   5  20  25  15 300 140 550  25  60 650

[[2]]
 [1]  0.1  1.1  1.7  0.9 20.7 10.1 38.2  2.3  4.8 45.8

[[3]]
 [1] "A" "A" "A" "A" "N" "B" "B" "C" "C" "C"
```

5.1.4 Le module Foreign

Le module **Foreign** fournit des moyens pour *importer des données* de fichiers produits par des systèmes statistiques tel que **Minitab, S, SAS, SPSS, Stata**, ... Il permet aussi d'exporter des données sous Stata.

- Les fichiers **Stata** '`.dta`' peuvent être importés (lus) et exportés (écrits) par **R** grâce aux fonctions `read.dta()` et `write.dta()`.
- La fonction `read.epiinfo()` permet de lire des fichiers '`.REC`' dans **R** en les transformant en « data frame ». Ces fichiers sont issus du logiciel **EpiInfo**.

- La fonction `read.mtp()` importe un « Minitab Portable Worksheet »et le résultat apparaît comme une liste **R**.
- La fonction `read.xport()` lie les fichiers au format « SAS Transport »(XPORT) et retourne un « data frame ». Si l'on possède **SAS**, on peut utiliser la fonction `read.ssd()` pour créer un script **SAS** permettant de sauver les tableaux de données **SAS** (`.ssd` ou `.sas7bdat`) au format de transport. De là, on appelle `read.xport()` pour lire le fichier qui vient d'être créé.
- La fonction `read.spss()` permet de lire les fichiers créés par les commandes « save »et « export »de **SPSS**. Il est retourné une liste avec un composant pour chaque variable du tableau de données **SPSS** sauvé.
- La fonction `read.S()` peut lire les objets produits par **S-Plus**. Cette fonction peut lire des objets **S**, mais pas tous. Elle lie particulièrement les vecteurs, les matrices, les « data frames »et les listes les contenant.
- La fonction `data.restore()` permet de lire les fichiers textes contenant des instructions créés par **S-Plus** à l'aide de la commande **S** `data.dump()`.

Exemple d'importation d'un fichier **SPSS** :

```
> laus <- read.spss("lausanne.sav",to.data.frame=TRUE)
```

Il est conseillé de fixer l'option `to.data.frame` à `TRUE` pour que les données soient facilement exploitables sous la forme d'un tableau de données.

5.1.5 Les autres modules

- Le module **e1071** grâce à sa fonction `read.octave` permet de lire un vecteur ou une matrice issu d'un fichier de données **Octave** ASCII créé par la commande **Octave** `save -ascii`.
  Octave est un système numérique d'algèbre linéaire libre qui fait partie du « GNU-project ». Il utilise un langage qui est compatible avec **Matlab**.
- Le langage **XML** peut être utilisé pour décrire non seulement le contenu d'un fichier mais aussi la structure de ce contenu. De ce fait, nous n'avons pas besoin de fournir les détails concernant la structure tels que la spécification d'une ligne comportant les variables, le séparateur utilisé, la représentation des valeurs manquantes, etc..., pour pouvoir lire les données.
  XML peut donc fournir des tableaux de données standards mais aussi des structures de données plus complexes. **XML** est devenu très populaire et apparaît comme un standard pour le transport et le balisage des données.
  Le module **XML** fournit des moyens de lecture et d'écriture des documents **XML** pour **R** et pour **S-Plus** dans l'espoir d'utiliser facilement cette technologie.
- **R** n'est pas conçu pour gérer de grosses quantités de données et il ne permet pas non plus à plusieurs utilisateurs d'intégrer en temps réel les modifications des uns et des autres.
  Ce travail est généralement réalisé par les systèmes de gestion de bases de données, qu'ils soient commerciaux (**Oracle, Microsoft SQL Server**, etc...) ou libres (**MySQL, PostgreSQL**, etc...)

Des modules faisant le lien entre les **SGBD** et **R** ont donc été développés pour pour bénéficier des capacités spécifiques de ces deux domaines d'application.

- Les **SGBD** pour le stockage, la gestion et l'extraction des données
- **R** pour les analyses statistiques et graphiques

On trouve alors différents modules liés à la manipulation des **SGDB** :

- **RmSQL** pour **MiniSQL**
- **RMySQL** pour **MySQL**
- **ROracle** pour **Oracle**
- **RPgSQL** pour **PostgreSQL**
- **RSQLite** pour **SQLite**

Ces outils permettent d'importer et d'exporter des tableaux de données entre **R** et les **SGBD**, d'exécuter des requêtes et d'importer les résultats dans **R** afin de les analyser.

5.2 Exportation

Comme nous avons pu le voir précédemment avec `write.dta` de **Foreign**, le module **XML** et les modules de bases de données, il est possible d'exporter des données à partir de **R**.

Les fonctions `write()` et `write.table()` permettent d'*exporter* des objets **R** en fichiers textes. La fonction `write()` sert pour les vecteurs et les matrices, la fonction `write.table()`, elle, exporte les « data frames »avec les noms de ligne et de colonne.

La syntaxe de ces fonctions avec la valeur par défaut de leurs arguments est :

```
write(x, file = "data",
      ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE)

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

Exemples :

```
Dans R :
> write(V3,"vect.txt")
> write.table(test,"test.dat")

Sous Unix :
Fichiers :          ‘‘vect.txt’’          ‘‘test.dat’’
                                        "V1" "V2" "V3"
                         A             "1"   5  0.1 "A"
                         A             "2"  20  1.1 "A"
                         A             "3"  25  1.7 "A"
                         A             "4"  15  0.9 "A"
                         N             "5" 300 20.7 "N"
                         B             "6" 140 10.1 "B"
                         B             "7" 550 38.2 "B"
                         C             "8"  25  2.3 "C"
                         C             "9"  60  4.8 "C"
                         C            "10" 650 45.8 "C"
```

Dans le module **MASS**, il existe une fonction `write.matrix` qui écrit (exporte) une matrice dans un fichier texte.

Exemple :

```
Dans R :
> write.matrix(mat,"matrix.dat")
```

```
Sous Unix :
Fichier : ‘‘matrix.dat’’
           2 0 0 0
           0 3 0 0
           0 0 5 0
           0 0 0 6
```

Il est aussi possible de rediriger les sorties de **R** vers un fichier en tapant :

```
> sink("nom du fichier")
.....
```

Commandes **R** (Seules les sorties sont envoyées dans le fichier)

```
.....
> sink()
```

Les fonctions `dump()` et `source()` permettent d'exporter des objets **R** dans des fichiers « ascii »et ensuite de les récupérer.

Exemple :

```
> dump(c("A","B","C"),"abc.R")
```

Sauvegarde les objets `A, B, C` dans le fichier texte ‘`abc.R`.

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

Récupération des objets sauvés dans ‘`abc.R`’.

La fonction `dput()` écrit dans un fichier texte la représentation d'un objet **R** et la fonction `dget()` utilise un fichier pour recréer un objet.

Exemple :

```
> dput(faithful,"faithful.txt")
> fait <- dget("faithful.txt")
```

Chapitre 6

Les fonctions

Les fonctions sont des objets **R** de mode `function` qui sont stockés en mémoire et qui peuvent contenir une suite d'instructions avec en plus la possibilité d'appeler d'autres fonctions. C'est sur ce principe de fonctions construites sur la base d'autres fonctions que repose une partie de la puissance et de la concision de ce langage.
Les fonctions forment **l'unité de base de la programmation** sous **R**.
Toute commande est une fonction qui peut avoir un nombre quelconque d'arguments.
Si '`f`' est une fonction, lorsqu'on écrit :

```
> g <- f
```

alors '`g`' devient une fonction égale à '`f`'.
Si l'on veut affecter à '`y`' la valeur de la fonction '`f`' en '`x`', on écrit :

```
> y <- f(x)
```

6.1 Syntaxe d'une fonction

L'*appel* d'une fonction se fait par :

```
> nom-de-la-fonction (arguments)
```

Le passage des arguments se fait, soit par rang (séparation par des virgules), soit par nom (nom de l'argument considéré), ou par mélange des deux.

Exemples :

```
> rep(1:3,3)   # passage des arguments par rang
> rep(1:3,,9)   # idem
> rep(1:3,length=9)   # passage des arguments par nom
> rep(1:3,times=3,9)    # passage par rang et par nom
```

Le second argument de la fonction `rep()` est `times` et le troisième `length`.
Les fonctions **R** possèdent des arguments obligatoires et des arguments optionnels affectés par défaut. Pour calculer la valeur d'une fonction, il est donc nécessaire de lui donner les arguments obligatoires. Si les autres ne sont pas fournis, la fonction utilise les valeurs par défaut.
La fonction `args()` donne les arguments d'une fonction.

Exemple de valeurs par défaut pour les arguments d'une fonction :

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

Les commandes d'aide permettent d'obtenir une aide concernant la syntaxe, les arguments et les principes d'utilisation de chaque fonction.
La fonction `example()` donne des exemples de démonstration pour une fonction considérée.

Exemples :

```
> example(median)
median> median(1:4)
[1] 2.5
median> median(c(1:3, 100, 1000))
[1] 3
```

Pour obtenir le code d'une fonction, il suffit de taper son nom :

```
> rm
function (..., list = character(0), pos = -1, envir = as.environment(pos),
    inherits = FALSE)
{
    names <- as.character(substitute(list(...)))[-1]
    list <- .Primitive("c")(list, names)
    .Internal(remove(list, envir, inherits))
}
```

6.2 Fonctions et opérateurs les plus utilisés

Certaines de ces fonctions sont présentées plus précisément dans les autres parties du document.

6.2.1 Les fonctions génériques

Ce sont les fonctions qui *s'appliquent à tout type d'objets*, mais qui exécutent une tâche spécifique de la classe de l'objet à laquelle elle s'applique.

Les principales fonctions génériques sont les fonctions :

- `plot`
  Cette fonction trace un graphique selon le type ou la classe de l'objet utilisé.
- `print`
  Permet de visualiser un objet.
- `summary`
  Donne un résumé de statistiques simples pour un objet.

En pratique, les fonctions réellement exécutées sont différentes pour différentes classes d'objets.
En effet, en tapant `summary(x)`, on fait appel à la fonction `summary.default()` si '`x`' est un vecteur, à la fonction `summary.data.frame()` si '`x`' est un « data frame », à la fonction `summary.matrix()` si '`x`' est une matrice, etc.

6.2.2 La manipulation de données

Fonctions	Descriptions
`c`	Sélection des valeurs créant un vecteur
`seq :`	Création d'une séquence de nombres
`rep`	Répétition de valeurs
`list`	Création d'une liste
`scan`	Lecture de données à partir d'un fichier ou du terminal
`rev`	Renverse l'ordre des éléments dans un vecteur ou une liste
`sort`	Trie de façon numérique ou alphanumérique ascendante
`rank`	Donne le rang des valeurs
`split`	Sectionne les données par groupes
`na.omit`	Supprime les observations avec des données manquantes (`NA`)
`round`	Arrondit les éléments
`dim`	Dimension(s) d'un objet
`length`	Longueur d'un objet
`mode`	Mode d'un objet
`attributes`	Visualisation des attributs d'un objet
`attach`	Ajoute un objet ou répertoire à l'espace de travail courant
`library`	Utilisation de modules
`objects ls`	Affichage d'objets
`rm remove`	Suppression d'objets
`fix`	Manipulation de fonctions et de « data frames »

6.2.3 Matrices, tableaux et mathématiques

Fonctions	Descriptions
`matrix`	Création de matrice
`array`	Création de tableau
`row col`	Identification de lignes et de colonnes
`diag`	Extrait ou remplace la diagonale d'une matrice
`diag`	Création d'une matrice diagonale
`cbind`	Concaténation par colonnes pour construire une matrice
`rbind`	Concaténation par lignes pour construire une matrice
`t`	Transpose une matrice
`solve`	Inverse une matrice
`%*%`	Opérateur de multiplication de matrice
`ts`	Création de séries chronologiques
`tsp`	Retourne les attributs de séries chronologiques
`min`	Minimum
`max`	Maximum
`range`	Minimum et maximum
`mean`	Moyenne empirique
`sd`	Ecart-type
`median`	Médiane
`quantile`	Quantiles
`apply`	Applique une fonction aux sections d'un tableau
`lapply` `sapply`	Applique une fonction aux composants d'une liste
`sum`	Somme des éléments d'un vecteur
`prod`	Produit des éléments d'un vecteur
`cumsum`	Somme cumulée des éléments d'un vecteur
`cumprod`	Produit cumulé des éléments d'un vecteur
`var`	Variance des éléments
`cov`	Covariance des éléments
`cor`	Corrélation linéaire
`log`	Logarithme
`exp`	Exponentiel
`sin`	Sinus
`cos`	Cosinus
`tan`	Tangente
`sqrt`	Racine carrée
`abs`	Valeur absolue

6.3 Les lois de probabilité

R fournit un jeu très complet de *tables statistiques*. Les fonctions fournies nous donnent la densité de probabilité cumulée (ou fonction de répartition), $P(X \leqslant x)$, la densité de probabilité, le quantile (ou fonction de répartition inverse) et des séries de nombres pseudo aléatoires générées suivant la loi de probabilité considérée.

Lois de probabilité	**Noms R**	**Arguments supplémentaires**
Beta	`beta`	`shape1, shape2, ncp`
Binomiale	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
Khi2	`chisq`	`df, ncp`
Exponentielle	`exp`	`rate`
Fisher (F)	`f`	`df1, df2, ncp`
Gamma	`gamma`	`shape, scale`
Géométrique	`geom`	`prob`
Hypergéométrique	`hyper`	`m, n, k`
Log-normale	`lnorm`	`meanlog, sdlog`
Logistique	`logis`	`location, scale`
Négative binomiale	`nbinom`	`size, prob`
Normal	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
Student (t)	`t`	`df, ncp`
Uniforme	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

Pour les lois Beta, Khi2, Fisher et Student `ncp` désigne le paramètre de décentrage.
Quatre préfixes sont possibles pour obtenir les sorties désirées :

- `d` : correspond à la densité de probabilité
- `p` : correspond à la fonction de répartition (valeur inverse du quantile)
- `q` : correspond au quantile
- `r` : correspond à la génération aléatoire de nombre

Le premier argument est '`x`'(une valeur) pour `d`*nom_loi*, '`q`'(un quantile) pour `p`*nom_loi*, '`p`'(une probabilité) pour `q`*nom_loi* et '`n`'(un nombre) pour `r`*nom_loi*.

Exemples avec la loi normale (arguments par défauts : `mean=0`, `sd=1`) :

```
> dnorm(0.37)       # donne l'ordonnée de la courbe de densité
[1] 0.3725483         à la valeur 0.37
> pnorm(0)     # donne la probabilité de la fonction de répartition
[1] 0.5          au quantile 0
> round(qnorm(0.975),2)  # donne le quantile pour la probabilité
[1] 1.96                   de 0.975 de la fonction de répartition
> rnorm(5)    # engendre 5 valeurs de la loi normale
[1] -0.4974692  0.3292089  0.4690761 -0.7798251  1.2353437
```

6.4 Ecriture de fonctions

Comme nous avons pu le voir, le langage **R** permet à l'utilisateur de créer des objets de mode `function`. Ces objets sont des fonctions **R** qui sont stockées en interne de façon appropriée et qui peuvent être utilisées dans des expressions, dans d'autres fonctions ... Avec cette méthode le langage gagne en puissance, élégance et facilité d'utilisation. Apprendre à écrire des fonctions permet d'utiliser **R** de façon confortable et productive.
Il faut préciser que la plupart des fonctions faisant partie du système **R** comme `mean()`, `var()`, etc, sont elles-mêmes écrites en **R** et ne diffèrent pas des fonctions écrites par les utilisateurs de ce point de vue.

La syntaxe générale de définition d'une fonction est la suivante :

```
nom_fonction <- function(arg1[=expr1], arg2[=expr2], ...){
        blocs d'instructions
}
```

Les *accolades* permettent de séparer les instructions par rapport à la signature de la fonction, les *crochets*, eux permettent de spécifier les valeurs par défaut des arguments de façon facultative.

Il existe *deux moyens* d'écrire ses propres fonctions :

- Directement à partir de la ligne de commande du logiciel

  Exemple définissant la moyenne d'un vecteur :

  ```
  > moyenne.vec <- function(x){
  + s <- sum(x); # Somme des éléments de x
  + n <- length(x);  # Nombre d'éléments de x
  + res <- round(s/n,2);   # Résultat arrondi
  + return(res)
  + }
  ```

- Par l'intermédiaire d'un éditeur de texte grâce à la fonction `fix()`.

  Exemple :

  ```
  > fix(moyenne.vect)
  ```

  Cette commande lance un éditeur de texte qui travaille avec le système **R** et qui permet de définir des fonctions. Le code de la fonction est écrit à partir de l'éditeur. Cette méthode est plus confortable et pratique que la première.

La fonction `return()` permet de *spécifier le résultat* de la fonction, lorsque l'instruction correspondante à `return` n'est pas utilisée, **R** retourne le résultat de la dernière expression évaluée dans la fonction.

On utilise l'argument « `...` » pour *spécifier un nombre indéterminé d'arguments.*
Les valeurs des arguments ne sont pas modifiées par les expressions contenues dans la fonction, en effet leur portée est locale. Lors de l'exécution, une copie des arguments est transmise à la fonction, laissant les originaux intacts. Pour réaliser une affectation globale, on utilise l'opérateur '`<<-`'.

De plus, il est conseillé d'intégrer des commentaires au programme pour une bonne compréhension du code source. Ces commentaires sont insérés à l'aide du caractère dièse '`#`'.

La fonction `edit()` permet de définir une nouvelle fonction à l'aide d'une fonction déjà existante.

Exemple :

```
> autre.fonction <- edit(moyenne.vec)
```

6.5 Eléments de programmation

6.5.1 Les instructions de sélection

- **L'expression** `if` :
  `if (expression logique) <expression1>`
  ou
  `if (expression logique) <expression1> else <expression2>`
  Si l'expression logique prend la valeur vrai (`TRUE`), l'expression 1 est évaluée, sinon c'est l'expression 2.
- **L'expression** `ifelse` :
  `ifelse (expression logique, vecteur1, vecteur2)`
  Si l'expression logique prend la valeur vrai, l'expression retourne le vecteur 1 sinon elle retourne le vecteur 2.

L'expression `switch()` aussi permet de réaliser des sélections.

6.5.2 Les instructions de répétition

- **L'expression** `while` :
  `while (expression logique, expression)`
  Tant que l'expression logique est vraie, l'expression est exécutée. La valeur finale est celle obtenue lors de la dernière évaluation de l'expression.
- **L'expression** `for` :
  `for (ind in vecteur) expression`
  `ind` est la variable boucle, `vecteur` est une expression vectorielle (souvent une séquence comme `1:30`) et `expression` est souvent un groupe d'expressions qui composent avec la variable `ind`. `expression` est évaluée de manière répétitive tant que `ind` parcourt les valeurs de `vecteur`.

L'instruction `repeat` aussi permet de réaliser des répétitions.
L'instruction `break`, pour sortir d'une boucle, peut être utilisée dans toutes ces boucles et elle est la seule façon de terminer la boucle `repeat`.
L'instruction `next` peut être utilisée pour sauter une étape et passer à la suivante.

Par ailleurs, **R** autorise la récursivité et il permet à l'utilisateur de créer ses propres opérateurs.

6.6 Quelques aides à la programmation

R met à disposition plusieurs fonction d'aide à la programmation.

`browser()`
Permet d'arrêter le programme et de contrôler la valeur de tous les paramètres puis de relancer. `browser()` est placé en un point quelconque d'une fonction.
Exemple :
- Edition d'une fonction
- Ajout de `browser()`
- Exécution

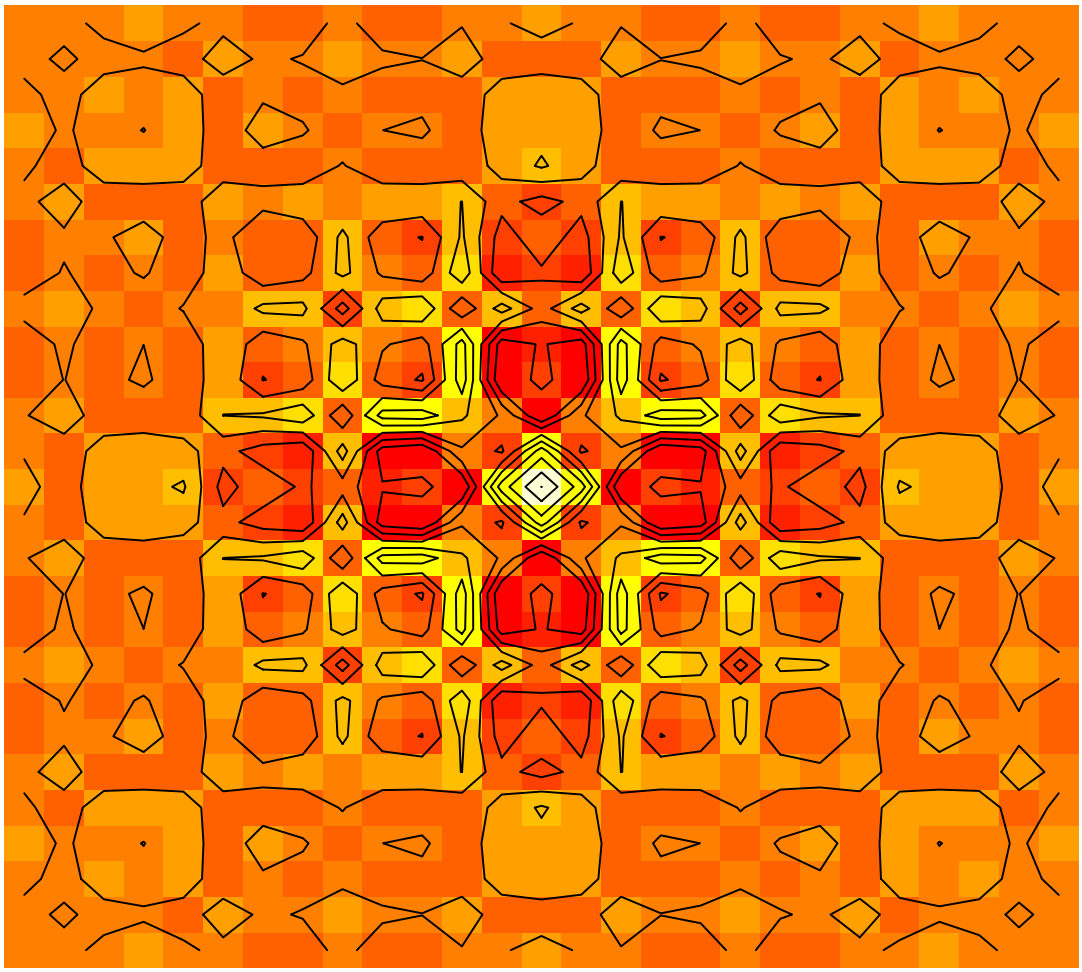
`traceback()`
Permet de connaître immédiatement la fonction dans laquelle s'est produite l'erreur.

Les fonctions `trace()` et `debugger()` donnent aussi de l'aide pour la programmation. Pour plus d'informations sur ces fonctions, il faut consulter l'aide de **R**.

Chapitre 7

Les graphiques

Les mathématiques peuvent être belles ...



$\cos(r^2)e^{-r/6}$

Les fonctions graphiques sont considérées comme une partie importante et très modifiable de l'environnement **R**. Il est possible d'utiliser ces moyens afin de produire une grande variété de graphiques statistiques mais aussi d'élaborer de nouveaux types de graphiques. Dans ce cadre là, **R** pourrait être vu comme un système de développement graphique. Nous nous contentons donc de donner un aperçu des possibilités du système dans le domaine graphique.

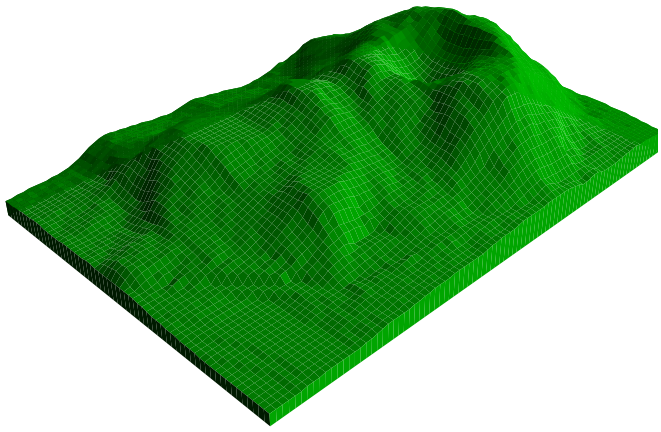
Les commandes graphiques peuvent être utilisés autant en mode interactif qu'en mode *batch*, mais le mode interactif est, dans la plupart des cas, le plus utile. L'utilisation interactive est également facile car, au démarrage, **R** lance un pilote (*device driver*) graphique qui ouvre au besoin des fenêtres graphiques pour l'affichage des graphiques interactifs. Même si cela se fait automatiquement, il est utile de savoir que la fonction `X11()` est employée sous *Unix*, `window()` sous *Windows* et `macintosh()` sous *MacOS 8/9* pour ouvrir une fenêtre graphique.

Une fois que le pilote est en marche, les commandes graphiques peuvent être utilisées pour produire et créer des graphiques.

Les commandes graphiques sont divisées en **trois grands groupes** :

- Les fonctions graphiques **de haut-niveau** créent un nouveau graphique sur la fenêtre graphique, avec éventuellement des axes, des labels, des titres, etc.
- Les fonctions graphiques **de bas niveau** rajoutent de l'information à un graphique existant, comme des points supplémentaires, des lignes ou des labels.
- Les fonctions graphiques **interactives** permettent de rajouter, ou d'enlever, de l'information sur un graphique existant de façon interactive, à l'aide d'un mécanisme de pointage, comme la souris.

En supplément, **R** garde une liste des *paramètres graphiques* qui peuvent être manipulés pour personnaliser les graphiques.



7.1 Les fonctions graphiques de haut niveau

Les fonctions graphiques de haut-niveau sont prévues pour **créer un graphique complet** à partir des données passées en argument de la fonction. Quand c'est approprié, des axes, labels et titres sont automatiquement générés (à moins de spécifier le contraire). Les commandes graphiques de haut niveau produisent toujours un nouveau graphique, effaçant le graphique courant si nécessaire.

7.1.1 La fonction plot()

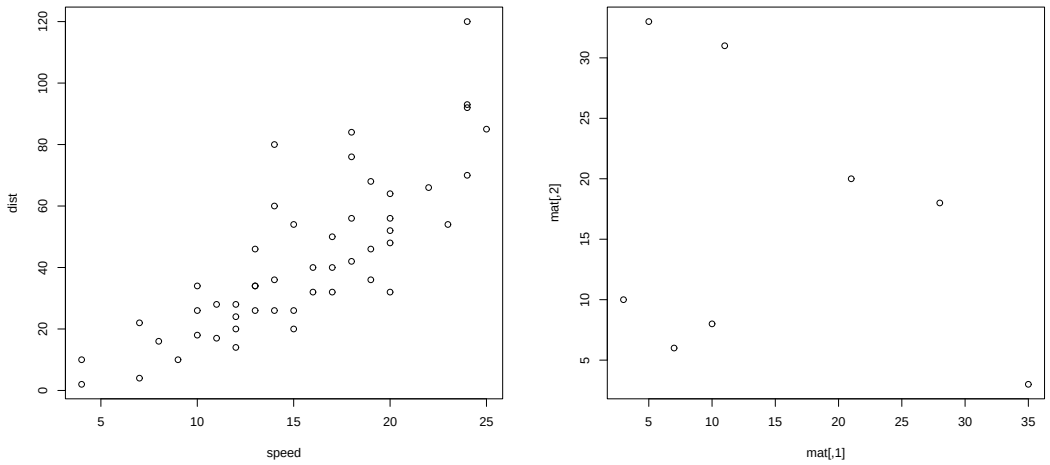
Une des fonctions graphiques les plus utilisées en **R** est `plot()`. C'est une fonction *générique* : le type de graphique produit dépend du type ou de la classe du premier argument. En effet, la fonction `plot()` est une fonction qui accepte tous les types d'objets et de ce fait peut engendrer des graphiques très diversifiés.

- `plot(x,y)`
  `plot(xy)`

  Si '`x`' et '`y`' sont des vecteurs de même taille, `plot(x,y)` produit un graphique avec les coordonnées de '`x`' en abscisse et les coordonnées de '`y`' en ordonnées. On peut obtenir le même résultat en fournissant un seul argument *(seconde forme)* qui soit une liste de deux éléments '`x`' et '`y`' ou une matrice à deux colonnes.

  Exemples :

```
> plot(speed,dist)
> mat <- matrix(c(5,3,7,10,35,28,21,11,33,10,6,8,3,18,20,31),ncol=2)
> plot(mat)
```

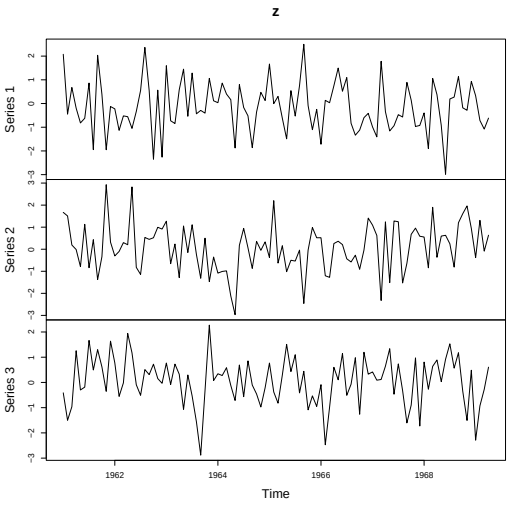


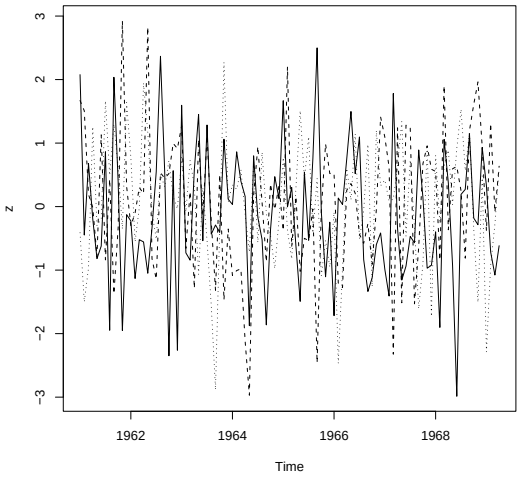
- `plot(x)`

  Si '`x`' est une série temporelle, cela produit un graphique série-temporelle, si '`x`' est un vecteur numérique, cela produit un graphique des coordonnées de '`x`' par rapport à leur indice dans le vecteur, et si '`x`' est un vecteur complexe, cela produit un graphique des parties imaginaires des éléments du vecteur par rapport aux parties réelles.

Exemples :

```
> z <- ts(matrix(rnorm(300),100,3), start=c(1961,1), frequency=12)
> plot(z)
> plot(z, plot.type="single", lty=1:3)
```



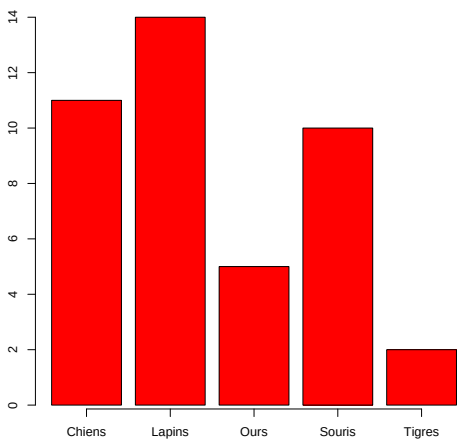


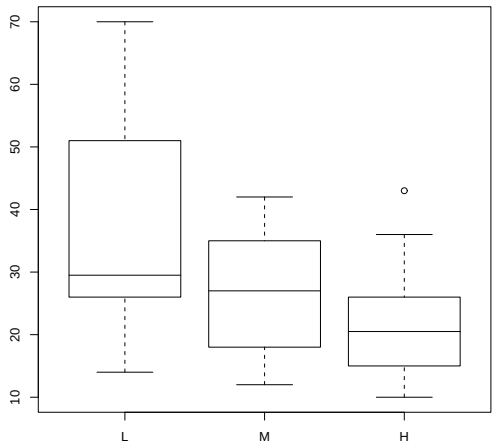
- `plot(f)`
  `plot(f,y)`

  '`f`' est un facteur, '`y`' un vecteur numérique. La première forme génère un diagramme en barres de '`f`' ; la seconde forme produit des diagrammes en boîtes de '`y`' pour chaque niveau de '`f`'.

  Exemples :

```
> f_factor(c(rep("Chiens",11),rep("Lapins",14),rep("Ours",5),
              rep("Souris",10),rep("Tigres",2)))
> plot(f)
> plot(tension,breaks)  # tension est un facteur
```



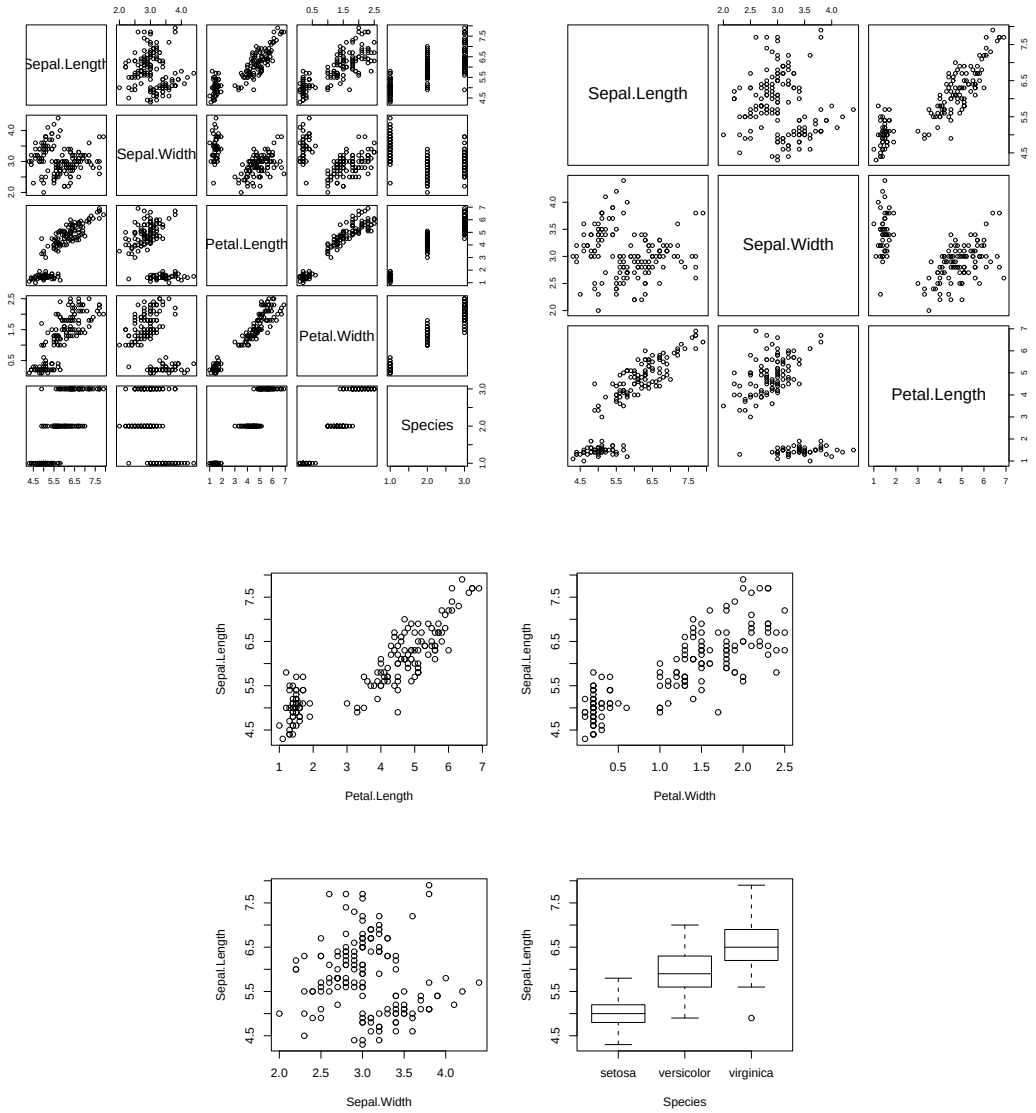


- `plot(jd)`
  `plot(~ expr)`
  `plot(y ~ expr)`

  '`jd`' est un jeu de données, '`y`' est un objet, `expr`' est une liste de noms d'objets séparée par '`+`' (e.g., `a+b+c`). Les deux premières formes produisent des graphiques des distributions des variables dans le jeu de données *(première forme)* ou d'un certain nombre d'objets nommés *(seconde forme)*. La troisième forme engendre les valeurs de '`y`' par rapport à chaque objet nommé dans '`expr`'.

  Exemples :

```
> plot(iris)
> attach(iris)
> plot(~Sepal.Length+Sepal.Width+Petal.Length)
> par(mfrow=c(2,2))
> plot(Sepal.Length~Petal.Length+Petal.Width+Sepal.Width+Species)
```



7.1.2 Graphiques de données multivariées

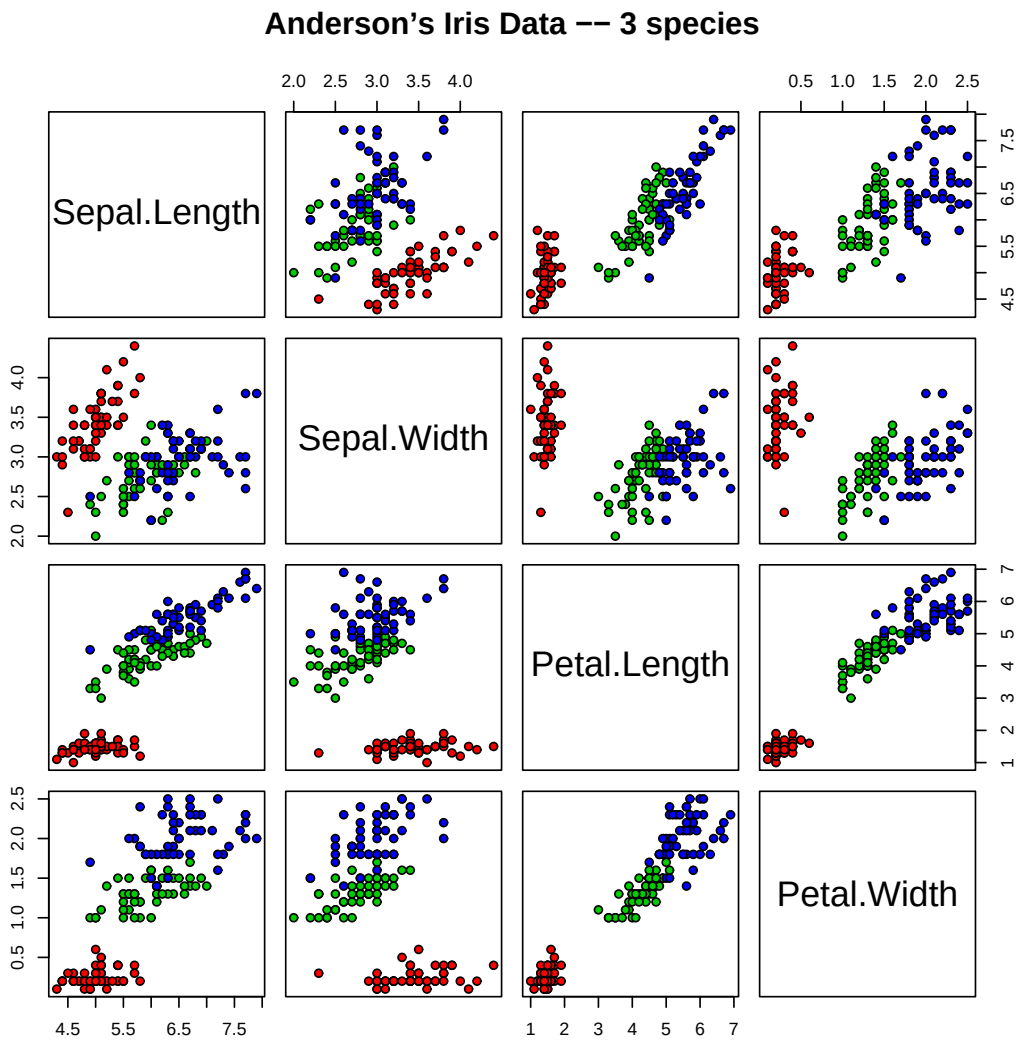
R fournit deux fonctions très utiles pour représenter des données multivariées. Si 'X' est une matrice numérique ou un jeu de données, la commande

```
> pairs(X)
```

produit une matrice des diagrammes de dispersion définie par les colonnes de 'X', c'est à dire que chaque colonne de 'X' est représentée par rapport à chaque autre colonne de 'X' et les *n(n-1)* graphiques sont placés dans une matrice. Le nom de *pairs* vient du fait qu'il y a un diagramme de dispersion pour chaque paire de colonnes de 'X'.

Exemple :

```
> pairs(iris[1:4], main = "Anderson's Iris Data -- 3 species",
      pch = 21, bg = c("red", "green3", "blue")[codes(iris$Species)])
```



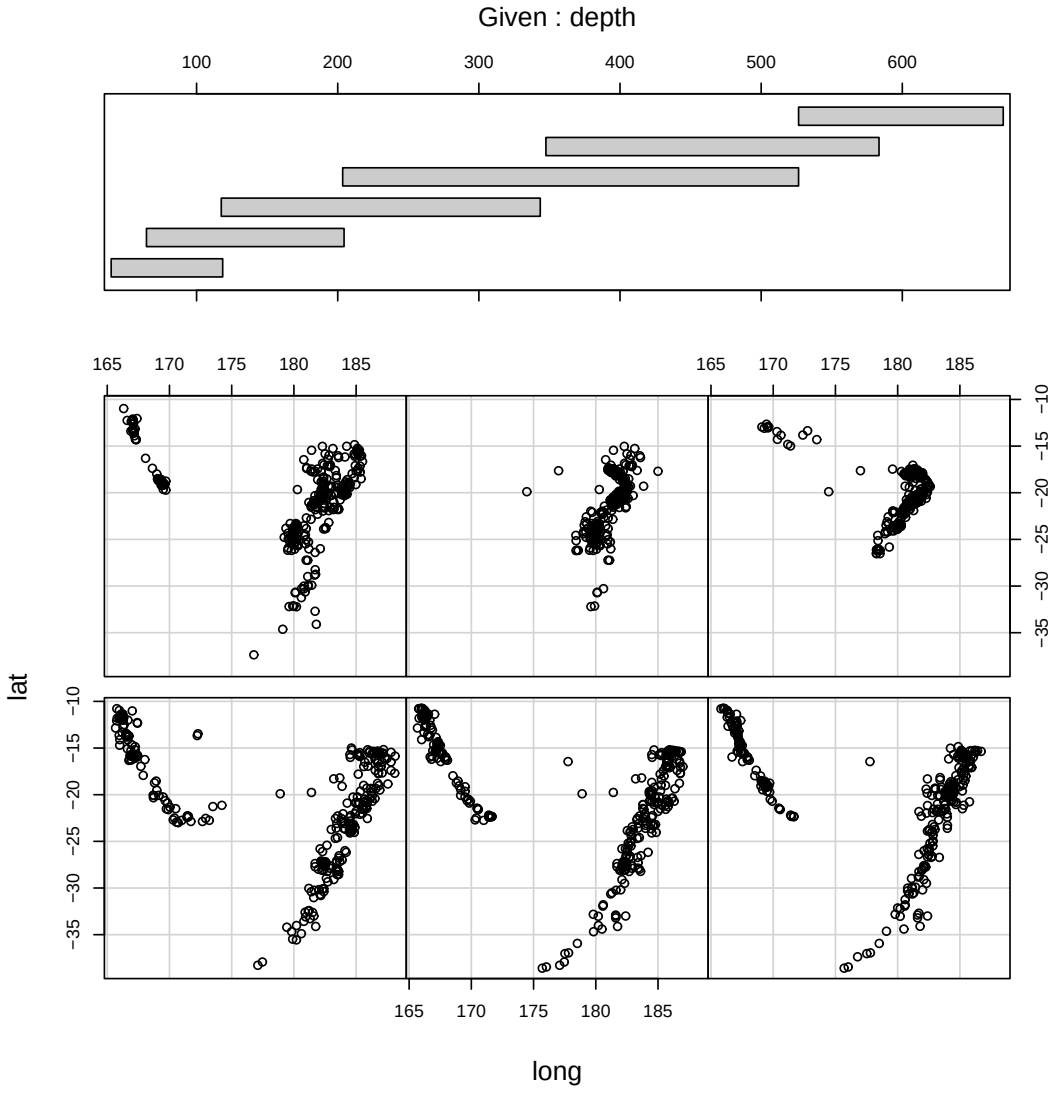
Quand on travaille avec trois ou quatre variables, un *coplot* peut être plus enrichissant. si ‘a’ et ‘b’ sont deux vecteurs numériques et ‘c’ est un vecteur numérique ou un facteur (tous de la même longueur), alors la commande

```
> coplot(a ~ b | c)
```

produit des graphiques de ‘a’ par rapport à ‘b’ pour des valeurs de ‘c’ données. Si ‘c’ est un facteur, cela signifie simplement que ‘a’ est représenté par rapport à ‘b’ pour chaque niveau de ‘c’. Quand ‘c’ est numérique, il est divisé en un certain nombre d’*intervalles de conditionnement* et ‘a’ est représenté par rapport à ‘b’ pour chaque intervalle de conditionnement de ‘c’. Le nombre et la position des intervalles peuvent être contrôlées par l’argument `given.values` de `coplot()`. La fonction `co.intervals()` est utile pour sélectionner les intervalles.

Exemple :

```
> coplot(lat ~ long | depth, data = quakes)
```



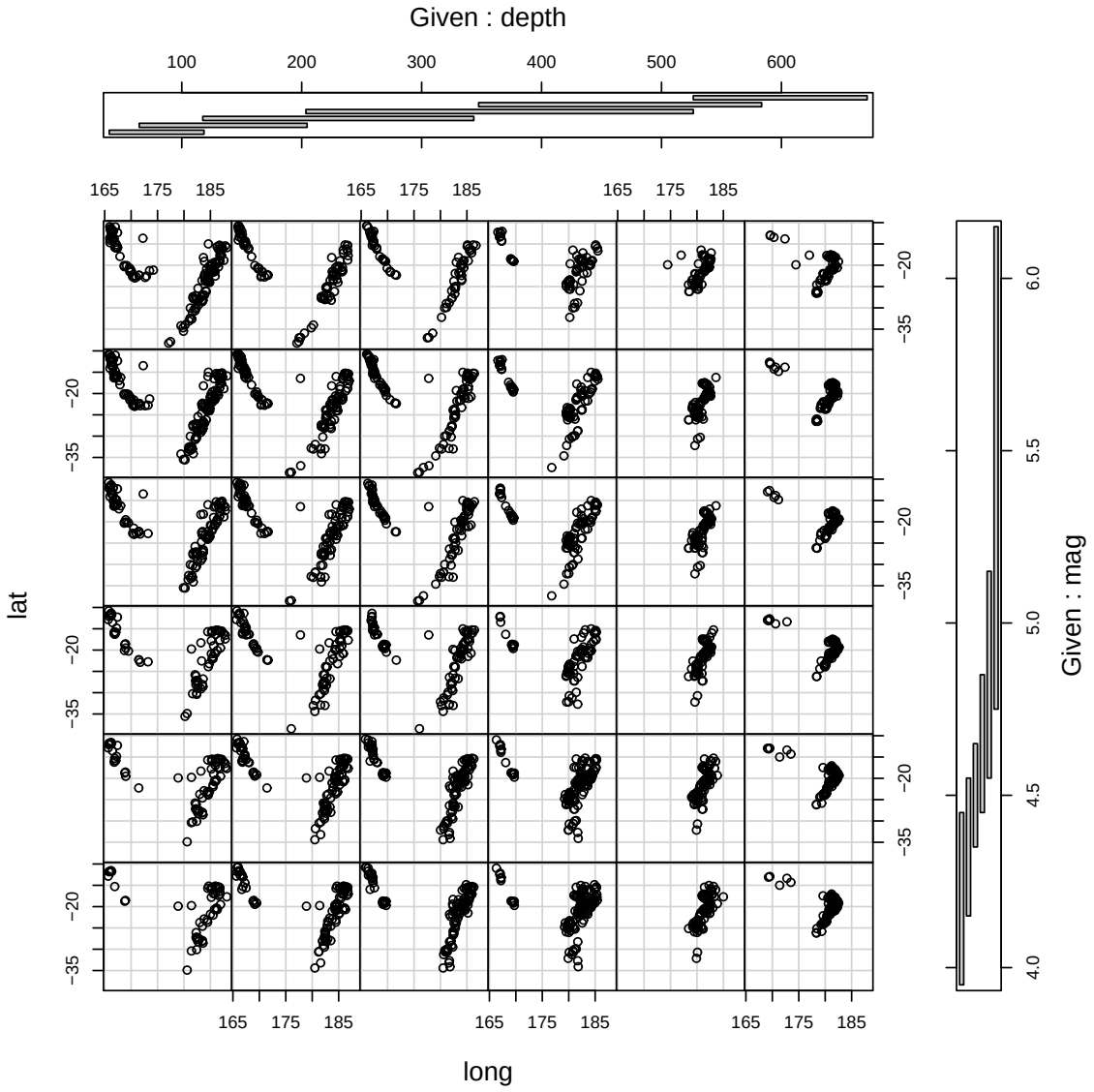
Il est aussi possible de donner deux variables *conditionnantes* avec une commande comme

```
> coplot(a ~ b | c + d)
```

qui produit des graphiques de dispersion de '`a`' par rapport à '`b`' pour chaque intersection des intervalles de conditionnement de '`c`' et '`d`'.

Exemple :

```
>  coplot(lat ~ long | depth * mag, data = quakes)
```



Les fonctions `coplot()` et `pairs()` prennent un argument `panel` qui peut être utilisé pour définir le type de graphique qui apparaît dans chaque cadre. Par défaut c'est `points()` pour produire un nuage de points, mais en fournissant d'autres fonctions graphiques de bas niveau à `panel`, il est possible de représenter n'importe quel type de graphique. Un exemple de fonction utile pour les *coplot* est `panel.smooth()`.

7.1.3 Graphiques particuliers

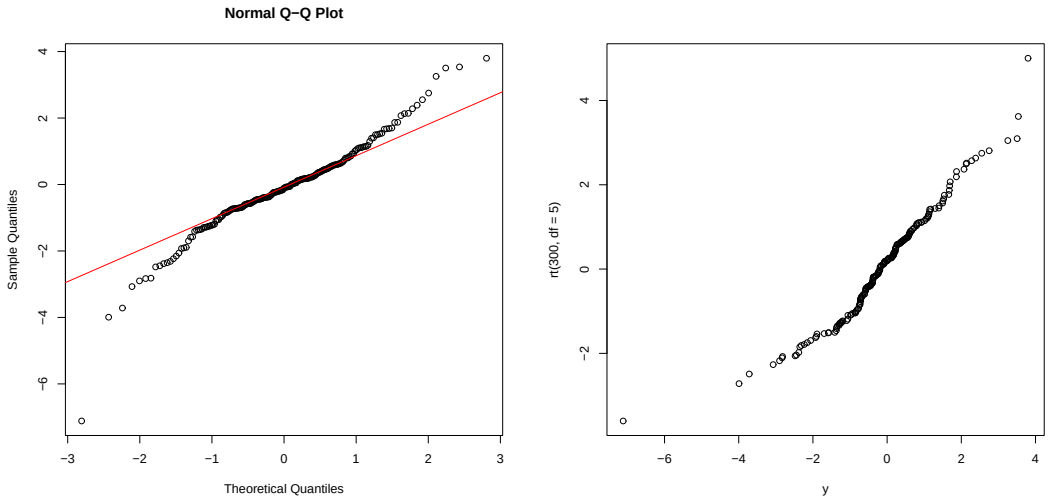
D'autres fonctions graphiques de haut niveau produisent des types de graphiques différents.
Quelques exemples :

- `qqnorm(x)`
  `qqline(x)`
  `qqplot(x,y)`

  Graphiques de comparaison de distributions. La première forme trace le vecteur numérique '`x`' par rapport aux résultats attendus d'une loi normale *(graphique qq)* et la deuxième ajoute une ligne à ce graphique qui passe à travers les quartiles de la distribution et des données. La troisième forme représente les quartiles de '`x`' par rapport à ceux de `y` pour comparer leur distributions respectives.

  Exemples :

```
> y <- rt(200, df = 5)
> qqnorm(y); qqline(y, col = 2)
> qqplot(y, rt(300, df = 5))
```

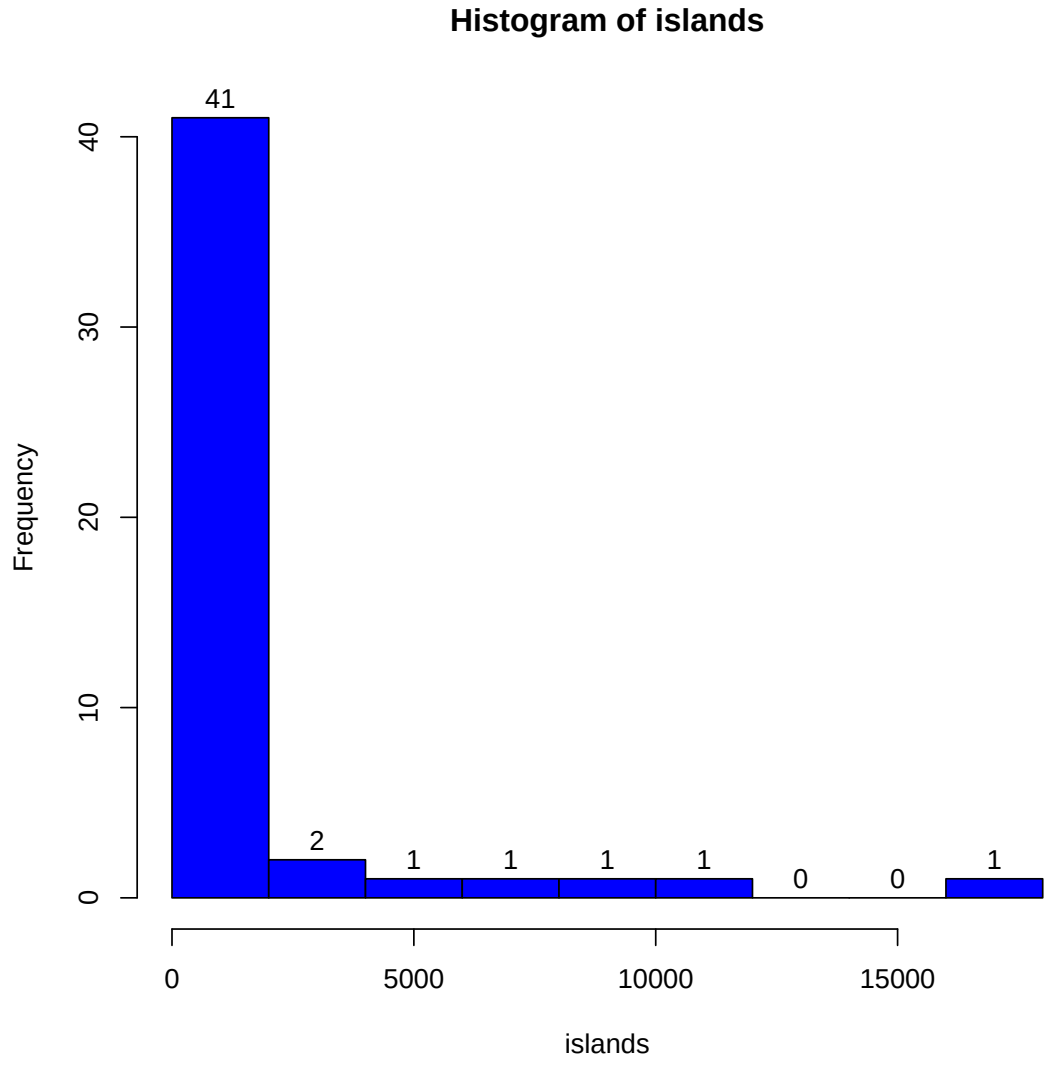


- `hist(x)`
  `hist(x, nclass=n)`
  `hist(x, breaks=b,...)`

  Produit l'histogramme du vecteur numérique '`x`'. Un nombre de classes adéquat est généralement choisi, mais on peut donner fixer une valeur par l'intermédiaire de l'argument `nclass`. On peut également choisir de donner les points de rupture avec l'argument `breaks`. Si `probability=TRUE` est donné, les barres représentent les fréquences relatives au lieu des comptages.

  Exemple :

  ```
  > hist(islands, col="blue", labels = TRUE)
  ```

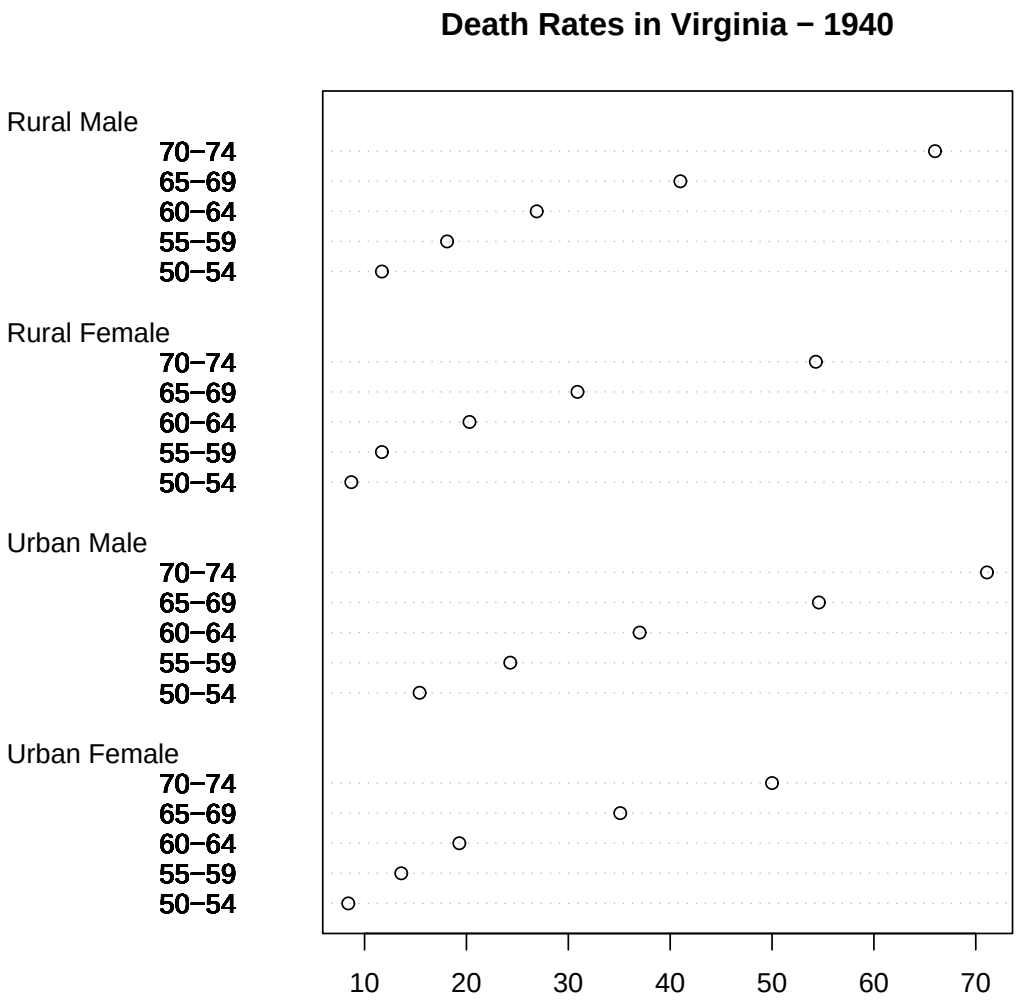


- `dotchart(x,...)`

  Construit un graphique de points des données de '`x`'. Dans un graphique de points, on a en ordonnée les labels des données de '`x`' et les valeurs en abscisse. Par exemple cela permet une sélection visuelle facile de données dont les valeurs sont dans une certaine fourchette.

  Exemple :

```
> dotchart(VADeaths, main = "Death Rates in Virginia - 1940")
```

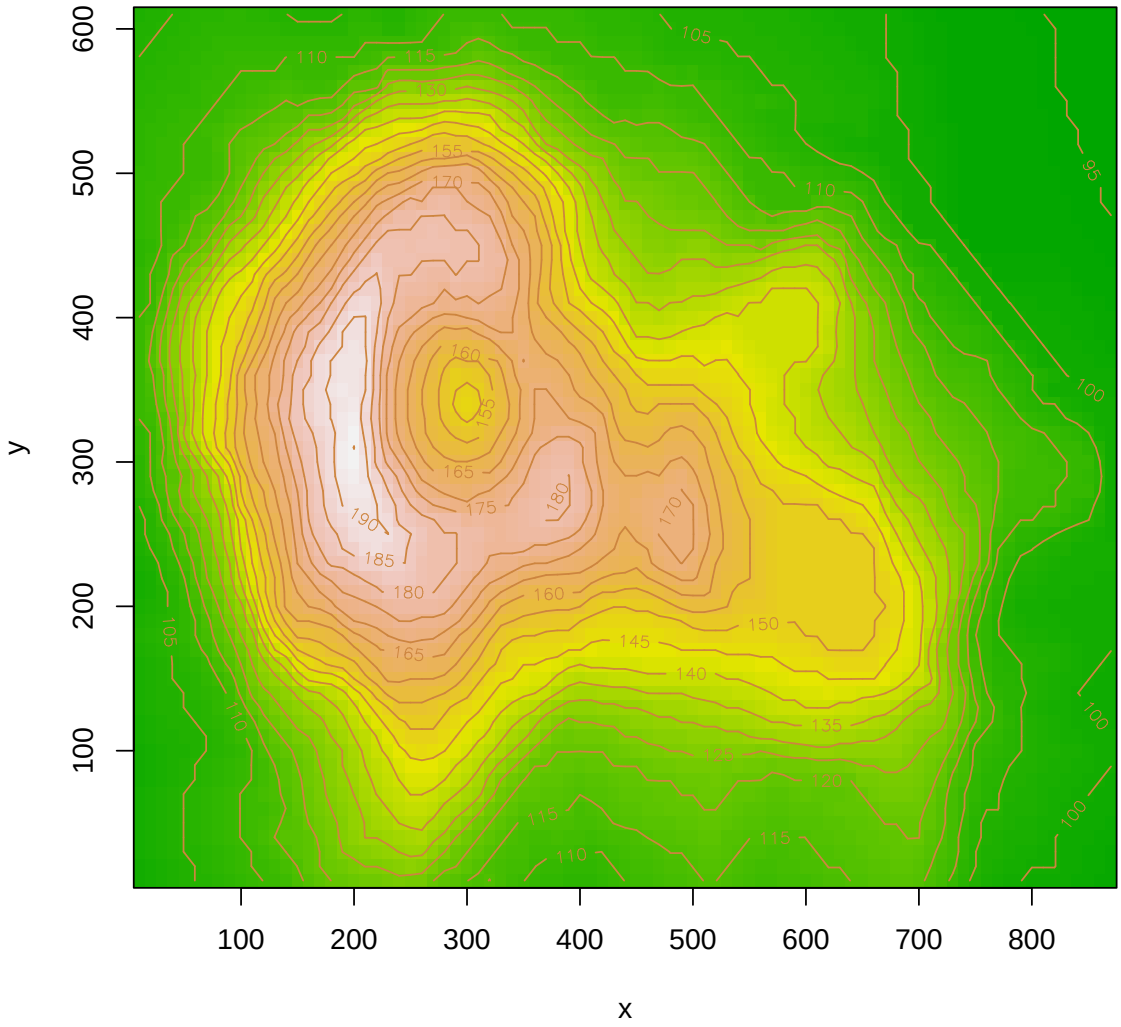


- `image(x,y,z,...)`
  `contour(x,y,z,...)`
  `persp(x,y,z,...)`

  Graphiques tridimensionnels. le graphique *image* dessine une grille de rectangles de différentes couleurs pour représenter les valeurs de '`z`', le graphique *contour* dessine des courbes de niveau pour représenter les valeurs de '`z`', et *persp* dessine une surface en trois dimensions.

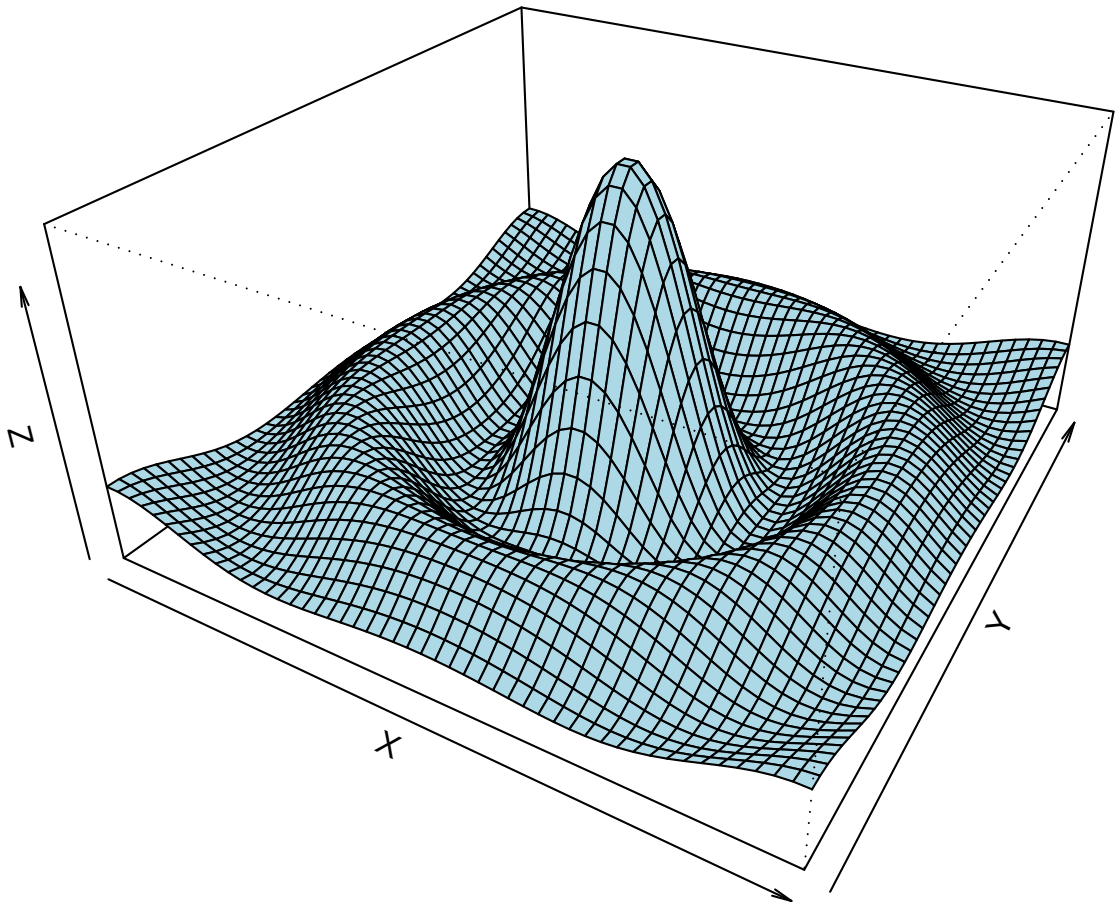
  Exemple de graphique *image* et *contour* :

```
> x <- 10*(1:nrow(volcano))
> y <- 10*(1:ncol(volcano))
> image(x, y, volcano, col = terrain.colors(100), axes = FALSE)
> contour(x, y, volcano, levels = seq(90, 200, by=5), add = TRUE,
          col = "peru")
> axis(1, at = seq(100, 800, by = 100))
> axis(2, at = seq(100, 600, by = 100))
> box()
```



Autre exemple de graphique *persp* :

```
> x <- seq(-10, 10, length=50)
> y <- x
> f <- function(x,y)
+ {
+     r <- sqrt(x^2+y^2)
+     10 * sin(r)/r
+ }
> z <- outer(x, y, f)
> z[is.na(z)] <- 1
> par(bg = "white")
> persp(x,y,z,theta = 30, phi = 30, expand = 0.5, col = "lightblue",
+       xlab = "X", ylab = "Y", zlab = "Z")
```



7.1.4 Arguments des fonctions graphiques de haut niveau

Un grand nombre d'arguments peuvent être passés aux fonctions graphiques de haut niveau :

- `add=TRUE`

  Oblige la fonction à agir comme une fonction graphique de bas niveau, superposant le graphique sur le graphique courant (seulement pour certaines fonctions).

- `axes=FALSE`

  Ne génère pas les axes. Ceci est utile pour ajouter des axes personnalisés avec la fonction `axis()`. Par défaut , nous avons `axes=TRUE`, les axes sont générés.

- `log="x"`
  `log="y"`
  `log="xy"`

  Les abscisses, les ordonnées ou les deux sont logarithmiques. Ceci marchera pour beaucoup de fonctions (pas toutes...).

- `type`

  L'argument `type` contrôle le type de graphique produit, de la façon suivante :

  - `type="p"` Dessine des points individuels (par défaut)
  - `type="l"` Dessine des lignes
  - `type="b"` Dessine des points connectés par des lignes *both*
  - `type="p"` Dessine des points traversés par des lignes.
  - `type="h"` Dessine des points reliés à l'axe des ordonnées par des lignes verticales *(high-density)*
  - `type="s"` Représente des fonctions en escalier
  - `type="S"` Représente des fonctions en escalier
  - `type="n"` Ne dessine rien. Cependant les axes sont générés (par défaut) et le système de coordonnées est mis en place en fonction des données. Cela est idéal pour créer des graphiques avec les fonctions graphiques de bas niveau par la suite.

- `xlab=string`
  `ylab=string`

  Légendes des axes des abscisses et des ordonnées. utilisation des arguments pour changer les légendes par défaut, qui sont habituellement les noms des objets dans l'appel de la fonction.

- `main=string`

  Titre de la figure, placé au dessus de cette dernière, en grosses lettres.

- `sub=string`

  Sous-titre, placé juste sous l'axe des abscisses dans une police plus petite.

7.2 Les fonctions graphiques de bas niveau

Parfois les fonctions graphiques de haut niveau ne produisent pas exactement le type de graphique que l'on recherche. Dans ce cadre, **les fonctions graphiques de bas niveau** peuvent être utilisées pour **rajouter de l'information** (comme des points, des lignes, du texte) au graphique courant.

Parmi les plus utiles de ces fonctions graphiques de bas niveau on a :

- `points(x,y)`
  `lines(x,y)`

  Rajoute des points ou des lignes au graphique courant. L'argument `type` de `plot()` peut également être passé à ces fonctions (par défaut il est à `"p"` pour `points()` et à `"l"` pour `lines()`.)

- `text(x,y,labels,...)`

  Ajoute du texte au graphique aux points données par '`x`' et '`y`'. Normalement `labels` est un vecteur de caractères ou d'entiers, et `labels[i]` est ajouté à la position `(x[i],y[i])`.
  Cette fonction est souvent utilisée de la façon suivante :

  ```
  > plot(x, y, type="n"); text(x, y, names)
  ```

  Le paramètre graphique `type="n"` supprime les points mais dessine les axes, et la fonction `text()` dessine les caractères spéciaux donnés par `names` aux emplacements des points.

- `abline(a,b)`
  `abline(h=y)`
  `abline(v=x)`
  `abline(lm.obj)`

  Rajoute une ligne de pente '`b`' et d'ordonnée à l'origine '`a`' au graphique courant. '`h=y`' peut être utilisé pour spécifier le point de coordonnée '`y`', la hauteur d'une droite horizontale qui passe sur le graphique. '`v=x`' similairement pour le point de coordonnée '`x`' d'une droite verticale. De plus, '`lm.obj`' peut être une liste avec un composant de longueur deux correspondant aux coefficients, ces derniers sont l'ordonnée à l'origine et la pente dans cet ordre.

- `polygon(x,y,...)`

  Engendre un polygone défini par les vecteurs ordonnés (`x`, `y`) et (optionnellement) le remplit de hachures.

- `legend(x,y,legend,...)`

  Ajoute une légende au graphique courant à la position spécifiée. Les caractères utilisés pour les points, les styles de lignes etc. sont identifiés par les labels du vecteur `legend`. Au moins un argument '`v`' (de la même longueur que `legend`) avec les valeurs correspondantes doit être donné :

  - `legend( , fill=v)` Couleurs pour des remplissages
  - `legend( , col=v)` Couleurs des points ou lignes

  - `legend( , lty=v)` Styles des lignes
  - `legend( , lwd=v)` Epaisseurs des lignes
  - `legend( , pch=v)` Type des points (vecteur de caractères)

- `title(main,sub)`

  Ajoute un titre `main` en haut du graphique courant en grande police et (optionnellement) un sous-titre `sub` en bas avec une police plus petite.

- `axis(side,...)`

  Ajoute un axe au graphique courant sur le côté donné par le premier argument (1 à 4 en partant du bas dans le sens des aiguilles d'une montre.) Les autres arguments contrôlent la position de l'axe, les marques et les labels. Ceci est utile pour ajouter des axes personnalisés en utilisant `plot()` avec l'argument `axes=FALSE`.

Les fonctions graphiques de bas niveau nécessitent en général des informations de position (e.g., les coordonnées 'x' et 'y') pour déterminer ou placer les nouveaux éléments graphiques. Les coordonnées sont alors des *coordonnées utilisateur* qui sont définies par les fonctions graphiques de haut niveau utilisées précédemment et choisies en fonction des données fournies.
Quand les arguments '`x`' et '`y`' sont nécessaires, il est également possible de fournir une liste avec deux éléments `x` et `y`. On peut également fournir une matrice avec deux colonnes. Ainsi des fonctions comme `locator()` (voir ci-dessous) peuvent servir à donner des positions de façon interactive.

7.3 Fonctions graphiques interactives

R fournit des fonctions qui permettent aux utilisateurs **d'extraire ou d'apporter de l'information** à un graphique en utilisant la souris.

On utilise la fonction `locator()` :

```
locator(n,type)
```

Cette fonction attend que l'utilisateur sélectionne un ou plusieurs points sur le graphique en utilisant le bouton gauche de la souris. Cela est répété '`n`' fois jusqu'à ce que l'utilisateur clique sur un autre bouton de la souris *(Unix, Windows)* ou hors de la fenêtre graphique *(Mac)*. L'argument '`type`' permet d'ajouter des points ou des lignes aux endroits sélectionnés sur le graphique. Par défaut, il n'y a pas l'ajout. Cette fonction retourne les coordonnées des points sélectionnés dans une liste comportant deux composants '`x`' et '`y`'.

`locator()` est généralement appelé sans argument. C'est particulièrement utile de sélectionner de façon interactive des positions pour les éléments graphiques tels que la légende, les étiquettes quand il est difficile de les calculer.
Par exemple, pour placer un texte à côté d'un point, la commande peut être utilisé.

```
> text(locator(1),"texte",agj=0)
```

Une autre fonction :

```
identify(x,y,labels)
```

Cette fonction permet à l'utilisateur de mettre en valeur des points correspondants aux valeurs de `labels` (en utilisant le bouton gauche de la souris). Ces points sont définis par '`x`' et '`y`' (si `labels` n'est pas spécifié, il apparaît l'indice du point sélectionné). Pour sortir du mode interactif, on clique sur un autre bouton de la souris. Cette fonction retourne les indices des points sélectionnés.

7.4 Les paramètres graphiques

Lorsqu'on crée des graphiques, **R** ne réalise pas toujours le résultat attendu. Mais il est possible de **personnaliser** cela en utilisant les **paramètres graphiques**. **R** propose une large gamme de paramètres qui contrôlent différents aspects tels que les axes, la légende, les couleurs, les étiquettes, ... Chaque paramètre est représenté par un nom et une valeur. (Exemple : Pour le paramètre des couleurs, le nom est `col` et la valeur une couleur)

La fonction `par()` est utilisé pour accéder et modifier les paramètres graphiques pour le graphique courant.

- `par()`

  Sans arguments, cette fonction retourne la liste de tous les paramètres graphiques ainsi que leurs valeurs.

- `par(c("col","lty"))`

  Avec un vecteur de caractère en argument, il est seulement retourné la liste des paramètres nommés.

- `par(col=4,lty=2)`

  Avec des arguments correspondants à des valeurs, on affecte les valeurs voulues aux paramètres.

En passant les paramètres graphiques avec la fonction `par()` la nouvelle valeur du paramètre sera prise en compte de façon permanente pour la fenêtre graphique courante.

Les paramètres graphiques peuvent aussi être passés par les fonctions graphiques comme des arguments. Cela a le même effet que s'ils étaient passés avec `par()` excepté qu'ils n'affectent que la fonction considérée.

Exemple :

```
plot(x,y,pch="+")
```

Donne un nuage de points utilisant le signe '`+`' pour représenter les points, sans changer la construction par défaut des prochains graphiques.

L'aide de la fonction `par()` fournit un résumé de l'ensemble des paramètres graphiques.
- Les paramètres graphiques contrôlent la façon dont les éléments graphiques (points, lignes, texte, polygones) sont dessinés.

Paramètres utilisés :

`pch lty lwd col font font.axis font.lab font.main font.sub adj cex`

- De plus, il existe des paramètres pour représenter les axes des graphiques qui définissent le type de ligne utilisé, l'échelle utilisée, les étiquettes.

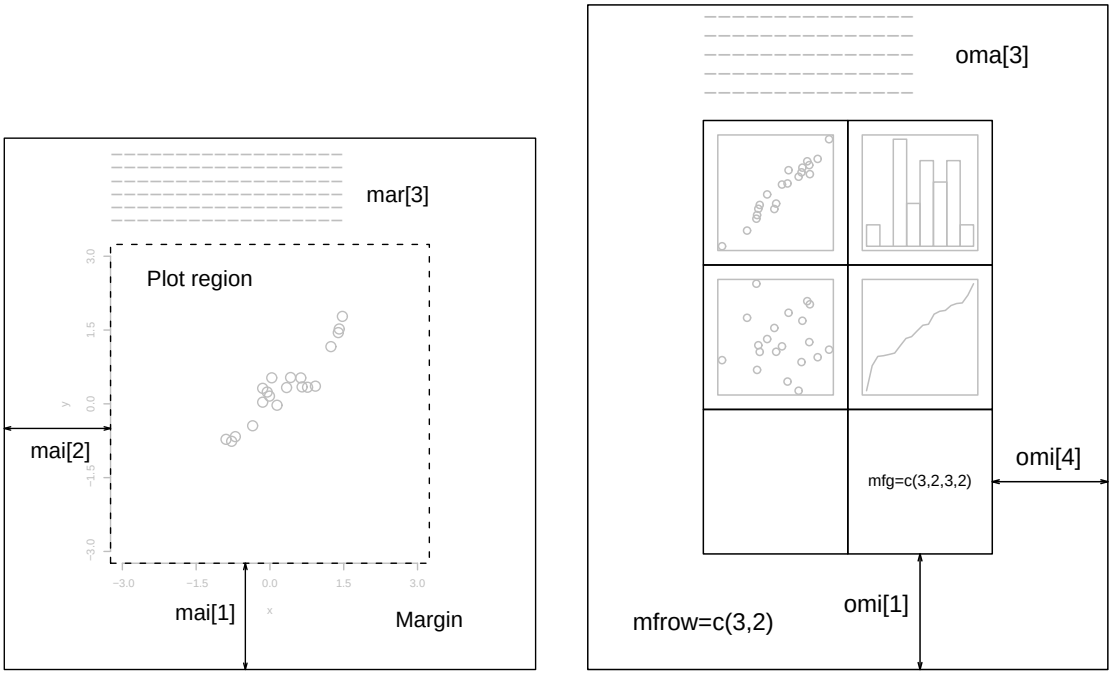
Paramètres utilisés :

`lab las mgp tck xaxs yaxs`

- Il existe aussi des paramètres permettant d'ajuster les marges à une figure et des paramètres permettant de créer un tableau de 'n' figures sur une même page.

Paramètres utilisés :

`mai mar mfcom mfrow mfg fig oma omi`



La fonction `mtext()` avec l'argument `outer=TRUE` permet d'ajouter du texte à l'extérieur des marges. Pour les arrangements plus complexes de plusieurs figures, les fonctions `split.screen()` et `layout()` sont utilisées.

7.5 Fonctions liées aux fenêtres graphiques

La démonstration graphique incluse dans **R** permet de visualiser une grande partie des possibilités graphiques du logiciel. (Commande : `demo(graphics)`)

Les différentes fonctions :

- La fonction `X11()` permet d'ouvrir une fenêtre graphique (vu précédemment) et la fonction `dev.off()` ferme la fenêtre courante ou une fenêtre spécifiée.
- Les fonctions `postscript()` et `pictex()` ouvrent un nouveau mode graphique qui permet de sauvegarder et d'imprimer des fichiers graphiques au format `.ps .eps` et `.tex`.
- La fonction `dev.list()` retourne le nombre et le nom de tous les modes graphiques utilisés.
- Les fonctions `dev.next()` et `dev.prev()` retournent le numéro et le nom du mode graphique suivant ou précédant au mode graphique courant.
- La fonction `dev.set(which=k)` peut être utilisé pour changer de mode graphique courant en utilisant un numéro de position dans la liste des modes. Cette fonction retourne le nombre et le nom du mode graphique sélectionné.
- Les fonctions `dev.copy(device,...,which=k)`, `dev.print(device,...,which=k)` font une copie de la fenêtre graphique '`k`'. `device` est une fonction de mode graphique et d'autres arguments '`...`' sont spécifiés si besoin. Mais `dev.print()` ferme immédiatement la copie de cette dernière et l'imprime.
- La fonction `dev.copy2eps()` crée un fichier image au format `.eps` du graphique présent dans la fenêtre courante. Par défaut, le nom de ce fichier est '`Rplot.eps`'. Cette commande est fort utile pour sauvegarder un graphique dans un fichier `.eps` et ensuite pour l'intégrer à l'élaboration d'un document en LaTeX.
- La fonction `graphics.off()` ferme toutes les fenêtres graphiques.

Certains modules créés comportent des fonctions graphiques et permettent de réaliser de nouveaux types de graphiques avec de nouveaux modes de fonctionnement.
Par exemple :

- Le module **lattice** permet des nouvelles représentations ainsi que des représentations en trois dimensions.
- Le module **xgobi** permet de réaliser des graphiques dynamiques interactifs.

Chapitre 8

Les statistiques

Il existe une importante différence de philosophie entre **R** et les principaux autres logiciels statistiques. En **R** une analyse statistique se fait en une **série d'étapes avec stockage des résultats intermédiaires** dans des objets. Ceci est identique pour **S**. Par opposition, des logiciels comme **SAS** et **SPSS** donneront des **sorties exhaustives** pour une régression ou une analyse discriminante alors que **R** donnera un minimum de sorties et stockera les résultats dans un objet pour consultation ultérieure à l'aide d'autres fonctions.

8.1 Les jeux de données

Pour se familiariser aux statistiques dans **R**, il est proposé plusieurs jeux de données qui sont inclus dans le logiciel ou dans les modules utilisés. Contrairement à **S-Plus**, ces jeux de données doivent être chargés de façon explicite en utilisant la fonction `data()`. Cette fonction appelée sans argument affiche l'ensemble des jeux de données disponibles. Lorsqu'on l'appelle avec un nom, elle charge le jeu de données correspondant.

Exemples :

```
> data()
Data sets in package 'base':

Formaldehyde            Determination of Formaldehyde concentration
HairEyeColor            Hair and eye color of statistics students
InsectSprays            Effectiveness of insect sprays
LifeCycleSavings        Intercountry life-cycle savings data
OrchardSprays           Potency of orchard sprays
PlantGrowth             Results from an experiment on plant growth
Titanic                 Survival of passengers on the Titanic
...

> data(HairEyeColor)
```

Dans la plupart des cas, ceci chargera un objet **R** du même nom , en général un « data frame ». Cependant, dans quelques cas, ceci chargera plusieurs objets, il faut regarder l'aide en ligne sur les données pour savoir.

De plus, cette aide apportera des informations supplémentaires sur le jeu de données. Pour cela, il suffit d'appeler l'aide à partir du nom du jeu.

```
> data(faithful)
> help(faithful)
faithful                package:base                R Documentation

Old Faithful Geyser Data

Description:

     The 'faithful' data frame has 272 rows and 2 columns; the waiting
     time between eruptions and the duration of the eruption for the
     Old Faithful geyser in Yellowstone National Park, Wyoming, USA.
...
```

Cela donne des indications générales sur les données (nombre, description, source, références) et le type des variables utilisées.

Pour accèder aux données d'autres modules, on utilise l'argument `package`. Exemple :

```
data(package="nls")
data(Puromycin, package="nls")
```

Si un module a été attaché par `library()`, ses jeux de données sont automatiquement inclus dans la recherche, de telle sorte que :

```
library(nls)
data()
data(Puromycin)
```

fera la liste de tous les jeux de données dans les modules attachés et chargera ensuite le jeu de données `Puromycin` à partir du premier module dans lequel ce jeu de données est trouvé.

Il est possible de modifier les données (de la forme « data frame ») de façon graphique avec la fonction `fix()`, appelée avec le nom du jeu.

```
> fix(faithful)
```

Cela ouvre une interface comportant un tableau de données, où chacune des données sont accessibles à l'aide de la souris et sont modifiables. Bien sûr, cela est aussi possible pour ses propres données.

D'autre part, il est possible de créer ou d'importer ses propres jeux de données comme cela a été vu dans le paragraphe *« Importation et exportation de données »*. L'utilisateur travaille sur des données extérieures au système pratiquement tout le temps. Ces données fournies par le logiciel servent le plus souvent de « matière »pour découvrir et apprendre les différentes possibilités de **R**.

8.2 La définition des modèles statistiques : les formules

Dans **R**, les formules permettent d'engendrer les modèles statistiques désirés pour la plupart des fonctions.
L'opérateur '`~`' permet de définir le modèle, la forme pour un modèle linéaire ordinaire est :

```
response ~ op_1 terme_1 op_2 terme_2 op_3 terme_3 ...
```

Où

- `response` est un vecteur ou une matrice définissant la variable réponse (variable à expliquer)
- `op_i` est un opérateur, '`+`' ou '`-`' indiquant l'inclusion ou l'exclusion d'un terme (d'une variable) dans le modèle.
- `terme_i` peut être un vecteur, une matrice, 1, un facteur ou une formule comportant des facteurs, vecteurs ou matrices reliés par des opérateurs. (`terme_i` représente les variables explicatives)

Les différents opérateurs pour les formules sont :

- Y ~ M — Y est modélisé par M.
- M_1 + M_2 — Donne les effets différentiels de M_1 et M_2.
- M_1 - M_2 — Donne les effets de M_1 mais pas de M_2.
- M_1 : M_2 — Donne les effets d'interactions de M_1 avec M_2.
- M_1 %in% M_2 — est similaire à M_1 : M_2 mais avec un codage différent.
- M_1 * M_2 — Donne les effets différentiels et les effets d'interactions de M_1 avec M_2 (équivalent à M_1 + M_2 + M_1 : M_2).
- M_1 / M_2 — Donne l'effet différentiel de M_1 et l'effet d'interaction de M_2 avec M_1, ceci est équivalent M_1 + M_2 %in% M_1.
- M^n — Donne l'ensemble des effets d'interactions des termes de M jusqu'au niveau n.

8.3 Analyses exploratoires

Les différentes analyses exploratoires présentées dans cette partie correspondent à la statistique unidimensionnelle et à la statistique bidimensionnelle de variables quantitatives et qualitatives. De plus dans cette partie, on retrouve certaines représentations graphiques observées préalablement dans le chapitre sur les graphiques.

8.3.1 Statistique exploratoire unidimensionnelle

Variable quantitative

On a noté l'âge de 15 salariés d'une entreprise (arrondi à l'année) :
41,32,46,44,51,21,38,44,51,29,35,44,56,49,38
La fonction `sort()` permet de trier un jeu de données soit de façon croissante soit de façon décroissante.
Exemple :

```
> jd_c(41,32,46,44,51,21,38,44,51,29,35,44,56,49,38)
> sort(jd)
 [1] 21 29 32 35 38 38 41 44 44 44 46 49 51 51 56
```

La fonction `summary()` permet d'obtenir un résumé statistique élémentaire du jeu de données.
Exemple :

```
> summary(jd)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  21.00   36.50   44.00   41.27   47.50   56.00
```

La variance empirique est obtenue grâce à la fonction `var()`.
Exemple :

```
> var(jd)
[1] 87.06667
```

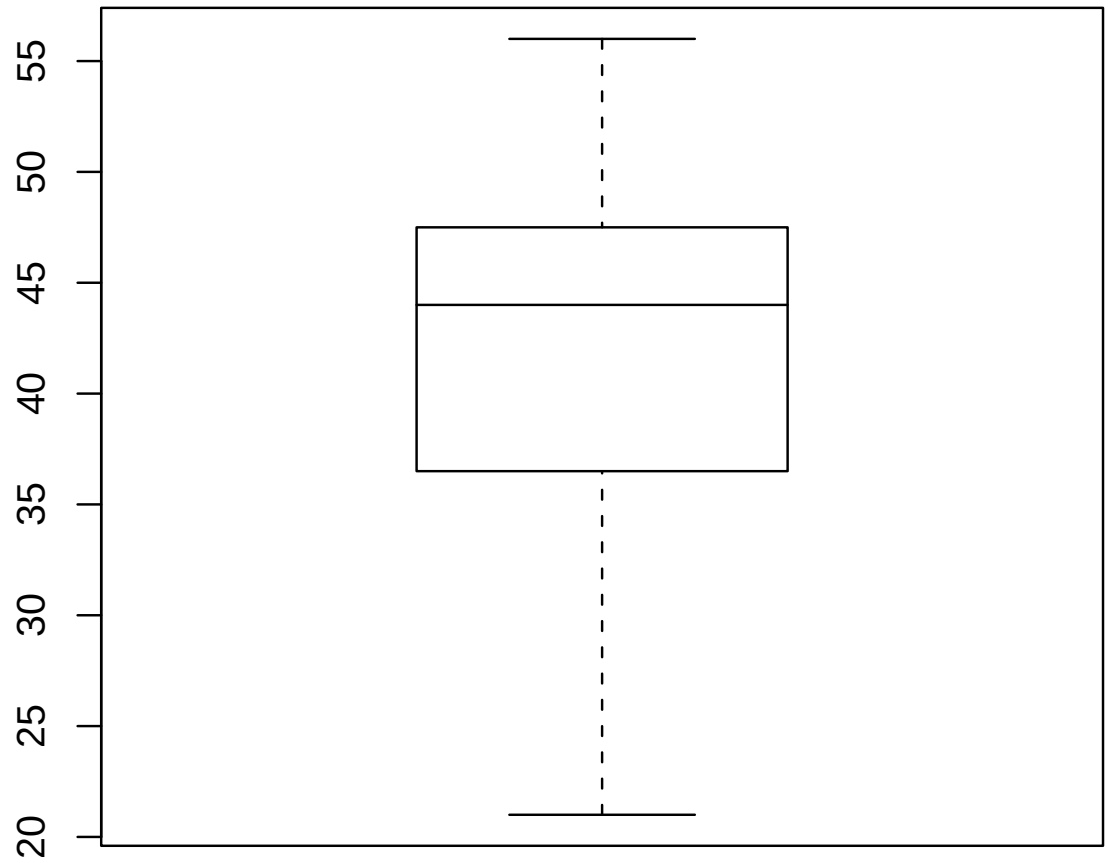
Présentation d'un jeu de données en tiges et feuilles (« Steam and leaf") grâce à la fonction `stem()`.

```
> stem(jd)

  The decimal point is 1 digit(s) to the right of the |

  2 | 19
  3 | 2588
  4 | 144469
  5 | 116
```

Les intérêts de cette présentation sont la simplicité, la prise en compte de l'ensemble des données et la visualisation de la distribution empirique.

La fonction `boxplot()` engendre un diagramme en boîte à partir d'un jeu de données. Exemple :

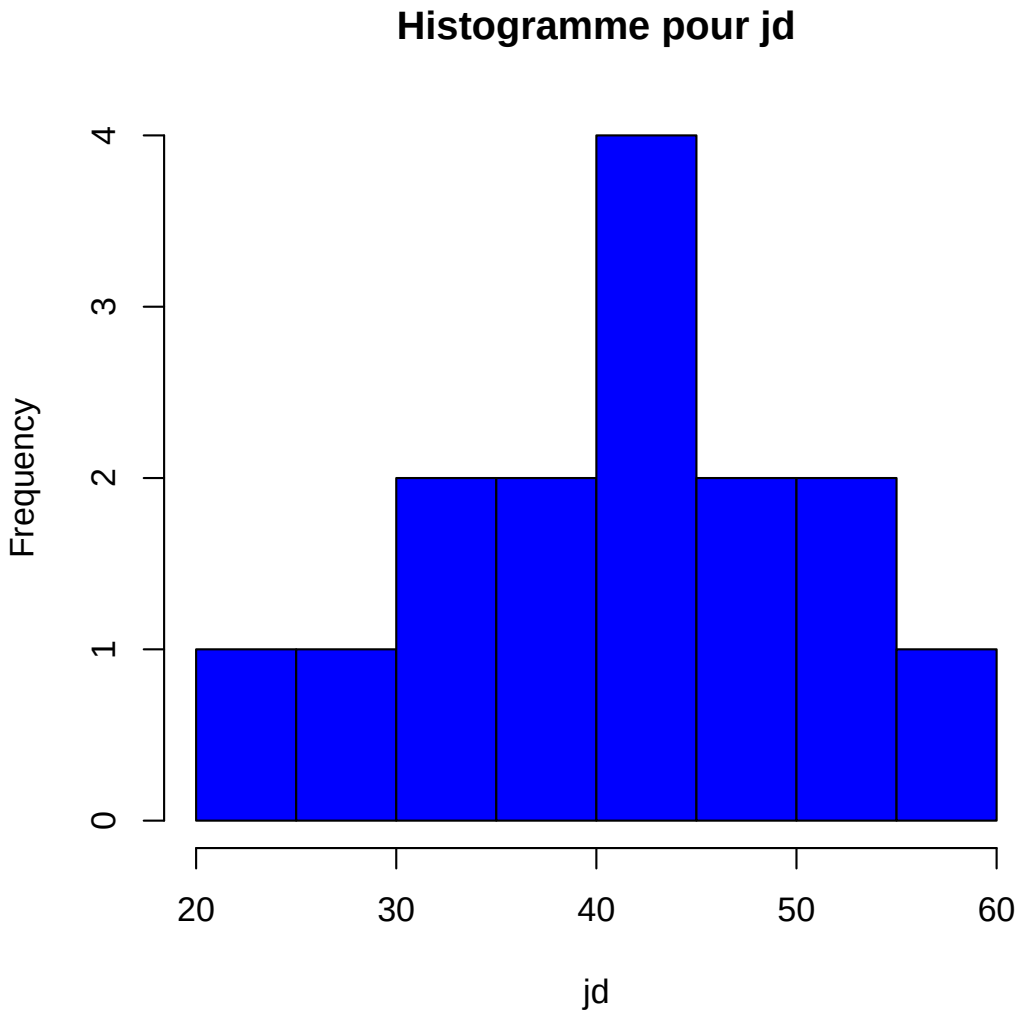
```
> boxplot(jd)
```



L'intérêt de ce diagramme en boîte est qu'il renseigne sur la dispersion des observations. Le corps de la boîte est construit à l'aide du premier quartile, de la médiane (second quartile) et du troisième quartile.
Ce diagramme fait apparaître les observations qui sont « loin" du « paquet central", ce sont elles qui agissent sur les résultats statistiques. Les avantages sont que ce diagramme est facile à interpréter, il est riche en informations. De plus, il est facilement comparable en disposant en « parallèle" plusieurs diagrammes.

La fonction `hist()` permet de réaliser un histogramme pour un jeu de données. Exemple :

```
> hist(jd,col="blue",main="Histogramme pour jd")
```



L'histogramme regroupe les données par souci de clarté dans la présentation. Il permet de mettre en évidence le caractère symétrique ou dissymétrique de la répartition des données. Mais surtout l'intérêt majeur de l'histogramme est de dégager la forme d'une fonction de répartition de probabilité.

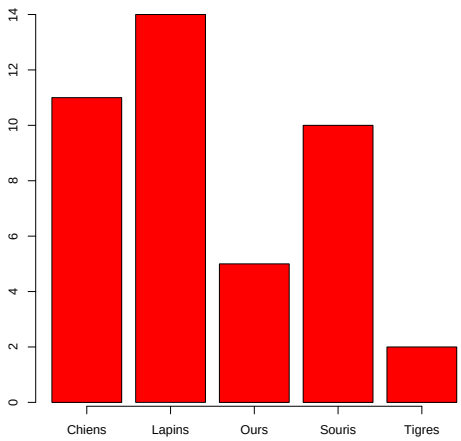
Variable qualitative

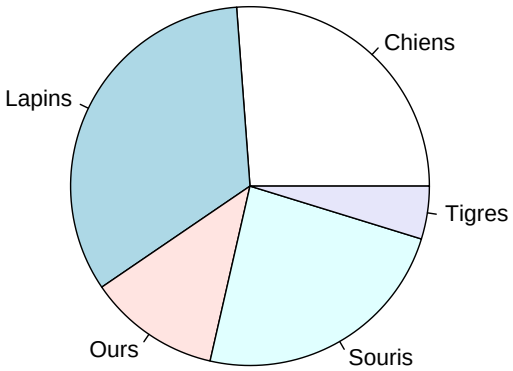
La fonction `summary()` donne l'effectif de chaque modalité de la variable qualitative.
Exemple :

```
> animaux_factor(c(rep("Chiens",11),rep("Lapins",14),rep("Ours",5),
                   rep("Souris",10),rep("Tigres",2)))
> summary(animaux)
Chiens Lapins   Ours Souris Tigres
    11     14      5     10      2
```

Présentation des données en diagrammes en colonnes et en secteurs

```
> plot(animaux)
> pie(summary(animaux))
```



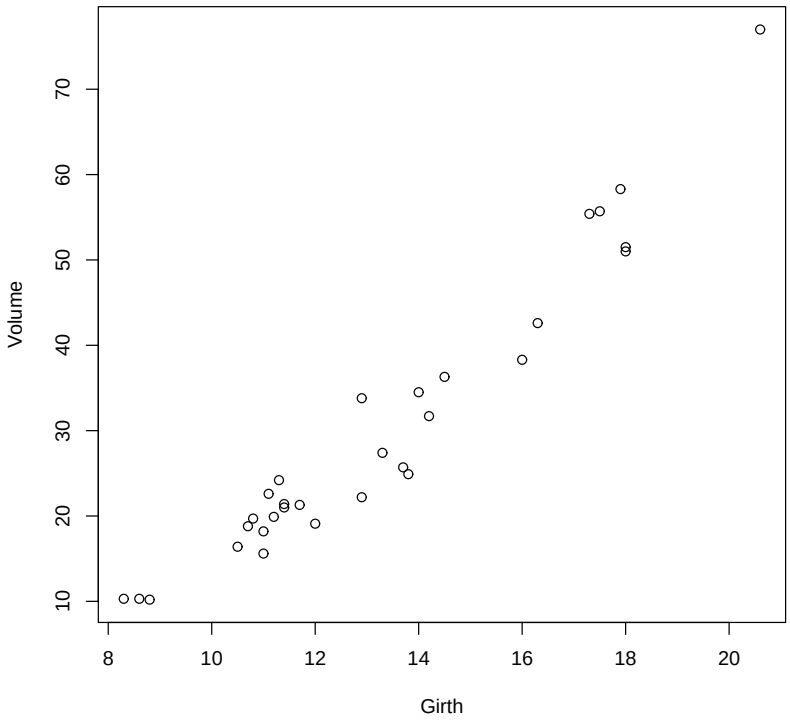


8.3.2 Statistique exploratoire bidimensionnelle

Variables quantitatives

Le nuage de points est un graphique très pratique afin de représenter les observations simultanées de deux variables quantitatives.
Exemple :

```
> data(trees)
> attach(trees)
> plot(Girth,Volume)
```



Ce graphique nous indique la façon dont les points se dispersent dans le plan. On l'appelle aussi diagramme de dispersion.(« scatter plot")
Il est possible d'obtenir la covariance et le coefficient de corrélation linéaire avec les fonctions `cov()` et `cor()`.
Exemples :

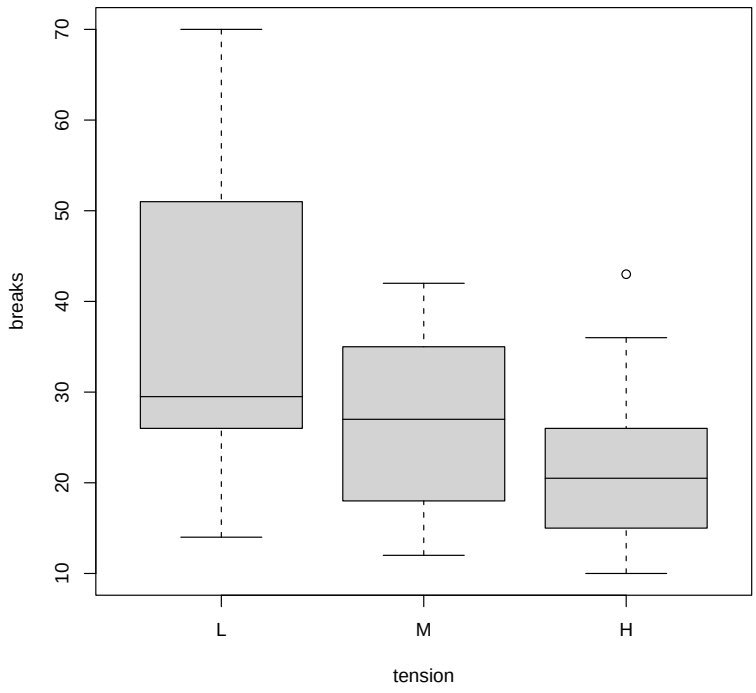
```
> cov(Girth,Volume)
[1] 49.88812
> cor(Girth,Volume)
[1] 0.9671194
```

Variables quantitative et qualitative

L'objectif est de voir l'évolution de caractéristiques statistiques selon le niveau de la variable qualitative.
On réalise des diagrammes en boites disposés en parallèle selon les modalités de la variable qualitative.

Exemple :

```
> plot(breaks~tension ,data = warpbreaks, col = "lightgray")
```



Pour la réalisation de ce graphique, les fonctions `plot()` et `boxplot()` donne le même résultat, avec en plus le nom des variables sur les axes pour la fonction `plot()`.

Variables qualitatives

On présente les données sous forme d'une table de contingence avec la fonction `table()`.
Exemple :

```
> animaux_factor(c(rep("Chiens",11),rep("Lapins",14),rep("Ours",5),
                   rep("Souris",10),rep("Tigres",2)))
> zoo <- factor(rep(c("A","B","C"),14))
> table(animaux,zoo)
        zoo
animaux  A B C
  Chiens 4 4 3
  Lapins 5 4 5
  Ours   1 2 2
  Souris 4 3 3
  Tigres 0 1 1
```

8.4 Régression linéaire simple

Un conducteur a relevé sa consommation d'essence pour divers parcours dont il connaît le kilométrage, il se propose de calculer la relation qui exprime la consommation en fonction du kilométrage en considérant soit que la consommation est proportionnelle au kilométrage (pas d'ordonnée à l'origine) soit qu'il y a une consommation du simple fait du démarrage.

8.4.1 Données brutes

Les données que l'on possède nous renseignent sur la consommation d'essence en litres (cs) de la voiture suivant la distance (km) parcourue par cette dernière. On a un échantillon de 10 observations.

Obs	km	cs
1	5	0.1
2	20	1.1
3	25	1.7
4	15	0.9
5	300	20.7
6	140	10.1
7	550	38.2
8	25	2.3
9	60	4.8
10	650	45.8

Commandes permettant de préparer les données :

```
> consom <- read.table("consom.txt")
# Création d'un jeu de données R (data.frame) à partir d'un fichier
  texte
> names(consom) <- c("km","cs")
# Nomme les variables V1 et V2 en km et cs
> attach(consom)
# Rend visible les vecteurs constituant consom
```

8.4.2 Méthodes

On réalise une régression simple avec comme variable expliquée la consommation (cs) et comme variable explicative (km) la distance.

Le modèle

$$y = X\beta + e$$
$$y_i = ax_i + b + e_i$$

Indicateurs de la qualité du modèle

Les indicateurs permettant de mesurer la qualité d'un modèle de régression linéaire simple sont :

- **Le coefficient de qualité** R^2 (carré du coefficient de corrélation)

$$R^2 = r^2(x,y) = \frac{cov(x,y)^2}{var(x)var(y)}$$

$$R^2 = r^2(x,y) = \frac{var(\hat{y})}{var(y)} \qquad 0 \leq R^2 \leq 1$$

  Plus ce coefficient est proche de 1, meilleur est le modèle.

- **Le** $\hat{\sigma^2}$

$$\hat{\sigma^2}(y) = \frac{\sum \hat{e}_i^2}{ddl}$$

  avec ddl = degré de liberté = nombre d'observations - nombre de paramètres inconnus
  Plus le $\hat{\sigma^2}$, mesurant l'importance des résidus, est faible, meilleur est le modèle.

- **Les erreurs standards**
  Elles doivent être faibles car elles mesurent l'incertitude des paramètres inconnus.

- **Les tests réalisés sur les paramètres**
  Les tests de Student réalisés sur chaque coefficient doivent être significatifs pour déduire que la variable explicative a significativement un effet.

Vérification de la qualité du modèle

Afin de vérifier si le modèle est bon, on doit visualiser :

- **Les résidus par rapport aux valeurs prédites**
  Il ne doit pas exister de liaisons entre les points.

- **Les valeurs prédites par rapport aux valeurs observées**
  Les points doivent être ajustés sur la première bissectrice.

8.4.3 Commandes R utilisées pour cette régression

Commandes permettant de mettre en œuvre la régression et ses résultats :

```
> regs <- lm(cs~km)
# Commande permettant de stocker la regression effectuée dans regs
> summary(regs)
# Donne les valeurs principales de la régression effectuée

Call:
lm(formula = cs ~ km)

Residuals:
    Min      1Q  Median      3Q     Max
-0.3555 -0.3062 -0.1555  0.2934  0.5484

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.0574285  0.1543543   0.372     0.72
km          0.0699026  0.0005325 131.262 1.27e-14 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3839 on 8 degrees of freedom
Multiple R-Squared: 0.9995,Adjusted R-squared: 0.9995
F-statistic: 1.723e+04 on 1 and 8 DF,  p-value: 1.269e-14
```

Pour afficher, les éléments contenant les résultats d'une analyse, on utilise la fonction `names()`.

```
> names(regs)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

> names(summary(regs))
 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "sigma"         "df"            "r.squared"     "adj.r.squared"
 [9] "fstatistic"    "cov.unscaled"
```

Il est aussi possible d'extraire les résultats fournis afin de pouvoir les réutiliser.

```
> summary(regs)["sigma"]
$sigma
[1] 0.3839064
```

Les fonctions `residuals()` et `predict()` permettent d'afficher les résidus calculés et les valeurs ajustées de la régression.

```
> round(residuals(regs),3) # arrondi à la troisième décimale
     1      2      3      4      5      6      7      8      9     10
-0.307 -0.355 -0.105 -0.206 -0.328  0.256 -0.304  0.495  0.548  0.306

> round(predict(regs),3) # arrondi à la troisième décimale
     1      2      3      4      5      6      7      8      9     10
 0.407  1.455  1.805  1.106 21.028  9.844 38.504  1.805  4.252 45.494
```

La commande `plot()` peut être utilisée directement sur l'objet de la régression, elle fournit quatres types de graphiques.

- Les résidus par rapport aux valeurs ajustées
- Les résidus standardisés par rapport aux quantiles théoriques
- Les résidus standardisés par rapport aux valeurs ajustées
- Les distances de Cook (Ceci est un indicateur qui permet de mesurer l'influence de chaque observation)

```
> plot(regs)
```

Il est aussi intéressant de réaliser ses propres graphiques d'analyse.

- Le nuage de points des observations et la droite de régression

```
> plot(consom, main="Nuage de points des observations et droite
      de régression")
> abline(regs)
```

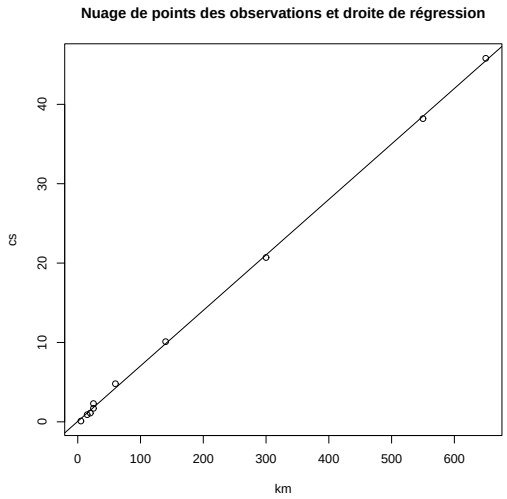
- Le nuage de points des valeurs prédites en fonction des valeurs observées

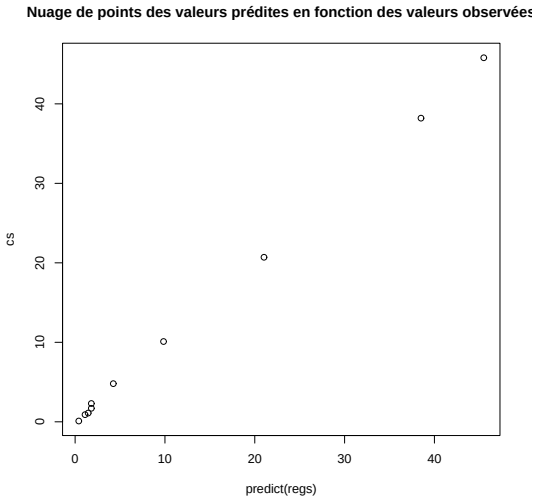
```
> plot(predict(regs),cs, main="Nuage de points des valeurs prédites
      en fonction des valeurs observées")
```

8.4.4 Résultats

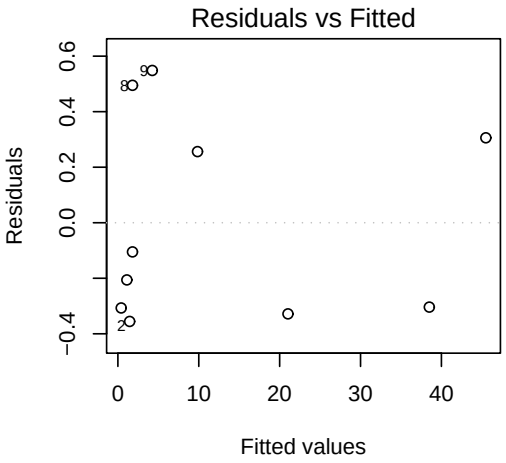
Modèle : $cs_i = akm_i + b + e_i$
Ce modèle prédit la consommation en fonction du nombre de kilomètres.

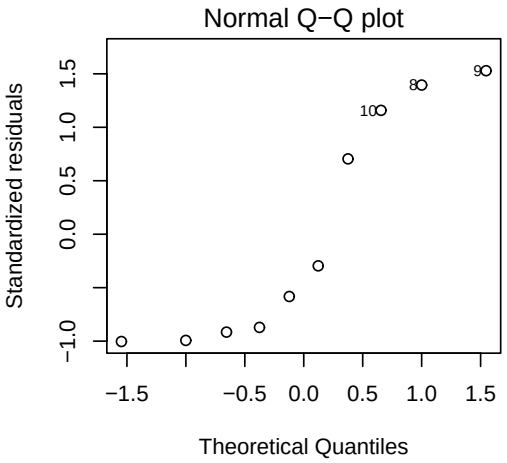
Résultats et graphiques
intercept correspond au coefficient b et km correspond au coefficient a du modèle.
$R^2 = 0.9995$
$\hat{\sigma^2} = 0.14738$
Standard Error :
- intercept = 0.15435
- km = 0.00053254
P-Value : `Pr(>|t|)`
- intercept = 0.72
- km = 1.27e-14 (<0.0001)

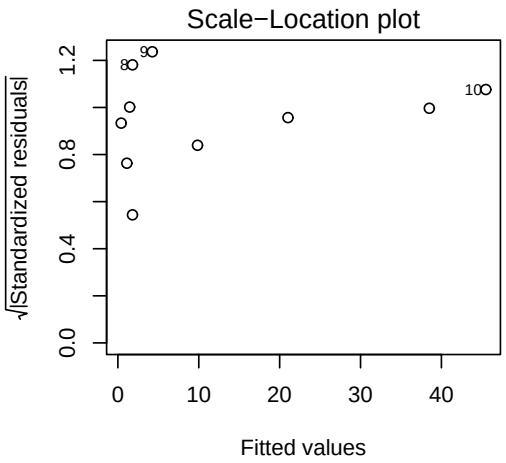


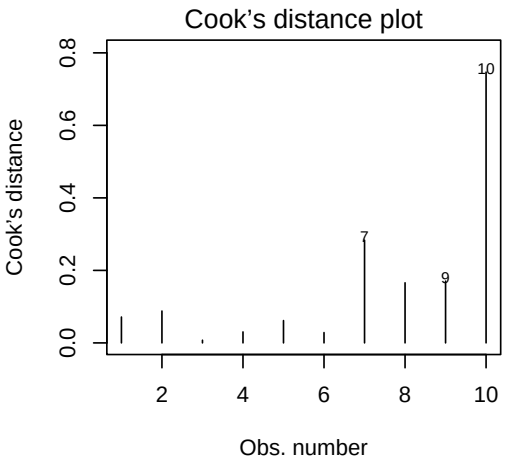


Sorties graphiques standards









Commentaires

- *Ce modèle a un R^2 très proche de 1* (0,9995) c'est à dire que la variation de consommation d'essence (cs) est expliquée à 99.95% par la distance parcourue (km), c'est donc un très bon ajustement des données.

- *L'estimation de la variance* ($\hat{\sigma^2} = 0.14738$) obtenue est faible donc tous les points sont très proches de la droite de régression.

- *Les erreurs standards* sont très faibles (proches de 0), montrant qu'il y a peu d'incertitude pour l'estimation des paramètres.

- *La droite de régression* obtenue pour ce modèle est : $cs = 0,0574 + 0,0699km$

- *La « P-Value »du test* sur le coefficient correspondant au km étant inférieure à 5%, il est possible de dire que les km ont significativement un effet sur la consommation. Il est donc possible de prédire la consommation de la voiture avec précision en fonction de la distance à parcourir. Cela veut dire que la voiture consomme sept litres aux cent kilomètres.

- *La représentation graphique des résidus* suivant les consommations prédites de ce modèle montrent que les points sont repartis de façon inégale et ne suivent pas une tendance particulière. On peut donc dire que la valeur des résidus est indépendante de la consommation, autrement dit que la prédiction est de bonne qualité. Tous les résidus sont compris entre -2 et 2, il n'y a pas d'observations aberrantes.

- De plus, le graphique représentant les consommations prédites en fonction des consommations observées montrent que les points sont alignés avec la première bissectrice ce qui permet de valider ce modèle. En effet, les prévisions effectuées sont bonnes, elles doivent normalement être égales aux valeurs observées c'est à dire que les points du graphique doivent être alignés sur la droite d'équation "cs = predict(regs)". Dans ce cas, il semble que tous les points soient très proches de cette droite.

- Pour cet ensemble de raisons, il est possible d'affirmer que *ce modèle est un très bon modèle.*

8.5 Regression linéaire multiple

Les données du fichier `'proces'` ont été recueillies dans le but d'etudier les variables qui favorisent le developpement des chenilles processionaires.
On a donc observé le nombre moyen de nids de chenilles par arbre et relevé d'autres variables telles que altitude, pente, ... , dont on pense qu'elles peuvent influencer le developpement de ces chenilles.
Les variables sont donc les suivantes :

- V1 = 'altitude de la placette'
- V2 = 'pente de la placette'
- V3 = 'nombre de pins par placette'
- V4 = 'hauteur moyenne de la placette'
- V5 = 'diamètre moyen de la placette'
- V6 = 'densité de peuplement de la placette'
- V7 = 'orientation de la placette'
- V8 = 'hauteur des arbres dominants de la placette'
- V9 = 'nombre de strates de végétation de la placette'
- V10 = 'mélange de peuplement
  1 : pas de mélange et 2 : mélange maximum'
- V11 = 'nb nids par arbre'

8.5.1 Données brutes

On pourra s'intéresser à l'explication de la variable '`V11`' par les autres variables.

Commandes permettant de préparer les données :

```
> proces <- read.table("proces")
> proces
     V1 V2 V3  V4   V5  V6  V7   V8  V9 V10  V11
1  1200 22  1 4.0 14.8 1.0 1.1  5.9 1.4 1.4 2.37
2  1342 28  8 4.4 18.0 1.5 1.5  6.4 1.7 1.7 1.47
3  1231 28  5 2.4  7.8 1.3 1.6  4.3 1.5 1.4 1.13
4  1254 28 18 3.0  9.2 2.3 1.7  6.9 2.3 1.6 0.85
5  1357 32  7 3.7 10.7 1.4 1.7  6.6 1.8 1.3 0.24
6  1250 27  1 4.4 14.8 1.0 1.7  5.8 1.3 1.4 1.49
...
```

8.5.2 Méthodes

On met en œuvre la méthode de régression descendante de sélection de variables explicatives. Cette méthode consiste à retirer du modèle complet les variables « inutiles »une par une avec une règle d'arrêt.
On réalise donc une régression multiple avec comme variable à expliquer '`V11`' et comme variables explicatives l'ensemble des autres variables ; mais seulement certaines seront retenues.

Le modèle

$$y = X\beta + e$$
$$y_i = a_0 + a_1 x_i^1 + ... + a_j x_i^j + ... + a_p x_i^p + e_i$$

Indicateurs et vérification de la qualité du modèle

Les indicateurs permettant de mesurer la qualité d'un modèle de régression linéaire multiple sont les mêmes que pour la régression linéaire simple.
Par contre, afin de comparer plusieurs modèles dont le nombre de variables explicatives est différent, on préfèrera comparer les $\hat{\sigma^2}(y)$ aux R^2 car ils prennent en compte le nombre de variables explicatives.
Pour ces comparaisons, le R^2, qui augmente avec le nombre de variables est remplacé par **le R^2 ajusté** qui pénalise par une fonction croissante le nombre de variables explicatives.

$$R^2_{aj} = \frac{(n-1)R^2-p}{n-p-1} \text{ avec p = nombre de variables du modèle}$$

En ce qui concerne la vérification du modèle, les méthodes seront les mêmes que pour la régression linéaire simple c'est-à-dire la visualisation des différents nuages de points afin de déceler les eventuelles anomalies. (par exemple : une tendance pour les résidus)

8.5.3 Commandes R utilisées pour cette régression

Commandes permettant de mettre en œuvre la régression et ses résultats :

```
> regm <- step(lm(V11~.,data=proces))
> regm

Call:
lm(formula = V11 ~ V1 + V2 + V3 + V4 + V5 + V9 + V10, data = proces)

Coefficients:
(Intercept)          V1          V2          V3          V4          V5
  8.574664   -0.003246   -0.039008    0.033546   -0.495130    0.119277
         V9         V10
 -0.791964   -0.467649

> summary(regm)

Call:
lm(formula = V11 ~ V1 + V2 + V3 + V4 + V5 + V9 + V10, data = proces)

Residuals:
     Min       1Q   Median       3Q      Max
-1.08075 -0.26865 -0.02528  0.21793  1.30590
```

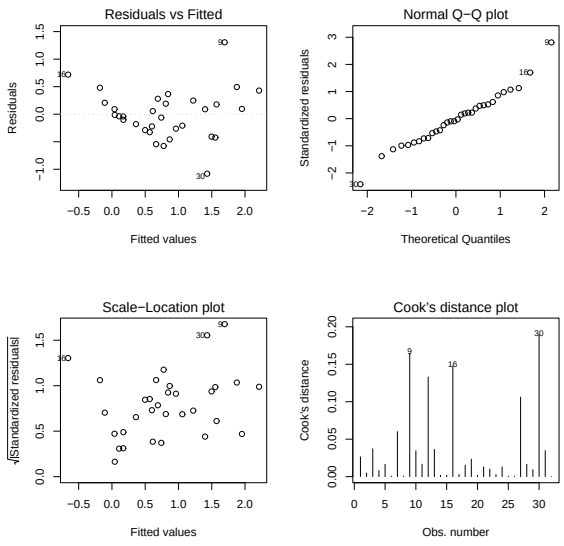
```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  8.5746635  1.6365143   5.240 2.27e-05 ***
V1          -0.0032459  0.0009002  -3.606  0.00142 **
V2          -0.0390085  0.0132840  -2.937  0.00721 **
V3           0.0335458  0.0234434   1.431  0.16534
V4          -0.4951302  0.2471322  -2.004  0.05654 .
V5           0.1192773  0.0550845   2.165  0.04052 *
V9          -0.7919635  0.3871204  -2.046  0.05189 .
V10         -0.4676485  0.3736079  -1.252  0.22274
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.502 on 24 degrees of freedom
Multiple R-Squared: 0.7091,Adjusted R-squared: 0.6243
F-statistic: 8.358 on 7 and 24 DF,  p-value: 3.539e-05

> summary(regm)["sigma"]
$sigma
[1] 0.5020037
> par(mfrow=c(2,2)); plot(regm)
```

8.5.4 Résultats

Modèle : $V11_i = a_0 + a_1 V1_i + a_2 V2_i + a_3 V3_i + a_4 V4_i + a_5 V5_i + a_6 V9_i + a_7 V10_i + e_i$
Ce modèle prédit le nombre de nids par arbre en fonction de l'altitude, de la pente, du nombre de pins, de la hauteur moyenne, du diamètre moyen, du nombre de strates de la placette et du mélange de peuplement.

Résultats et graphiques

$R^2 = 0.7091$ $R^2_{aj} = 0.6243$ $\hat{\sigma^2} = 0.252$



8.6 L'analyse en composantes principales

R propose plusieurs plusieurs modules pour réaliser l'Analyse en Composantes Principales (**mva, multiv**, ...) et le module **multidim** a été choisi.
Ce module a été développé par des français André Carlier et Alain Croquette du laboratoire de Statistique et Probabilité de l'Université Paul Sabatier de Toulouse. A la base, il a été développé pour **S-Plus** et il est encore développé à l'heure actuelle. Un portage sur **R** a été effectué par Mathieu Ros, ancien étudiant de l'Université Paul Sabatier. Ce module est actuellement disponible dans le répertoire dédié aux modules en développement du **CRAN**.
Les fonctions présentées dans cette partie dédié à l'ACP sont toutes issues de ce module.

8.6.1 présentation

On dispose d'un tableau '`x`' formé de 'p' variables quantitatives rangées en colonne. L'Analyse en Composantes Principales (ACP) permet d'analyser les liaisons linéaires existant entre ces variables, d'effectuer une analyse des *dissemblances / ressemblances* entre les lignes de '`x`', et de réaliser une interprétation simultanée qui met en relation les deux interprétations précédentes.

8.6.2 Les données

Lors d'une crue de la rivière « Baget », située dans un massif karstique des Pyrénées, près de Moulis en Ariège, on a effectué des prélèvements à 34 instants successifs (répartis sur environ 3 jours, la durée de la crue), et on a mesuré la température de l'eau et le débit de la rivière. Les prélèvements ont été ensuite analysés et ont donné lieu à 7 analyses chimiques ou électriques. L'étude des caractéristiques de l'eau pendant la crue permet de comprendre quel a pu être son trajet souterrain. Les 9 mesures effectuées pour les 34 prélèvements sont les suivants :

- ca concentration en ion calcium
- mg concentration en ion magnésium
- cl- concentration en ion chlorure
- so4- concentration en acide sulfurique
- hco3- concentration en carbonate
- co2 concentration en gaz carbonique
- c conductivité
- te température
- deb débit de la rivière.

Le tableau des crues du « Baget »(`crbg`) :

```
> crbg
     ca  mg   cl  so4 hco3  co2   c    te   deb
1  62.8 3.5 1.75 27.5  173 2.93 300 10.40 0.092
2  62.8 4.0 1.65 27.0  177 2.92 300 10.40 0.490
3  64.5 3.6 1.65 28.5  179 3.10 303 10.40 0.560
4  64.0 5.0 1.70 26.0  184 3.33 308 10.40 0.605
5  61.6 5.0 1.90 28.0  178 3.08 303 10.40 0.690
6  63.5 4.4 1.75 27.5  179 2.76 302 10.50 0.740
...
```

8.6.3 La fonction acp()

Généralités

La fonction `prcomp()` de **S-Plus**, qui calcule les composantes principales d'un tableau de données a été réécrite, pour accepter des options nouvelles, sous le nom d'`acp()`.
Après l'exécution de cette fonction, avec comme unique argument le nom du tableau, un premier ensemble de résultats est affiché.
Effectuons l'ACP sur le tableau des crues du « Baget ».

```
> crbg.acp <- acp(crbg)
ACP du tableau "crbg" sur variables réduites

Inertie totale : 9

Pourcentage d' inertie expliquée :

            f1 f2 f3 f4 f5 tot
inertie exp 44 38 11  4  2  98
inertie cum 44 81 93 97 98  98
```

Arguments de la fonction acp()

Le seul argument obligatoire de la fonction `acp()` est le tableau de données, avec les individus en lignes et les variables en colonnes.
Les arguments optionnels sont :

- Le vecteur des poids des individus '`wt`' (par défaut équipondération des individus)
- Un vecteur définissant la diagonale de la *matrice diagonale des poids* dans l'espace des variables, '`d`' (par défaut un vecteur de « `1` »)

La fonction `acp()` admet trois autres options, deux permettant de contrôler le centrage et la réduction des variables et une permettant un contrôle préalable des contributions des lignes et des colonnes à l'inertie totale :

- `reduc=F` : supprime la réduction des variables (par défaut `reduc=T`)
- `ctr=F` : supprime le centrage des variables (par défaut `ctr=T`)

- `contav=T` : permet un contrôle préalable : un affichage des contributions des individus et des variables permet de vérifier la pertinence des options choisies (par défaut `contav=F`).
- `method="acp"` : Permet de savoir quelle est la fonction qui appelle l'ACP.
- `tol=10^(-8)` : Valeur en dessous desquelles les valeurs propres sont ramenées à 0 (pour éviter des valeurs propres négatives, dues aux erreurs d'arrondis).

Dans le cas où les noms de variables ne figurent pas dans le tableau de données, la fonction `acp()` leur attribue les noms v1,...,vp.

Les résultats de l'acp

Les résultats (ou les valeurs) de la fonction sont regroupés dans un objet de classe `acp` ayant une structure de liste (il faut faire `names(crbg.acp)` pour avoir les noms des composants de la liste). Cette liste contient des résultats calculés, des indicateurs logiques des options choisies, et les noms des objets ayant servi à l'appel.
Dans la pratique et sauf besoin spécifique, il n'est pas nécessaire de connaître la liste des composants, qui est donnée ci-dessous :

- `$values` : valeurs propres (on note $(\lambda_s)^2$ la s-ième valeur propre)
- `$vectors` : vecteurs propres (on note v_{js} la j-ième composante du s-ième vecteur propre)
- `$cmpr` : composantes principales (on note c_{is} la s-ième composante principale du i-ème vecteur individu)
- `$d` : vecteur définissant les poids des variables dans la distance entre individus
- `$pi` : vecteur des poids des individus
- `$reduc` : indicateur logique (analyse sur variables réduites ou non)
- `$ctr` : indicateur logique (analyse sur variables centrées ou non),
- `$moy` : moyennes
- `$sigma` : écart-types
- `$xnom` : nom du tableau de données
- `$wtequal` : indicateur logique (individus équipondérés ou non)
- `$dusual` : indicateur logique (distance usuelle ou non)
- `$dnom` : nom du vecteur définissant la distance
- `$isup` : (optionnel) composantes principales des individus et barycentres supplémentaires
- `$vsup` : (optionnel) coordonnées des variables supplémentaires

Dans notre exemple, cette liste résultat a pour nom `crbg.acp`.
Si on veut accéder aux composantes principales, arrondies à 3 décimales, on peut taper :

```
> round(crbg.acp$cmpr, 3)
      f1     f2     f3     f4     f5     f6     f7     f8     f9
1  1.624 -2.609  0.961  0.480 -1.000 -0.087 -0.163  0.032 -0.109
2  1.700 -2.198  0.201  0.379 -0.271  0.190  0.142 -0.053 -0.087
3  1.576 -1.735  0.314 -0.175 -0.592  0.262  0.246 -0.131 -0.037
4  1.780 -0.857 -0.033  1.021  0.245  0.278  0.540  0.061  0.060
5  1.604 -1.490 -0.422  1.353  0.247 -0.281 -0.361 -0.065 -0.094
6  1.996 -1.698 -0.224  0.266  0.202 -0.413  0.095 -0.045  0.115
7  2.482 -0.764  0.105 -0.325  0.140  0.009  0.304 -0.069  0.280
8  2.462  0.812  0.523 -0.587 -0.004 -0.044 -0.202 -0.004 -0.044
9  2.821  0.394 -0.090  0.596  0.696  0.048 -0.210  0.095 -0.160
10 2.406 -0.412 -0.094 -0.047  0.261  0.050 -0.170  0.102  0.126
.....................................................................
```

8.6.4 Les fonctions numériques associés à l'ACP

La fonction print.acp()

Soit l'objet `crbg.acp`, résultat de l'ACP précédente, en tapant le nom de cet objet, on obtient :

```
> crbg.acp
ACP du tableau "crbg" sur variables réduites

Inertie totale : 9

Pourcentage d'inertie expliquée :

            f1 f2 f3 f4 f5 tot
inertie exp 44 38 11  4  2  98
inertie cum 44 81 93 97 98  98
```

Sous **R**, taper le nom d'un objet équivaut à taper : `print(nom-de-l'objet)`. L'objet `crbg.acp` étant de classe `acp`, taper le nom de cet objet est encore équivalent à taper `print.acp(crbg.acp)`.
Cette fonction redonne les arguments ayant servi dans l'appel de l'ACP, et fournit l'inertie totale et les pourcentages d'inertie expliquée par les 5 premiers facteurs.
Cette fonction admet les arguments optionnels :

- `digits` : Nombre de décimales dans les pourcentages expliqués (la valeur par défaut est 0).
- `nmax` : Nombre maximum de facteurs pris en compte dans les tableaux (la valeur par défaut est 5).

La fonction summary.acp()

Elle est appelée par la fonction générique `summary()`. C'est une fonction analogue à la précédente, mais qui fournit aussi les contributions des variables aux facteurs, à l'inertie totale, et qui donne les valeurs des variances initiales des variables (ramenées à une somme égale à 1000).
Elle s'appelle simplement par :

```
> summary(crbg.acp)
 Contribution des variables aux facteurs, a l'inertie totale
 et variances initiales (relatives) des variables

        f1   f2   f3   f4   f5 Inertie Variances
ca      24  224   77  174    8     111        21
mg     130   70   58  483  165     111         2
cl     165   83   11   36    0     111         0
so4    212   40    4    4   54     111       213
hco3    50  200   73    3  197     111       148
co2     87  135   90   89  474     111         1
c       66  212   10    7    4     111       615
te     229    1   38   19   13     111         0
deb     38   35  639  186   87     111         0
Total 1000 1000 1000 1000 1000    1000      1000
```

Cette fonction admet les mêmes arguments optionnels que la fonction `print.acp()`.

La fonction contri()

La fonction `contri()`, dont un extrait du résultat est donné ici, donne les contributions de chaque individu et de chaque variable à la variance de chaque facteur et à l'inertie totale. De plus elle calcule, pour chaque individu, l'indice d'originalité (proportionnel au carré de la distance de l'individu au point moyen) dans la colonne `ORI` et donne des indices mesurant la qualité des approximations de chaque individu et de chaque variable dans les sous-espaces principaux : ces indices sont des carrés de cosinus, ou, pour les variables, des carrés de coefficients de corrélation. On peut sélectionner les indices associés aux variables seules en ajoutant l'argument `ind=F`, ou aux individus seuls en tapant `var=F`. Enfin, on peut sélectionner les tableaux un par un dans un menu, avec l'option `ask=T` (pour sortir du menu, on donne la réponse 0).
Remarque : si les individus sont équipondérés, l'indice d'originalité et la contribution à l'inertie totale sont égaux.

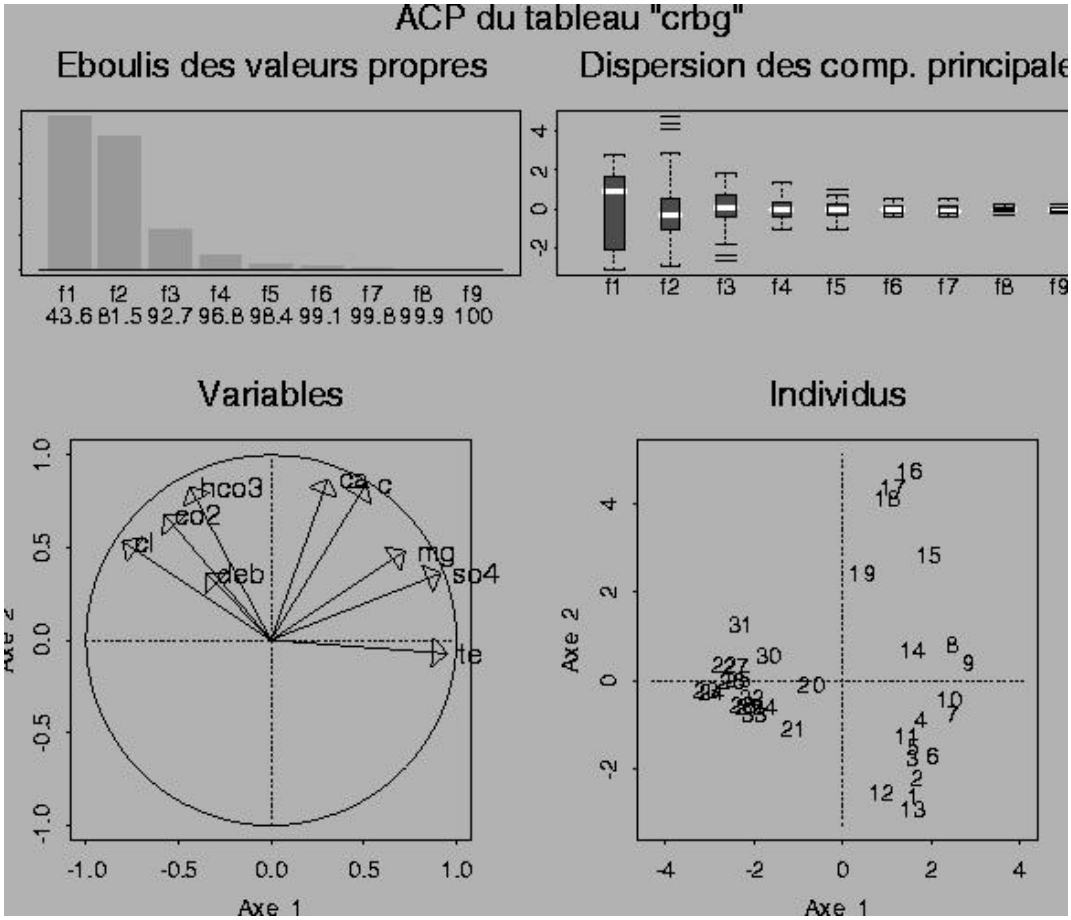
```
> contri(crbg.acp,ask=T)
```

```
Choix des contributions et indices de qualité

1: Contribution des facteurs aux individus
2: Qualité de la représentation des individus sur les ss-esp. principaux
3: Contribution des individus aux facteurs
4: Contribution des facteurs aux variables
5: Qualité de la représentation des variables sur les ss-esp. principaux
6: Contribution des variables aux facteurs
Sélection: 5

Qualité de la représentation des variables sur les ss-esp. principaux
      f1 f1a2 f1a3 f1a4 f1a5 tot
  ca  95  854  929  993  995 1000
  mg 520  753  800  970  998 1000
  cl 642  927  941  956  956 1000
 so4 843  973  976  977  985 1000
hco3 194  881  951  953  983 1000
 co2 334  797  886  927  992 1000
   c 264  979  990  992  993 1000
  te 905  911  947  952  955 1000
 deb 125  256  923  988  999 1000
 Tot 436  815  927  968  984 1000

Choix des contributions et indices de qualité

1: Contribution des facteurs aux individus
2: Qualité de la représentation des individus sur les ss-esp. principaux
3: Contribution des individus aux facteurs
4: Contribution des facteurs aux variables
5: Qualité de la représentation des variables sur les ss-esp. principaux
6: Contribution des variables aux facteurs
Sélection:  6

 Contribution des variables aux facteurs
      f1   f2   f3   f4   f5  f>5  Tot VARI
  ca  24  223   73  176   11   37  111   21
  mg 133   68   47  464  194   10  111    2
  cl 164   84   14   39    0  305  111    0
 so4 215   38    3    2   60  101  111  213
hco3  49  201   70    7  202  116  111  148
 co2  85  136   88  114  437   57  111    1
   c  67  210   11    7    4   51  111  615
  te 231    1   36   12   19  314  111    0
 deb  32   39  659  179   73    7  111    0
     1000 1000 1000 1000 1000 1000 1000 1000
```

```
Choix des contributions et indices de qualité

1: Contribution des facteurs aux individus
2: Qualité de la représentation des individus sur les ss-esp. principaux
3: Contribution des individus aux facteurs
4: Contribution des facteurs aux variables
5: Qualité de la représentation des variables sur les ss-esp. principaux
6: Contribution des variables aux facteurs
Sélection: 0
```

8.6.5 La fonction plot.acp()

Cette fonction est appelée par la fonction `plot()` quand elle agit sur un objet de classe `acp`. Elle représente sur un même écran quatre graphiques :



Graphique résultat de la fonction `plot.acp()` sur les données du Baget

1) **L'éboulis des valeurs propres**
   Il visualise la contribution des composantes principales à l'inertie totale. Ces contributions sont égales aux variances des composantes principales.

2) **Les diagrammes en boite des composantes principales**
Complémentaire du précédent, il représente la dispersion de ces mêmes composantes par des diagrammes en boite ou boxplots. Ces boxplots permettent de situer les médianes des composantes principales, les quartiles, mais aussi, et ce qui est important dans ce contexte, les points extrêmes (éventuellement aberrants ou « outliers »).

3) **Le cercle des corrélations**
Il est défini par défaut par les deux premiers facteurs. Le cercle de corrélation sur le plan 1-2 permet de visualiser :

- les coefficients de corrélations entre les variables et les deux premiers facteurs : la corrélation entre un facteur et une variable est la coordonnée de cette variable sur l'axe associé.
- la qualité de la représentation des variables sur le plan principal. Le coefficient de corrélation multiple entre une variable et les deux premiers facteurs est égale à la longueur de la projection du vecteur variable sur le plan.
- Les corrélations entre les variables deux à deux : si au moins un vecteur variable 'j_0' est sur le cercle unité, cette variable est dans le plan déterminé par les deux premiers facteurs. Dans ce cas, on peut lire graphiquement les corrélations de n'importe quelle variable 'j' avec 'j_0' : cette corrélation est égale à la longueur de la projection du vecteur 'j' sur le vecteur 'j_0', comptée positivement si les variables forment un angle aigu et négativement sinon.

4) **La représentation isométrique des individus**
On obtient par défaut leur représentation sur le plan principal. Cette représentation isométrique permet de faire une analyse des proximités des individus, d'y représenter d'éventuels barycentres illustratifs, ou encore des partitions.

Deuxième partie

Annexes

Annexe A

Compléments d'information

A.1 Le site du CICT : site miroir du CRAN

Philippe Baqué et Joseph Saint Pierre, ingénieurs au **CICT**, Centre Interuniversitaire de Toulouse, ont mis en place sur le site internet du **CICT** un site miroir du **CRAN**.
Ceci grâce à *rsync* qui met à jour ce réseau de distribution de **R** d'heure en heure sur le site miroir. Il est donc actuellement possible de récupérer les fichiers sources, les modules et la documentation de la même manière que sur le site officiel du **CRAN**.
Il est conseillé aux utilisateurs français de passer par ce site miroir afin d'éviter de surcharger le réseau. De plus, le téléchargement des différents fichiers est plus rapide du fait qu'il est moins distant.
L'adresse à laquelle il est possible de se connecter à ce site miroir du **CRAN** est : http ://diane.cict.fr/CRAN/.

A.2 Modules supplémentaires extérieurs au CRAN

Certains modules supplémentaires compatibles avec **R** ne sont pas accessibles par le **CRAN**.
En effet, les développeurs de ces modules n'ont pas trouvé nécessaire de les mettre à disposition par l'intermédiaire du **CRAN**.
Néanmoins, il est possible de les récupérer en connaissant les adresses des sites internet où ils sont disponibles.
Par exemple, Jim Lindsey, professeur à l'Université de Liège en Belgique met à disposition sur sa page internet (http ://alpha.luc.ac.be/ lucp0753/rcode.html) différents modules portant sur la manipulation des séquences moléculaires, les modèles non-linéaires généralisés, les fonctions de probabilité, les modèles pour mesures répétées normales et non normales, etc.
Il est possible de trouver sur internet un grand nombre de modules de **R** non répertoriés dans le réseau **CRAN**. La plupart du temps ces modules sont liés à des activités spécifiques. La génétique par exemple :
http ://www.stat.uni-muenchen.de/ strimmer/rexpress.html

A.3 Recherche, aide et communication

Le site officiel de **R** comporte plusieurs pages permettant de faire connaître, de manipuler et d'aider les utilisateurs à maîtriser l'environnement **R**.
Ces pages comportent plusieurs moyens d'informations et de recherche :

- Il est possible de rechercher certains mots dans la documentation, dans les courriers électroniques archivés sur le site de **R** par l'intermédiaire d'un site de recherche.
  (« R site search »-> http ://finzi.psych.upenn.edu/search.html)
  Ce site est efficace et classe l'ensemble des éléments trouvés par ordre d'importance (de fréquentation du document).
  Ce site est développé par Jonathan Baron du département de psychologie de l'Université de Pennsylvanie aux Etats Unis.
- Afin de pouvoir communiquer avec les membres du projet **R** et sur l'environnement **R** , il existe trois listes de discussion relatives à différents sujets.
  - **r-announce** : Cette liste permet de communiquer le développement de **R**, du logiciel, de la documentation et des modules. De plus, cette liste permet d'annoncer les nouveaux produits disponibles.
  - **r-help** : C'est la liste principale de discussion de **R**, elle permet aux utilisateurs de poser des questions et d'apporter des réponses au sujet des problèmes et des solutions par rapport à l'utilisation de **R**, à sa comparaison et à sa compatibilité avec **S** et **S-Plus**. Des exemples d'utilisation peuvent aussi être envoyés.
  - **r-devel** : Cette liste de discussion est dédiée aux versions futures de **R** et à leurs tests.
- Il existe une importante documentation liée au projet **R**. Cette documentation est réalisée par l'équipe de développement de **R** (« **R** Developpement Core Team ») et par certains utilisateurs. L'équipe de **R** a produit et met à jour en permanence six manuels écrit en anglais.
  - *« An Introduction to R »* qui donne une introduction au système **R** et montre comment utiliser **R** au niveau des analyses statistiques et des graphiques.
  - *« The R language definition »* détaille et définit le langage **R**.
  - *« Writing R Extensions »* qui explique comment créer ses propres modules, écrire une aide **R**...
  - *« R Data Import/Export »* décrit les moyens d'importer et d'exporter des données à l'aide du logiciel **R** ou de modules spécifiques.
  - *« R Installation and Administration »* décrit les méthodes d'installation et d'administration de **R**
  - *« The R Reference Index »* contient tout les fichiers d'aide de **R** (logiciel et modules standards)

En plus de ces manuels, il est proposé aux utilisateurs le « **R-FAQ** » qui est un document correspondant aux questions les plus fréquemment posées. Ce document est au format *html* sur le site de **R** et est aussi disponible au format *pdf*. En fait, il existe réellement trois « **R-FAQ** » dédiés aux différents systèmes d'exploitations.(*Unix, Windows et MacOS*)
D'autres documentations ont été créées par les utilisateurs de **R**. Ces documentations ne sont pas automatiquement rédiger en anglais.

Annexe B

Glossaire

A

- **ACP** : Analyse en Composantes Principales

C

- **CICT** : Centre Inter Universitaire de Calcul de Toulouse
- **CRAN** : The Comprehensive R Archive Network, Réseau de distribution de **R**
- **copyleft** : Règle de protection des libertés fondamentales pour les logiciels

E

- **ESS** : Emacs Speacks Statistics, module de Emacs

F

- **FSF** : Free Software Foundation

G

- **GNU** : Système d'exploitation (GNU/Linux -> Linux)
- **GLP** : General Public License

H

- **HTML** : HyperText Markup Language

L

- **LaTeX** : Ensemble de programmes (logiciel) écrits en TeX pour réaliser des documents scientifiques et techniques

M

- **modules** : Ils sont les « packages »(du **CRAN** ou extérieurs au **CRAN**) compatibles avec **R** , ils peuvent aussi être appelés logiciels supplémentaires.
- **Multidim** : Module fonctionnant sous **S-Plus** et **R** pour l'analyse des données multidimensionnelles.

P

- **PDF** : Portable Document Format

R

- **R** : Environnement, système -> Langage et logiciel

S

- **S** : Langage de programmation
- **SAS** : Logiciel de statistique
- **Scheme** : Langage de programmation
- **SGBD** : Système de gestion de bases de données
- **SID** : Statistique et Informatique Décisionnelle
- **S-Plus** : Logiciel de statistique utilisant le langage **S**
- **SPSS** : Logiciel de statistique

T

- **TD** : Travaux dirigés
- **TEX** : Langage de programmation
- **TP** : Travaux pratiques

U

- **UNIX** : Système d'exploitation
- **UPS** : Université Paul Sabatier

W

- **Windows** : Système d'exploitation

X

- **XML** : eXtensible Markup Language

Bibliographie

[1] Dominique Bouillet, *UNIX Guide de l'utilisateur*, Ellipses, 1990.

[2] R Developpement Core Team, *An Introduction to R*, http ://www.R-project.org/, 2002

[3] R Developpement Core Team, *The R language definition*, http ://www.R-project.org/, 2002

[4] R Developpement Core Team, *Writing R extensions*, http ://www.R-project.org/, 2002

[5] R Developpement Core Team, *R Data Import/Export*, http ://www.R-project.org/, 2002

[6] R Developpement Core Team, *R Installation and Administration*, http ://www.R-project.org/, 2002

[7] R Developpement Core Team, *The R Reference Index*, http ://www.R-project.org/, 2002

[8] R Developpement Core Team, *R FAQ*, http ://www.R-project.org/, 2002

[9] Free Software Foundation, *GNU's Not Unix !*, http ://www.gnu.org/, 2002.

[10] Emmanuel Paradis, *R pour les débutants*, http ://cran.r-project.org/, 2000.

[11] Yves Brostaux, *Introduction à l'environnement de programmation statistique R*, http ://cran.r-project.org/, 2002.

[12] André Carlier et Alain Croquette, *Initiation au langage S-Plus*, http ://www.lsp.ups-tlse.fr/, 1999.

[13] André Carlier et Jean-René Mathieu, *Modelisation statistique*, Cours Trauvaux Dirigés et Travaux Pratiques IUP SID, 2001-2002.

[14] André Carlier et Alain Croquette, *La bibliothèque MULTIDIM*, http ://www.lsp.ups-tlse.fr/, 1999.

Index