

# INTRODUCTION à R

Max Bruciamacchie

28 janvier 2013



# Table des matières

<b>1</b>	<b>Premiers pas</b>	<b>9</b>
1.1	Premier script . . . . .	9
1.2	Fonctionnement de R . . . . .	10
1.2.1	Les extensions . . . . .	10
1.2.2	Les objets R . . . . .	10
1.2.3	Vectorisation . . . . .	10
1.2.4	Aide . . . . .	11
1.2.5	Organisation des répertoires . . . . .	11
<b>2</b>	<b>Manipulation des données</b>	<b>13</b>
2.1	Import/export . . . . .	13
2.1.1	La fonction read . . . . .	13
2.1.2	Exportation des données . . . . .	13
2.1.3	Connexion à une base de données externe . . . . .	14
2.1.4	Choix du fichier à l'aide d'une boîte de dialogue . . . . .	14
2.2	Stockage dans des objets . . . . .	15
2.2.1	Les 3 principaux objets . . . . .	15
2.2.2	Inspection . . . . .	17
2.2.3	Conversion . . . . .	18
2.2.4	Accès . . . . .	18
2.3	Création . . . . .	19
2.3.1	Fonctions de base . . . . .	19
2.3.2	Nouvelle colonne dans un data.frame . . . . .	22
2.3.3	Transformation d'une variable continue en classes . . . . .	22
2.3.4	Selon une loi de probabilité . . . . .	23
2.3.5	Création de data.frame . . . . .	24
2.3.6	Création de matrices . . . . .	24
2.3.7	Création de listes . . . . .	25
2.3.8	Création de fonctions . . . . .	26
2.4	Extraction/Suppression . . . . .	27
2.4.1	Objets booléens et instructions logiques . . . . .	27
2.4.2	Indexation directe . . . . .	28
2.4.3	Indexation par nom . . . . .	29
2.4.4	Indexation par condition . . . . .	29
2.4.5	Fonction which . . . . .	30
2.4.6	Suppression d'une colonne . . . . .	30
2.5	Modifications . . . . .	30
2.5.1	Présentation paysage ou portrait . . . . .	30
2.5.2	Fusion . . . . .	31
2.5.3	Aggrégation . . . . .	32
2.5.4	Concaténer des vecteurs/matrices . . . . .	34
2.6	Calculs . . . . .	35
2.6.1	Fonctions disponibles . . . . .	35
2.6.2	Tri . . . . .	36
2.6.3	Gestion des données manquantes . . . . .	36

2.6.4	Opérations sur les Matrices/Vecteurs . . . . .	38
2.7	Récurrence . . . . .	39
2.7.1	Boucles et tests . . . . .	39
2.7.2	Les fonctions apply, tapply, sapply, lapply, mapply, eapply . . . . .	40
2.7.3	La fonction summary . . . . .	43
2.7.4	Boucles ou Fonctions récurrentes ? . . . . .	44
2.8	Simulation . . . . .	44
2.9	Gestion des chaînes de caractères . . . . .	45
2.9.1	Les chaînes de caractères existantes . . . . .	45
2.9.2	Les fonctions de base . . . . .	45
2.9.3	Librairie stringr . . . . .	46
<b>3</b>	<b>Graphiques</b>	<b>47</b>
3.1	Les bases . . . . .	48
3.1.1	Un exemple simple . . . . .	48
3.1.2	Paramètres des graphes . . . . .	48
3.1.3	Mise en forme . . . . .	50
3.1.4	Graphes multiples . . . . .	50
3.2	Fonction plot . . . . .	54
3.2.1	Quelques exemples . . . . .	54
3.2.2	Les options . . . . .	57
3.3	Fonctions graphiques secondaires . . . . .	59
3.3.1	Liste des fonctions secondaires . . . . .	59
3.3.2	Exemples de fonctions secondaires . . . . .	60
3.4	Les autres fonctions graphiques . . . . .	62
3.4.1	Les différents types de graphes . . . . .	62
3.4.2	Exemples . . . . .	62
3.4.3	Exemples : graphiques de distribution-comparaison . . . . .	64
3.4.4	Dessiner une fonction . . . . .	64
3.4.5	Légende . . . . .	64
3.5	Périphériques graphiques . . . . .	64
3.6	Gestion couleur . . . . .	65
3.7	Histogramme . . . . .	65
3.8	Camembert . . . . .	66
3.9	Fonction boxplot . . . . .	66
3.9.1	Fonction vioplot . . . . .	67
3.9.2	Comparaison vioplot/boxplot . . . . .	67
3.9.3	Apport ? . . . . .	67
3.9.4	bagplot : un boxplot en 2D . . . . .	68
3.10	Les graphiques treillis . . . . .	69
3.11	Exemples . . . . .	69
3.11.1	Nuages de points . . . . .	69
3.11.2	Graphique 2 séries . . . . .	71
3.11.3	Eclater nuage par rapport à une variable z . . . . .	72
<b>4</b>	<b>Package ggplot2</b>	<b>73</b>
4.1	Introduction . . . . .	73
4.2	qplot() . . . . .	73
4.3	ggplot . . . . .	77
4.3.1	Premiers pas . . . . .	77
4.3.2	Différentes formulations . . . . .	77
4.3.3	Mise en forme . . . . .	78
4.3.4	Sous-ensembles . . . . .	79
4.3.5	Nuages de points . . . . .	80
4.3.6	Histogrammes . . . . .	81
4.3.7	DotPlot . . . . .	82
4.3.8	Courbes . . . . .	82

4.3.9	Camemberts . . . . .	82
4.3.10	Boîtes à moustaches . . . . .	82
4.3.11	Dessin de fonctions . . . . .	83
4.3.12	Graphiques multiples . . . . .	83
4.3.13	Librairie plyr . . . . .	84
4.3.14	Cartographie . . . . .	84
<b>5</b>	<b>Dessins</b>	<b>87</b>
5.1	Graphiques 3D . . . . .	87
5.1.1	Courbes de niveau . . . . .	87
5.1.2	Représentation en 3D . . . . .	88
5.2	Traitement d'image . . . . .	89
5.3	Connexion à Google Map . . . . .	90
<b>6</b>	<b>Analyse des données</b>	<b>93</b>
6.1	Exploration . . . . .	93
6.2	Régression . . . . .	93
6.2.1	Modèle linéaire . . . . .	93
6.2.2	Régression logistique . . . . .	94
6.3	ACP . . . . .	97
6.4	Classifications . . . . .	97
6.4.1	La fonction agnes() . . . . .	97
<b>7</b>	<b>Modélisation</b>	<b>99</b>
7.1	Analyse de la qualité de la régression . . . . .	99
7.1.1	Modèle de croissance exponentiel . . . . .	100
7.1.2	Fonction logistique . . . . .	100
7.1.3	Estimation des paramètres d'une loi de probabilités . . . . .	101
<b>8</b>	<b>Compléments</b>	<b>103</b>
8.1	Commandes de base . . . . .	103
8.2	Touches . . . . .	104
8.3	Aide . . . . .	104



# Introduction

R est un logiciel statistique dérivé de S, langage développé en 1976 chez Lucent Technologies. En 1988, S est devenu un produit commercial appelé SPlus.

R est entièrement libre de droits<sup>1</sup>. Il peut être installé sous différents systèmes (Windows, Linux ou Mac).

R est avant tout conçu pour analyser des données (graphiques, test, modélisation, ...), mais grâce à son langage, il permet également de les manipuler (extraire des sous-ensembles, croiser des tables, créer des variables complémentaires, accéder à des bases de données externes ...). Il permet de traiter tout type de données, y compris celles géoréférencées.

Il est possible de trouver sur internet des interfaces qui peuvent faciliter la prise en main de R.

## R Commander

L'utilitaire R Commander crée une interface permettant d'utiliser des menus plutôt que de taper des lignes de commandes. C'est très pratique pour un utilisateur novice mais retarde l'apprentissage des nombreuses possibilités de R.

Il permet d'importer les données et de réaliser de nombreux traitements.

La fenêtre de script permet de récupérer l'instruction correspondant au menu retenu. On peut s'en inspirer pour modifier la syntaxe et réaliser des opérations qui ne sont pas disponibles dans les menus.

L'utilitaire nécessite un terminal. Dans l'environnement MacOS cela comprend :

- l'application X11 (dans le DVD d'installation de Mac OS X)
- l'environnement tcltk

Pour utiliser R Commander, il faut :

- lancer l'application R
- installer le package correspondant (Rcmdr) en tapant dans la ligne de commande

```
> install.packages("Rcmdr", dependencies=TRUE)
```

L'utilitaire est lancé en tapant dans la ligne de commande

```
> library(Rcmdr)
```

## Rattle

L'utilitaire Rattle crée également une interface permettant d'utiliser des menus plutôt que de taper des lignes de commandes.

Il faut installer le package correspondant en utilisant la commande

```
> install.packages("rattle")
```

Sous MacOS, il semble y avoir un bug lié à la langue. Il faut donc lui dire que le langage est l'anglais.

```
> Sys.setenv(LANGUAGE="en")
```

---

1. Pour citer R dans une publication utilisez la fonction citation()

L'utilitaire est lancé en tapant

```
> library(rattle)
> rattle()
```

## RStudio

C'est également une interface sous forme de logiciel autonome. Par rapport à des logiciels comme Tinn-R<sup>2</sup>, il a l'avantage d'être multiplateforme. Il ne contient pas de script pré-établi pour réaliser des analyses standards, mais il facilite l'utilisation de R. Nous travaillerons avec ce logiciel.

Le site <http://rstudio.org> fournit de la documentation sur ce logiciel.

A l'ouverture, l'écran est divisé en 4 parties : fenêtre de script, console, historique et objets créés, gestion des packages et figures.

Quelques points.

- RStudio gère plusieurs types de documents : R Script, Tex File, R Sweave
- RStudio possède la complémentation automatique : il suffit de taper les premières lettres puis d'appuyer sur la touche tabulation pour faire apparaître les fonctions disponibles. Cela marche également pour les arguments.
- Dans la console, la flèche vers le haut permet de rappeler une commande précédente. Si on tape dans la console une chaîne de caractères, la combinaison de touche ctrl+flèche vers le haut rappelle les commandes précédentes qui contiennent cette chaîne de caractères.
- RStudio peut transformer automatiquement des lignes sélectionnées en fonction, avec le menu Code - Extract function. Il transforme automatiquement les paramètres en arguments.

---

2. Tinn-R est u éditeur pour R très utilisé dans l'environnement Windows



# Chapitre 1

## Premiers pas

### 1.1 Premier script

```
> n <- 10
> x <- runif(n)
> y <- 1 - 2 * x + rnorm(n)
> res <- lm(y~x)
> plot(x,y)
> plot(y ~ x)
> plot.lm(res)
```

Dans ce script,

- le symbole "<-" signifie affectation<sup>1</sup>.
- Si un objet n'existe pas l'affectation le crée. Sinon l'affectation écrase la valeur précédente.
- la fonction runif génère des valeurs aléatoires<sup>2</sup>.
- la fonction rnorm génère des valeurs d'une loi normale de moyenne 0 et d'écart-type 1.
- la fonction lm (linear model) permet des régressions linéaires.
- la fonction plot génère des graphiques.

### Commentaires

- L'assignation peut être remplacée par l'égalité.
- Taper ?runif dans la fenêtre de commande permet d'avoir les différents arguments de la fonction runif. L'aide en ligne indique qu'il y a d'autres fonctions, dunif, punif et qunif, respectivement, fonction densité de probabilité, fonction de répartition et fonction quantile. Elle permet de savoir que runif possède 3 arguments, le nombre de mesures souhaitées (dans notre cas, 10), les bornes inférieure et supérieure de l'intervalle, qui par défaut prennent les valeurs 0 et 1.
- L'aide en ligne permet de savoir que de très nombreuses fonctions de probabilité ont été programmées.
- On peut également taper ?rnorm et de manière plus générale chaque fois que l'on souhaite consulter l'aide en ligne.
- Un logiciel classique affiche directement les résultats d'une analyse. Avec R l'affichage d'une analyse est réduit au minimum. C'est à l'opérateur de demander des résultats supplémentaires. C'est pourquoi il est nécessaire de stocker les résultats dans un "objet". Ils pourront être réutilisés dans des analyses ultérieures.
- A titre d'exemple, le résultat de la régression est stocké dans l'objet que nous avons décidé de nommer "res". Pour analyser les résultats d'une régression, il est possible d'utiliser les fonctions suivantes qui correspondent au :

```
> summary(res) # résumé du modèle
> summary(res, cor=T) # résumé du modèle avec coefficient de corrélation des coefficients
```

---

1. En algorithmique, l'affectation signifie qu'au moment de l'instruction, la première variable va prendre la valeur de la seconde. C'est pourquoi l'instruction `x <- x + 2` a un sens alors que l'opérateur égalité (`x = x + 2`) n'en aurait pas.

2. Avec R, une fonction, pour être exécutée, s'écrit toujours avec des parenthèses. Si l'opérateur oublie les parenthèses, R affichera le contenu des instructions de cette fonction.

```
> plot(res) # dessin du modèle
> residuals(res) # affichage des résidus
> coefficients(res) # coefficients du modèle
> predict(res) # valeurs estimées
```

- La commande `methods` donne la liste des implémentations de cette méthode. exemple : `methods(plot)`
- La fonction `plot` est très puissante. Ses possibilités seront abordées au paragraphe 3.2.

## 1.2 Fonctionnement de R

### 1.2.1 Les extensions

Toutes les librairies ne sont pas chargées au lancement du logiciel. En septembre 2012 il y avait 4035 packages disponibles.

- `library()` retourne la liste des librairies installées.
- `library(LIB)` charge la librairie LIB
- `library(help = LIB)` retourne la liste des fonctions de la librairie LIB. On peut faire la même chose en cliquant sur le nom du package dans la fenêtre Packages
- `search()`, `searchpaths()` retourne la liste des librairies chargées.

### 1.2.2 Les objets R

Nous avons vu qu’avec R les résultats d’un traitement sont stockés dans un “objet”. Tous les objets sont stockés dans un espace de travail. Ils sont présents dans la mémoire vive de l’ordinateur, d’où une très grande rapidité d’exécution<sup>3</sup>. Les commandes `ls()` ou `rm()` permettent respectivement de les lister ou de supprimer l’objet cité dans la parenthèse. On peut par exemple, à partir de fichiers bruts, créer des tables intermédiaires qui seront ensuite analysées. Ces tables intermédiaires peuvent être sauvegardées en tant qu’image (format binaire très compressé) et rechargées à volonté.

La fenêtre “Workspace” permet également de supprimer tout ou partie des objets.

Il existe 2 types d’objets :

- Objets simples (ou atomiques) : ils ne comportent que des éléments de même mode.
- Objets composés (ou hétérogènes) : ils comportent des éléments qui peuvent être de modes différents.

Remarque : les objets dont le nom commence par un point sont cachés.

### 1.2.3 Vectorisation

Le vecteur est l’unité de base de R : un scalaire (entier ou réel) est considéré comme un vecteur de taille 1, et une matrice (tableau) comme une collection de vecteurs (les colonnes).

R permet de travailler directement sur un vecteur sans être obligé de faire une boucle. C’est une approche très très puissante.

```
> tailles <- c(167, 192, 173, 174)
> poids <- c(86, 74, 83, 50)
> imc <- poids/tailles^2
> imc # visualisation des résultats
```

```
[1] 0.003083653 0.002007378 0.002773230 0.001651473
```

Taille, poids et imc sont des objets. Le nom d’un objet doit obligatoirement commencer par une lettre et peut comporter des chiffres et des points. R distingue les majuscules des minuscules. Certains noms (`c`, `q`, `t`, `C`, `D`, `F`, `I`, `T`,) sont utilisés par R. Afin d’éviter des confusions, il est préférable de ne pas les utiliser.

---

3. R chargeant l’intégralité des données en mémoire vive, une machine relativement puissante peut être nécessaire pour travailler sur de gros fichiers.

R conserve un historique des commandes entrées. Il peut être supprimé en totalité ou pour partie. Il est également possible de l'enregistrer.

Il est possible de rajouter des commentaires. La caractère qui marque le début d'un commentaire s'obtient sur Mac avec la combinaison de touches commande - majuscule plus touche " C ".

NB : En cas de problème on peut quitter R en tapant ctrl + C ou bien Esc. L'ajout de l'extension ".R" au nom du fichier permet à R de reconnaître le fichier correspondant comme un script.

#### 1.2.4 Aide

Pour les mots-clefs du langage, ou pour des commandes non alphanumériques, il faut mettre des guillemets.

`?"for"`

`?"["`

Si on ne connaît pas le nom de la commande :

`apropos("stem")`

`help.search("stem")`

`help.start()`

#### 1.2.5 Organisation des répertoires

R permet le libellé relatif des répertoires. A titre d'exemple d'organisation, les données seront toujours dans ../data, les scripts dans ../scripts, etc. Les 2 points permettent de remonter d'un niveau.

Les fonctions `setwd()` et `getwd()` permettent respectivement de définir ou de connaître le répertoire de travail actif.

```
> getwd() # Quel est le répertoire de travail ?
> setwd("../AFI/Traitement") # Par rapport au dossier actif le nouveau dossier de travail est
> # le sous-dossier Traitement du dossier AFI (à condition que ces dossiers existent !).
> setwd("~/") # Permet de revenir au dossier de l'utilisateur
```

La commande `dir` permet de lister un répertoire.

```
> dir("~/Documents") # Liste le répertoire Documents de l'utilisateur
> dir("~/Documents", pattern=".doc") # idem mais que les .doc
```



## Chapitre 2

# Manipulation des données

### 2.1 Import/export

Les fichiers de données peuvent être lus en local ou sur un serveur distant via internet. Chaque fichier lu va être affecté à un objet.

#### 2.1.1 La fonction read

```
> tab <- read.table("http://pbil.univ-lyon1.fr/R/donnees/bourse.txt", h = T)
> # Par sécurité, fixer le répertoire de travail puis travailler en relatif
> setwd("/Users/Max/Documents/Cours/Statistiques/Logiciels/R/Cours/MonCoursR/Textes")
> # Vous pouvez également télécharger le fichier, puis l'affecter à un objet
> download.file("http://pbil.univ-lyon1.fr/R/donnees/bourse.txt", "../Data/bourse.txt")
> tab <- read.table("../Data/bourse.txt", h = T)
> # Souvent les fichiers sont au format CSV
> tab <- read.csv("../Data/bourse.csv", h = T, sep = ";", dec=",")
> arbres <- read.csv("../Data/arbres.csv", h = T, sep = ";", dec=",")
```

#### Variantes

Elles sont très nombreuses. A titre d'exemple,

- la fonction `read.fwf` sert à lire un fichier où les données ont un format de largeur fixe (fixed width format).
- la fonction `read.dta` du package `foreign` permet de lire des données au format binaire.
- il est possible de lire directement des fichiers au format `.xls` à condition d'installer la librairie `gdata` pour les format `xls` ou la librairie `xlsx` pour le format correspondant.

```
> library(gdata)
> read.xls("../Data/bourse.xls", sheet=1, method="tab")
```

#### 2.1.2 Exportation des données

Dans le script qui suit, la première instruction permet d'enregistrer la table `tab` au format texte avec comme séparateur la tabulation, la seconde enregistre la table `Placettes` au format `csv` avec comme séparateur de colonnes le point-virgule et la virgule comme séparateur décimal.

```
> write.table(tab, "toto.txt", row = F, sep = "\t")
> write.table(Placettes, "../CSV/Placettes.csv", quote=F, sep=";", dec = ",", row.names = F)
```

On peut également utiliser l'instruction `write.csv` ou bien `write.dbf` (pour cette dernière il faut le package `foreign`).

La fonction `save` permet de sauvegarder un objet ou une série d'objets dans un format optimisé. Cet objet R pourra être réutilisé par R dans tout environnement en utilisant l'instruction `load`.

```
> save(Arbres, file="../Tables/TraitTablesArbresAFI.RData")
> load("../Tables/TraitTablesArbresAFI.RData")
```

Il est possible de sauver plusieurs objets à la fois.

```
> save(df, dfbis, file="essai.RData")
```

### 2.1.3 Connexion à une base de données externe

R permet également de se connecter à une base de données professionnelle. Le script ci-après donne un exemple de connexion à une base PostgreSQL.

```
> library(RPostgreSQL)
> drv <- dbDriver("PostgreSQL")
> con <- dbConnect(drv, dbname="BD_TP", user="eleve", password="eleve", host="147.100.156.36")
> dbListTables(con) # ---- liste des tables (pas forcément nécessaire)
> dbListFields(con, "arbres") # ---- liste des champs (pas forcément nécessaire)
> arbresbis <- dbReadTable(con, "arbres")
> tarifs <- dbReadTable(con, "tarifs")
> # ----- On peut également utiliser une requête
> rs <- dbSendQuery(con, "SELECT * FROM arbres")
> arbresbis <- fetch(rs, n = -1) # L'option n=-1 signifie toutes les lignes
> rs <- dbSendQuery(con, "SELECT * FROM tarifs")
> tarifs <- fetch(rs, n = -1)
> # ----- Déconnexion
> dbDisconnect(con)
> rm(con, drv, rs)
```

#### Commentaires

- L'objet `drv` indique le driver à utiliser en fonction du type de base de données : MySQL, Oracle, ODBC, PostgreSQL, SQLite.
- L'objet `con` fixe les paramètres de la connexion. Il faut au préalable que l'administrateur vous ait donné les droits d'accès.
- Les fonctions `dbListTables` et `dbListFields` fournissent respectivement la liste des tables présentes dans la base de données ou la liste des champs disponibles dans une table.
- La fonction `dbReadTable` permet de lire la totalité de la table.
- La fonction `dbSendQuery` envoie une requête en langage SQL. Dans l'exemple, on souhaite récupérer toutes les lignes de la table "arbres", puis de la table "tarifs".
- Le résultat de chaque requête est stocké dans un data.frame (voir paragraphe 2.2.1) grâce à la fonction `fetch`. L'option `n=-1` signifie toutes les lignes, autrement on indique le nombre de lignes que l'on souhaite importer.
- Lorsque l'on a terminé les requêtes on se déconnecte.

On peut réaliser la même chose avec des données géoréférencées grâce au package `rgeos`. Si vous en avez les droits vous pouvez modifier la base de données à l'aide du package `RODBC`.

Si au moment de l'import les caractères spéciaux sont mal traduits, repasser sous LibreOffice et tester différents système d'encodage.

### 2.1.4 Choix du fichier à l'aide d'une boîte de dialogue

```
> file.choose() # ouverture d'une boîte de dialogue
> tab <- read.csv(file.choose(), h = T, sep = ";", dec=",")
> file.show(file.choose()) # affichez le contenu d'un fichier au format .txt
> setwd(dirname(file.choose())) # il faut choisir un fichier existant dans le répertoire
> # le répertoire choisi ne peut donc pas être vide
```

## 2.2 Stockage dans des objets

Les données peuvent être stockées dans un objet qui se caractérise par un type (scalaire, vecteur, tableau, etc.), un mode (numérique, caractère, logique), une taille, une valeur et un nom. Pour connaître le mode d'un objet `x`, on utilise la fonction `mode`. Seuls les tableaux de données et les listes peuvent contenir des éléments de modes différents. La taille d'un objet est le nombre d'éléments qu'il contient. Elle s'obtient à l'aide de la fonction `length`. Pour un tableau ou une matrice on peut utiliser la fonction `dim`.

### 2.2.1 Les 3 principaux objets

#### Data.frame

Les données importées sont souvent transformées en objet `data.frame`. Ce sont des tableaux dont les colonnes sont de nature variable. Les `data.frame` se manipulent à la fois comme des matrices (indices lignes et colonnes) et comme des listes (appel de colonnes par leur nom).

Les `data.frame` peuvent être créés par la procédure suivante

```
> df <- data.frame(
+   group = c(rep('A', 8), rep('B', 15), rep('C', 6)),
+   sex = sample(c("M", "F"), size = 29, replace = TRUE),
+   age = runif(n = 29, min = 18, max = 54)
+ )
```

Il existe un type de `data.frame` particulier, les `data.table` qui ont l'intérêt d'être beaucoup plus rapides pour les sélections de sous-ensembles ou les fusions. Il nécessite la librairie correspondante.

```
> library(data.table)
> dt <- data.table(
+   group = c(rep('A', 8), rep('B', 15), rep('C', 6)),
+   sex = sample(c("M", "F"), size = 29, replace = TRUE),
+   age = runif(n = 29, min = 18, max = 54)
+ )
```

Il est également possible de créer un `data.table` directement à partir d'un `data.frame`.

```
> dt <- data.table(df)
```

L'instruction `tables()` envoie la liste des `data.table` en mémoire.

```
> tables()
```

Quelques exemples de manipulation de `data.table`.

```
> DT = data.table(x=rep(c("a","b","c"),each=3), y=c(1,3,6), v=1:9)
> setkey(DT,x)
> DT["b",] # extract data for key-column = "b"
> DT[,v] # extract the v column
> DT[,w:=1:3] # add a w column
> DT[,sum(v),by=x]
> x <- data.table(a=1:3, b=2:4, key='a')
> y <- data.table(a=1:3, c=c('a','b','c'), key='a')
> merge(x,y)
```

#### Matrice

Les matrices sont des tableaux, mais contrairement aux `data.frames`, leurs éléments sont tous du même type. Il y a aussi un type "array", qui généralise les matrices en dimension quelconque.

Exemple de matrice à 3 dimensions, issu des données disponibles dans R et accessible grâce à la fonction `data`.

```
> data(HairEyeColor)
> HairEyeColor
```

```
, , Sex = Male
```

	Eye			
Hair	Brown	Blue	Hazel	Green
Black	32	11	10	3
Brown	53	50	25	15
Red	10	10	7	7
Blond	3	30	5	8

```
, , Sex = Female
```

	Eye			
Hair	Brown	Blue	Hazel	Green
Black	36	9	5	2
Brown	66	34	29	14
Red	16	7	7	7
Blond	4	64	5	8

## Liste

Une liste est une structure qui regroupe des objets pas nécessairement de même type (scalaire, vecteur, chaînes de caractères, listes...). Les listes permettent d'automatiser un traitement à l'ensemble des éléments d'une liste. Elles sont souvent utilisées par les différentes fonctions comme élément de stockage des résultats. A titre d'exemple utilisez l'objet "res" généré au paragraphe 1.1 pour avoir un aperçu d'une liste.

```
> str(res)
```

```
List of 12
```

```
$ coefficients : Named num [1:2] 0.954 -2.485
.. attr(*, "names")= chr [1:2] "(Intercept)" "x"
$ residuals    : Named num [1:10] 0.294 0.313 -0.649 -0.488 -0.287 ...
.. attr(*, "names")= chr [1:10] "1" "2" "3" "4" ...
$ effects      : Named num [1:10] 1.208 1.553 -0.727 -0.577 -0.438 ...
.. attr(*, "names")= chr [1:10] "(Intercept)" "x" "" "" ...
$ rank         : int 2
$ fitted.values: Named num [1:10] -0.189 -0.85 -0.393 -0.495 -1.022 ...
.. attr(*, "names")= chr [1:10] "1" "2" "3" "4" ...
$ assign       : int [1:2] 0 1
$ qr           :List of 5
..$ qr        : num [1:10, 1:2] -3.162 0.316 0.316 0.316 0.316 ...
.. .. attr(*, "dimnames")=List of 2
.. .. ..$ : chr [1:10] "1" "2" "3" "4" ...
.. .. ..$ : chr [1:2] "(Intercept)" "x"
.. .. attr(*, "assign")= int [1:2] 0 1
..$ qraux: num [1:2] 1.32 1.33
..$ pivot: int [1:2] 1 2
..$ tol   : num 1e-07
..$ rank  : int 2
.. attr(*, "class")= chr "qr"
$ df.residual : int 8
$ xlevels     : Named list()
$ call        : language lm(formula = y ~ x)
$ terms       :Classes 'terms', 'formula' length 3 y ~ x
.. .. attr(*, "variables")= language list(y, x)
.. .. attr(*, "factors")= int [1:2, 1] 0 1
.. .. .. attr(*, "dimnames")=List of 2
.. .. .. ..$ : chr [1:2] "y" "x"
```



```

.. .. .. $ : chr "x"
.. ..- attr(*, "term.labels")= chr "x"
.. ..- attr(*, "order")= int 1
.. ..- attr(*, "intercept")= int 1
.. ..- attr(*, "response")= int 1
.. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
.. ..- attr(*, "predvars")= language list(y, x)
.. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
.. .. ..- attr(*, "names")= chr [1:2] "y" "x"
$ model      : 'data.frame':      10 obs. of  2 variables:
..$ y: num [1:10] 0.105 -0.537 -1.042 -0.983 -1.31 ...
..$ x: num [1:10] 0.46 0.726 0.542 0.583 0.795 ...
..- attr(*, "terms")=Classes 'terms', 'formula' length 3 y ~ x
.. .. ..- attr(*, "variables")= language list(y, x)
.. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
.. .. ..- attr(*, "dimnames")=List of 2
.. .. .. $ : chr [1:2] "y" "x"
.. .. .. $ : chr "x"
.. .. ..- attr(*, "term.labels")= chr "x"
.. .. ..- attr(*, "order")= int 1
.. .. ..- attr(*, "intercept")= int 1
.. .. ..- attr(*, "response")= int 1
.. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
.. .. ..- attr(*, "predvars")= language list(y, x)
.. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
.. .. ..- attr(*, "names")= chr [1:2] "y" "x"
- attr(*, "class")= chr "lm"

```

## 2.2.2 Inspection

Les fonctions présentées au tableau n°2.1 peuvent s'appliquer à l'ensemble d'un tableau ou à un seul vecteur, à une seule colonne. Il faut pour cela utiliser les règles d'accès à une colonne développées au paragraphe 2.2.4.

TABLE 2.1 – Principales commandes pour inspecter les données.

Dimension d'un tableau	<code>dim(tab)</code>
Résumé	<code>summary(tab)</code>
Liste des variables	<code>names(tab)</code>
Premières lignes	<code>head(tab)</code>
Nombre d'éléments	<code>length(tab)</code>
Nombre de modalités	<code>table(tab[,2])</code>
Infos	<code>str(tab)</code>
Idem mais mieux pour objet plus complexe	<code>unclass(tab)</code>
Nombre de cas NA dans chaque colonne	<code>colSums(is.na(tab))</code>

```

> data(iris)
> str(iris)
> head(iris)
> head(iris,10)
> summary(iris)
> unclass(iris)
> colSums(iris[,1:4])
> table(iris$Species)

```

Le script ci-dessus montre que ces fonctions peuvent s'appliquer à l'ensemble du tableau ou à un sous-ensemble de variables (2 dernières commandes). Il faut pour cela connaître les règles de sélection d'un sous-ensemble présentées au paragraphe 2.4.

### 2.2.3 Conversion

Si suite à l'importation le type de données n'est pas le bon on peut le convertir. La conversion d'un facteur en caractères est fréquemment utilisé lors des recodages du fait qu'il est impossible d'ajouter de nouvelles modalités à un facteur.

- `is.xxx(obj)` teste si `obj` est un objet de type `xxx`. Exemple : `is.real(x)`
- `as.xxx(obj)` contraint (si possible) `obj` au type d'objet `xxx`. Exemple : `as.character(x)`

```
> tab$an <- as.factor(tab$an)
> tab$an <- as.numeric(tab$an)
```

C'est vrai pour une colonne mais aussi pour un objet. On peut ainsi transformer un vecteur ou une matrice en `data.frame` ou vice-versa.

Les deux lignes suivantes montrent comment à partir d'une suite de caractères on peut la transformer en facteur, tout en indiquant l'ordre des différentes modalités.

```
> douleur <- c("rien", "fort", "moyen", "moyen", "leger")
> fdouleur <- factor(douleur, levels = c("rien", "leger", "moyen", "fort"), ordered = TRUE)
```

La fonction `levels` permet de gérer les modalités des facteurs.

```
> x=c(rep(10,3),rep(12,2),rep(13,4))
> xFact=as.factor(x)
> levels(xFact)
```

```
[1] "10" "12" "13"
```

```
> levels(xFact)[1]="12"
> levels(xFact)
```

```
[1] "12" "13"
```

Attention : Les facteurs ne peuvent pas être concaténés avec `c(x, y)`. Il faut pour cela les transformer au préalable en vecteur.

```
> h <- factor(c(as.vector(f), as.vector(g)))
```

### 2.2.4 Accès

Les trois instructions suivantes illustrent différentes façons d'accéder à une colonne d'un `data.frame`.

```
> iris$Sepal.Length
> iris[,2]
> iris[["Sepal.Length"]]
```

La première solution permet de récupérer un élément d'une liste, sans avoir à mettre son nom entre guillemets, contrairement à l'opérateur `[[`. L'intérêt de l'opérateur `[[` est que son argument peut être une variable. L'opérateur `[[` ne permet d'extraire qu'un seul élément à la fois (à la différence de l'opérateur `[` pour les vecteurs).

```
> tab <- iris[2:7,3:5]
> tab
> tab <- iris[, -5]
```

### Reconnaissance automatique des variables

La fonction `attach()` permet de nommer chaque colonne avec son intitulé sans avoir besoin de mettre un dollar devant le nom de la colonne.

La fonction `detach()` permet de faire l'opération inverse.

## 2.3 Création

Nous avons vu que l'opérateur d'assignation `<-` est une façon implicite de créer un objet ; le mode et le type de l'objet sont en effet automatiquement déterminés. On peut aussi créer un vecteur de façon explicite en spécifiant simplement son mode et sa longueur ; la valeur des éléments du vecteur sera alors initialisée selon le mode : 0 pour numérique, FALSE pour logique, "" pour caractère.

```
> x <- vector(mode = "numeric", length = 5)
> x <- matrix(data = 0, nrow = 2, ncol = 3)
> # Création d'un data.frame
> seq1 <- c(0.2, 0.3, 0.4, 0.1)
> seq2 <- c(0.25, 0.35, 0.15, 0.25)
> df <- data.frame(seq1, seq2)
> df <- data.frame(seq1, seq2, row.names = c("A", "C", "G", "T"))
> # Création d'une liste
> L1 <- 58402
> res <- list(L1, seq1)
> res <- list(longueur = L1, composition = seq1) # permet de nommer les différents éléments de la li
> res <- c("Essence", res)
```

```
vector(mode = "numeric", length = 5)
```

### 2.3.1 Fonctions de base

Les opérateurs disponibles sont assez classiques : +, -, \*, /, etc.

`x %% y` modulus ( $x \bmod y$ ) ex : `5%%2` is 1  
`x %/% y` integer division `5%/2` is 2

La fonction `c( )` permet de créer des scalaires ou des vecteurs en listant les valeurs :

```
> x <- c(1,4,9)
> x
```

```
[1] 1 4 9
```

Remarque : la fonction `c( )` concatène des scalaires ou des vecteurs

```
> x=c(1,4,9)
> y=c(x,2,3)
> y
```

```
[1] 1 4 9 2 3
```

On peut de la même façon créer des suites arithmétiques de raison 1 ou -1. Il n'est pas nécessaire que les bornes soient entières.

```
> c(1.4:7)
```

```
[1] 1.4 2.4 3.4 4.4 5.4 6.4
```

```
> c(9:1)
```

```
[1] 9 8 7 6 5 4 3 2 1
```

```
> c(1:3, 10:12)
```

```
[1] 1 2 3 10 11 12
```

La fonction `seq(a,b,t)` généralise le procédé en permettant d'indiquer la raison ( $t$ ).

```
> seq(1,4,0.1)
```

```
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8
[20] 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0
```

La fonction `rep(y,n)` permet de créer un vecteur constitué de l'élément  $y$  répété  $n$  fois,  $y$  pouvant être un scalaire ou un vecteur.

```
> rep(c(1,2),4)
```

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

```
> rep(1:5, length = 12)
```

```
[1] 1 2 3 4 5 1 2 3 4 5 1 2
```

```
> rep(1:5, 2)
```

```
[1] 1 2 3 4 5 1 2 3 4 5
```

```
> rep(1:5, each = 2)
```

```
[1] 1 1 2 2 3 3 4 4 5 5
```

```
> rep(c("un", "deux"), c(6, 3))
```

```
[1] "un" "un" "un" "un" "un" "un" "deux" "deux" "deux"
```

La fonction `gl` permet de générer une variable sous forme de facteur.

```
> gl(2,5,10, labels=c("Che","Het"))
```

```
[1] Che Che Che Che Che Het Het Het Het Het
Levels: Che Het
```

```
> (1:10)^2 # carré des 10 premiers entiers
```

```
[1] 1 4 9 16 25 36 49 64 81 100
```

```
> (1:10)^2 - 1 # carré des 10 premiers entiers moins 1
```

```
[1] 0 3 8 15 24 35 48 63 80 99
```

```
> cumprod(1:10) # produit cumulé des 10 premiers entiers
```

```
[1] 1 2 6 24 120 720 5040 40320 362880
[10] 3628800
```

Nous avons vu qu'il est possible de générer des valeurs qui suivent une loi de probabilité. L'exemple suivant génère une série de 10 valeurs d'une loi exponentielle de paramètre 3.

```
> rexp(10, rate=3)
```

```
[1] 0.23620994 0.01298597 0.17729340 0.76052215 0.25658907 0.51128304
[7] 0.80384379 0.70319041 0.19439372 0.30752673
```

La fonction `sample` permet de tirer dans un lot un certain nombre de valeurs. L'exemple suivant génère 100 valeurs entre 3 essences.

```
> x <- factor( sample(c("Chêne", "Hêtre", "AF"), 100, replace=T) )
```

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

```
> expand.grid(h=c(60,80), w=c(100, 300), sex=c("Male", "Female"))
```

	h	w	sex
1	60	100	Male
2	80	100	Male
3	60	300	Male
4	80	300	Male
5	60	100	Female
6	80	100	Female
7	60	300	Female
8	80	300	Female

L'exemple suivant permet de générer toutes les combinaisons possibles de 3 variables.

```
> x <- c("A", "B", "C")
> y <- 1:2
> z <- c("a", "b")
> expand.grid(x,y,z)
```

	Var1	Var2	Var3
1	A	1	a
2	B	1	a
3	C	1	a
4	A	2	a
5	B	2	a
6	C	2	a
7	A	1	b
8	B	1	b
9	C	1	b
10	A	2	b
11	B	2	b
12	C	2	b

La fonction `paste` concatène du texte. Par défaut elle insère un espace. Si on ne souhaite pas d'espace ou bien un autre séparateur, il faut l'indiquer.

```
> paste("North","Pole")
```

```
[1] "North Pole"
```

```
> paste("North","Pole",sep="")
```

```
[1] "NorthPole"
```

```
> paste(paste(c("X","Y"),rep(1:5,each=2),sep="_"),"txt",sep=".")
```

```
[1] "X_1.txt" "Y_1.txt" "X_2.txt" "Y_2.txt" "X_3.txt" "Y_3.txt" "X_4.txt"
[8] "Y_4.txt" "X_5.txt" "Y_5.txt"
```

```
> paste(letters[1:10], collapse = "")
```

```
[1] "abcdefghij"
```

```
> paste(letters[1:10], sep= "")
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Insérer dans une série de valeurs

```
> append(1:5, 0:1, after=3)
```

```
[1] 1 2 3 0 1 4 5
```

### 2.3.2 Nouvelle colonne dans un data.frame

Le script suivant montre comment réaliser très facilement cette opération très fréquente. Il n'est pas nécessaire que la nouvelle colonne existe.

```
> arbres$Diam99 <- (arbres$Diam1_99 + arbres$Diam2_99)/2
```

La nouvelle colonne peut être le résultat d'une fonction complexe de plusieurs colonnes. Vous pouvez également utiliser la fonction `transform` qui est peut-être plus lisible.

```
> arbres <- transform(arbres,
+                      Diam99 = (Diam1_99 + Diam2_99)/2,
+                      Diam10 = (Diam1_10 + Diam2_99)/2
+ )
```

### 2.3.3 Transformation d'une variable continue en classes

On utilise la fonction `cut` qui possède un certain nombre d'arguments dont l'argument `breaks` qui indique soit le nombre souhaité de classes, soit les limites des classes sous forme d'un vecteur. A titre d'exemple, le script ci-après permet de créer la variable catégorie de diamètre.

```
> arbres$Cat <- cut(arbres$Diam99, breaks = c(0, 17.5, 27.5, 47.5, 200),
+                  labels = c("PER", "PB", "BM", "GB"), include.lowest = T, right = F)
```

La fonction `cut` convertit un vecteur numérique en facteur. L'argument

- `breaks` fournit les limites des classes sous forme de liste (cf-paragraphe 2.3.1),
- `include.lowest` indique si la liste des limites contient la plus petite des valeurs.
- `right` indique si la borne droite de l'intervalle est incluse ou non. Si oui dans ce cas la borne gauche ne l'est pas.
- `labels` permet de modifier les noms de modalités attribués aux classes.

Limites selon des classes d'égale effectif (utilisation des quantiles)

```
> vol <- runif(100, 0.1, 5)
> df <- as.data.frame(vol)
> decoupe <- quantile(df$vol, probs=seq(0,1,length=4))
> df$clas <- cut(df$vol, breaks=decoupe, include.lowest=T, labels =c("Vol1","Vol2","Vol3"))
> # Si on souhaite fusionner 2 niveaux, il suffit de leur donner le même nom
> df$clas2 <- df$clas
> levels(df$clas2) <- c("Vol1","Vol2","Vol1")
> # Modification ordre des facteurs
> df$MaClas <- factor(df$clas, levels = c("Vol2","Vol1","Vol3"))
```

En analyse de variance, le premier niveau est utilisé comme référence.

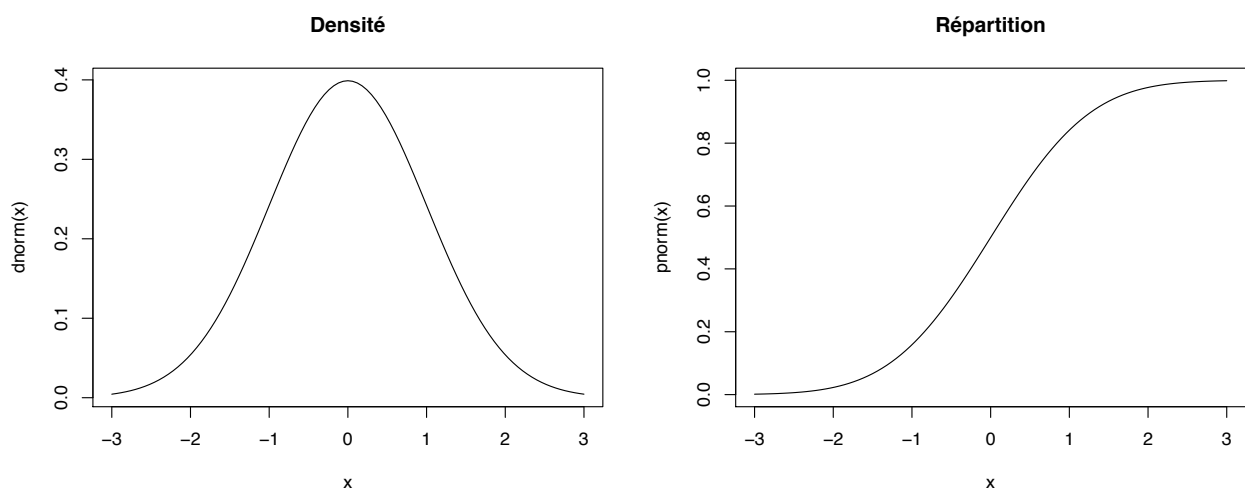
### 2.3.4 Selon une loi de probabilité

Nous avons vu au paragraphe 1.1 que l'on pouvait avoir accès pour de nombreuses lois de probabilité à leur fonction de densité, de répartition, etc.

```
> qnorm(0.025)
```

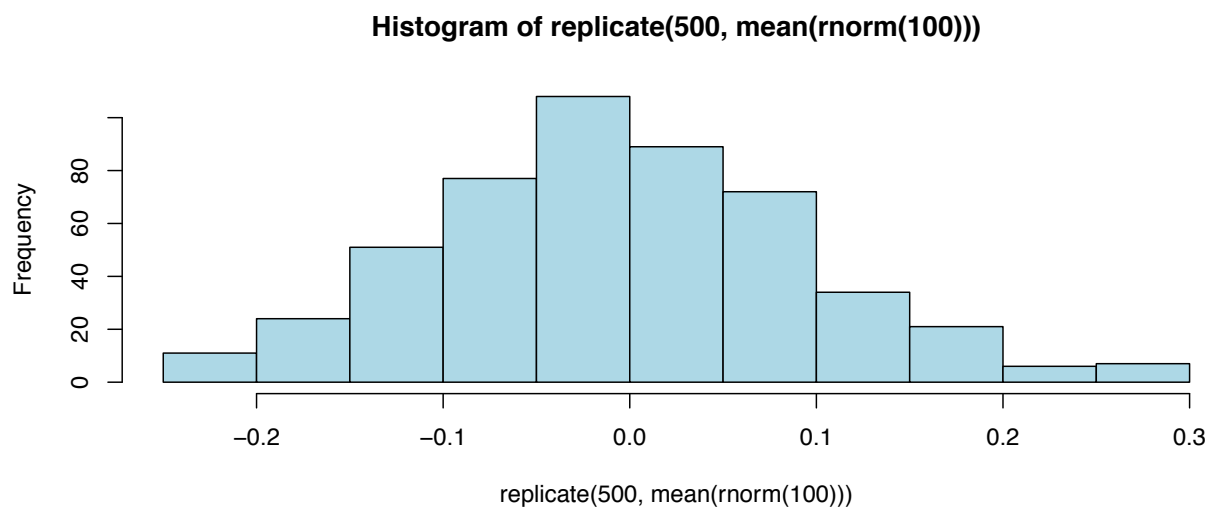
```
[1] -1.959964
```

```
> par(mfrow = c(1,2)) # mise en page
> curve(dnorm(x), xlim=c(-3,3))
> title(main="Densité")
> curve(pnorm(x), xlim=c(-3,3))
> title(main="Répartition")
```



La fonction par sera longement détaillée au paragraphe 3.1.2. La fonction curve permet de dessiner une fonction sans être obligé de générer toutes les valeurs. La fonction replicate permet de recommencer plusieurs fois une opération. Exemple

```
> hist(replicate(500, mean(rnorm(100))), col = "lightblue")
```



### 2.3.5 Création de data.frame

```
> numPeople <- 10
> sex <- sample(c("male","female"),numPeople,replace=T)
> age <- sample(14:102, numPeople, replace=T)
> income <- sample(20:150, numPeople, replace=T)
> minor <- age<18
> population <- data.frame(sex=sex, age=age, income=income, minor=minor)
```

### 2.3.6 Création de matrices

Les matrices sont créées avec la fonction `matrix()` à partir d'un vecteur. On doit fixer le nombre de colonnes `ncol` et/ou le nombre de lignes `nrow`. Par défaut la matrice est remplie colonne par colonne. Pour remplir ligne par ligne, on ajoute l'argument `byrow=T`.

```
> mat <- matrix(c(2,3,5,7,11,13,15,17,19),ncol=3, byrow=T)
> mat
```

```
      [,1] [,2] [,3]
[1,]     2     3     5
[2,]     7    11    13
[3,]    15    17    19
```

Attention : si la dimension du vecteur n'est pas égale au produit ( $ncol \times nrow$ ), le logiciel R poursuit le remplissage de la matrice en utilisant le début du vecteur.

```
> matrix(c(2,3,5,7,11),ncol=2, byrow=T)
```

```
      [,1] [,2]
[1,]     2     3
[2,]     5     7
[3,]    11     2
```

Un vecteur peut être transformé en matrice par simple modification de ses dimensions.

```
> x <- 1:15 # permet de générer une série des 15 premiers entiers
> dim(x) <- c(5, 3) # transforme x en matrice de 5 lignes et 3 colonnes
```

Nommer les colonnes d'une matrice

```
> colnames(x)=paste("var",1:3,sep="_")
```

On accède aux valeurs d'une matrice en indiquant les numéros de lignes et/ou de colonnes.

```
> x[1,2]
```

```
var_2
     6
```

```
> x[1,]
```

```
var_1 var_2 var_3
     1     6    11
```

```
> x[,3]
```

```
[1] 11 12 13 14 15
```



Sélectionner les lignes de `mat` quand les valeurs de la deuxième colonne sont  $> 7$

```
> x[x[,2]>7,]
```

```
      var_1 var_2 var_3
[1,]      3      8     13
[2,]      4      9     14
[3,]      5     10     15
```

La fonction `diag` retourne une matrice diagonale lorsque le paramètre d'entrée est un vecteur. Si le paramètre d'entrée est une matrice, alors elle retourne un vecteur constitué de la diagonale de la matrice.

```
> diag(1:4)
```

```
      [,1] [,2] [,3] [,4]
[1,]     1     0     0     0
[2,]     0     2     0     0
[3,]     0     0     3     0
[4,]     0     0     0     4
```

```
> diag(mat)
```

```
[1]  2 11 19
```

La fonction `outer` permet de créer une matrice dont les valeurs correspondent à la fonction choisie.

```
> f=function(x,y) 0.3 * exp(-0.5 * ((x - 3)^2 + (y - 3)^2)) +
+               0.7 * exp(-0.5 * ((x - 6)^2 + (y - 4)^2))
> x <- seq(-1, 9, length = 100)
> y <- seq(-1, 7, length = 100)
> M=outer(x,y,f)
> plot(M)
> image(M)
> persp(M)
```

### 2.3.7 Création de listes

```
> rdn=list(serie=c(1:100), taille=100, type="arithm")
```

Chaque élément de la liste peut être appelé soit en utilisant (son nom précédé du signe `$` ou de son indice entre double crochet `[[.]`)

```
> maliste <- list(a = "CHS", b = "CHP", c = "HET")
> maliste[[1]]
```

```
[1] "CHS"
```

```
> maliste[1]
```

```
$a
[1] "CHS"
```

```
> maliste$d <- 1:10
```

La dernière instruction montre qu'une liste peut convenir divers types d'informations.

### 2.3.8 Création de fonctions

L'utilisateur peut écrire ses propres fonctions. La fonction va retourner le contenu de l'objet situé sur la dernière ligne. On peut spécifier une liste d'éléments à rendre à l'issue de l'exécution : `return`

```
> etendue <- function(x) {
+   return(list(mini=min(x), maxi=max(x)))
+ }
```

Pour pouvoir exécuter une fonction, il faut la charger en mémoire, soit en tapant chacune des lignes au clavier soit en l'écrivant dans un fichier et en "compilant" ce dernier avec la fonction source.

Les variables définies dans les fonctions sont considérées comme locales. Si ces variables existaient déjà alors leur contenu n'est pas modifié. Si elles n'existaient pas, alors elles n'existeront plus après l'appel de la fonction. Si la fonction utilise une variable non définie dans la fonction, alors R va la chercher dans l'environnement extérieur.

Exemple simple :

```
> TarifSch <- function(type, num, diam) {
+   vol <- NA
+   if (type=="SchR") vol = round(5/70000*(8+num)*(diam-5)*(diam-10),3)
+   if (type=="SchL") vol = round(5/90000*(8+num)*diam*(diam-5),3)
+   vol
+ }
> TarifSch("SchR", 8, 45)
```

```
[1] 1.6
```

La fonction ci-dessous est plus complexe. Elle permet de calculer la valeur d'une douglaiaie de 6 ans, en intégrant une incertitude sur le prix unitaires lors de la coupe finale.

```
> MonteCarlo <- function(Choix) {
+ # ----- Initialisation
+ taux <- 0.03
+ Fonds <- 1000
+ Volume <- 400
+ AnneeFin <- 60
+ Age <- 6
+ Annees <- c(0,1,3,5,40,AnneeFin)
+ Montant <- c(-1500, -500, -500, -500, 5000, 0)
+ Depart <- c(31,1)
+ Fin <- c(60, 60)
+ MontantB <- c(-15, 30)
+ moyPU <- 63
+ DistPU <- c(rep(50,5),rep(55,15),rep(60,25),rep(65,30),rep(70,15),rep(75,10))
+ ecart <-0.1
+ Nb <- 1000
+
+ # ----- Incertitude sur les prix
+ PU <- switch(Choix,
+             moyPU, # Choix = 1, certitude
+             sample(DistPU,Nb, replace=T), # Choix = 2, Répartition fournie par op
+             runif(Nb, moyPU*(1-2*ecart), moyPU*(1+2*ecart)), # Choix = 3, Loi uniforme
+             rnorm(Nb, moyPU, moyPU*ecart)) # Choix = 4, Loi normale
+
+ # ----- Calculs
+ BNA <- sum(Montant/(1+taux)^Annees) # Frais variables
+ BNA <- BNA + sum(MontantB*((1+taux)^(Fin-Depart+1)-1)/taux/(1+taux)^Fin) # Frais par période
+ BNA <- BNA + Fonds*(1/(1+taux)^AnneeFin - 1) # Intégration du fonds
+ BNA <- BNA + PU*Volume/(1+taux)^AnneeFin # Intégration de la recette
```

```

+ BASI <- BNA*(1+taux)^AnneeFin/((1+taux)^AnneeFin-1)
+ Valeur <- BASI*(1+taux)^Age
+ # ----- Sorties
+ hist(Valeur)
+ Quant <- quantile(Valeur, probs = c(0.01, 0.1))
+ c(mean(Valeur), Quant, mean(Valeur[which(Valeur<=Quant[2])]))
+ }

```

Cette fonction s'utilise en tapant le nom de la fonction et une valeur comprise entre 1 et 4 pour le type d'incertitude. Elle renvoie la valeur moyenne, la VaR à 1% et à 10% ainsi que la moyenne des valeurs inférieures à 10%.

```
> MonteCarlo(2)
```

```

              1%      10%
4082.520 2778.525 3266.730 3124.453

```

## 2.4 Extraction/Suppression

### 2.4.1 Objets booléens et instructions logiques

#### Les opérateurs

Les opérations logiques : `<` , `>` , `<=` , `>=` , `!=` [différent], `==` [égal] retournent TRUE ou FALSE.

Les opérateurs ET (&) et OU (|)<sup>1</sup> permettent de définir plusieurs conditions. Exemple, extraire les composantes `>8` ou `<2`, puis celles qui sont `>8` et `<2`.

```

> x <- c(1:10)
> x[(x>8) | (x<2)]

```

```
[1] 1 9 10
```

```
> x[(x>8) & (x<10)]
```

```
[1] 9
```

On peut réaliser un test simultané de deux conditions, ou bien un test séquentiel de deux conditions (la deuxième expression n'est évaluée que si la première est vraie), on utilise pour cela un double opérateur.

#### Les comparaisons

L'opérateur `==` (resp `!=`) vérifie l'égalité (resp. la différence) de deux vecteurs en faisant une comparaison terme à terme : il retourne un vecteur logique. Pour effectuer une comparaison globale d'objets, il faut utiliser la fonction `identical`. Si les vecteurs ne sont pas de même longueur, le plus court est complété automatiquement.

```

> x <- seq(2, 10, by = 2)
> y <- 2 * (1:5)
> identical(x, y)

```

```
[1] TRUE
```

---

1. Sous MacOS l'opérateur OU s'obtient par la combinaison de touches Shift + alt + L

### 2.4.2 Indexation directe

Les crochets simples permettent de faire un sous-ensemble ligne ou colonne.

```
> x <- seq(1,20,2)
> x[c(1:2,5)] # lignes 1, 2 et 5

[1] 1 3 9

> x[1:3] # Les trois premières lignes

[1] 1 3 5

> x[-4] # Tous sauf le quatrième élément de x

[1] 1 3 5 9 11 13 15 17 19

> x[-c(1, 6)] # supprime la première et la 6ème ligne

[1] 3 5 7 9 13 15 17 19

> x[-length(x)] # Tous les éléments de x sauf le dernier

[1] 1 3 5 7 9 11 13 15 17

> x[x>=5] <- 0 # Remplace toutes les valeurs supérieures à 5 par 0
> x

[1] 1 3 0 0 0 0 0 0 0 0

> mat[,1]

[1] 2 7 15

> mat[1:2,2:3]

      [,1] [,2]
[1,]    3    5
[2,]   11   13

> df <- data.frame(cbind(x=1, y=1:10), fac=sample(1:20, 10, replace=TRUE))
> df[1] # retourne un data.frame

      x
1    1
2    1
3    1
4    1
5    1
6    1
7    1
8    1
9    1
10   1

> df[,1] # retourne un vecteur

[1] 1 1 1 1 1 1 1 1 1 1
```

Attention : `vect[-j]` retourne le vecteur `vect` sans la `j` ème coordonnée.

Pour accéder aux objets d'une liste on utilisera soit les crochets simples (dans ce cas une liste est retournée), soit les crochets doubles (dans ce cas l'objet est extrait).

### 2.4.3 Indexation par nom

```
> mean(arbres$Diam1_99, na.rm = TRUE) # sélection équivalente à arbres[, "Diam1_99"]
```

```
[1] 35.00082
```

Il est possible de sélectionner plusieurs colonnes sur le modèle de l'exemple suivant :

```
> df[, c("id", "sexe", "age")]
```

### 2.4.4 Indexation par condition

Les opérateurs suivant peuvent être utilisés.

==	égalité
&	et
	ou
!=	différent

Exemple

```
> arbres[arbres$Essence == "CHE",]
```

Attention : l'opérateur == pour tester l'égalité de valeurs numériques peut poser problème. La fonction all.equal() permet de faire des comparaisons raisonnables ( $10^{-16}$ ). Idem mais avec la fonction subset qui a comme avantage d'éliminer les lignes avec données manquantes.

```
> arbres$Distance <- as.numeric(arbres$Distance)
> arbresCHE <- subset(arbres, Essence == "CHE" & Distance < 10)
```

Intérêt de la variable subset : on peut spécifier le nom des variables sans guillemet et leur appliquer directement l'opérateur d'exclusion -.

```
> d2 <- subset(d, select = c(sexe, sport))
> d2 <- subset(d, age > 25, select = -c(id, age, bricol))
```

Il est préférable d'accompagner la fonction subset de la fonction droplevels. Elle sert à effacer les modalités qui ne sont plus utilisées.

```
> t <- subset(t, Type=="A")
> t <- droplevels(t)
```

Suppression de toutes les lignes dont la colonne Distance est vide

```
> arbres <- arbres[arbres$Distance != "",]
```

Sélection d'un sous-tableau contenant toutes les lignes mais dont les colonnes ont un nom qui commence par "D" ou "S".

```
> tab[, substr(names(tab), 0, 1) %in% c("D", "S")]
```

Remarque : la fonction substr permet d'extraire des caractères dans une chaîne de caractères. Les fonctions de manipulation de chaîne de caractères seront développées au paragraphe 2.9.

Attention :

Par exemple df\$V1 et df[, 1] sont des vecteurs alors que df["V1"] est un data.frame.

L'instruction with permet de mettre en dénominateur commun une sélection

```
> with(tab[tab$C13 > 1.0,], plot(C13, HTOT))
> t$Diam <- with(t, (Diam1+Diam2)/2)
```

### 2.4.5 Fonction which

La fonction `which(vec)` retourne les indices des coordonnées du vecteur `vec` qui prennent la valeur `TRUE`.

```
> x=(1:10)^2
> which(x>15)
```

```
[1] 4 5 6 7 8 9 10
```

```
> names(which(sapply(iris, is.numeric))) # noms des variables numériques
```

```
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
```

```
> numero <- which(sapply(iris, is.numeric)) # numéro des variables numériques
> pairs(iris[, numero]) # Représentation graphique
```

Cas particuliers : `which.max(x)` retourne `which(x==max(x))`

### 2.4.6 Suppression d'une colonne

Le plus simple consiste à y affecter une valeur nulle. Il est également possible de supprimer plusieurs colonnes à la fois.

```
> arbres$Coupe <- NULL
> arbres <- arbres[,-match(c("Statut", "Sante_08"), names(arbres))]
```

## 2.5 Modifications

### 2.5.1 Présentation paysage ou portrait

#### La fonction stack

`stack` reformate un tableau en empilant les colonnes numériques pour faire un vecteur unique et un facteur indiquant les colonnes d'origine. `unstack` fait l'opération inverse.

```
> df <- data.frame(x = rnorm(10), y = rnorm(10, 1), z = rnorm(10, 2))
> tab1 <- stack(df)
> tab2 <- stack(df, select = -z)
> dfbis <- unstack(tab1)
> all.equal(dfbis, df) # Vérification
```

La fonction `stack` est souvent utile dans le cas de l'analyse de variance ou d'un modèle linéaire pour présenter les données non pas sous forme de plusieurs colonnes mais de 2 colonnes.

#### La fonction reshape

La fonction `reshape` du package de base permet de basculer d'un mode d'organisation des données à l'autre. Attention les arguments à utiliser ne sont pas les mêmes selon que l'on souhaite une présentation en paysage ou en portrait.

```
> MyData <- data.frame( Num = 1:4,
+                       Diam99 = c(NA, 34, 17, 45),
+                       Diam10 = c(21, 38, 19, 48),
+                       Diam19 = c(23, 41, 23, 52))
> MyData$Acct1 <- MyData$Diam10 - MyData$Diam99 # Calcul de l'accroissement
> MyData$Acct1 <- NULL # suppression de la colonne
> MyData <- reshape(MyData, varying= 2:4, direction="long", v.names="Diam")
> MyData <- MyData[order(MyData$Num), ]
```

```
> MyData$Num <- as.factor(MyData$Num)
> plot(MyData$Diam ~ MyData$time)
> # ----- autre exemple : présentation paysage
> df3 <- data.frame(school = rep(1:3, each = 4), class = rep(9:10, 6),
+                   time = rep(c(1,1,2,2), 3), score = rnorm(12))
> wide <- reshape(df3, idvar = c("school","class"), direction = "wide")
> wide
```

### Le package reshape

Il est également possible d'utiliser le package reshape. Les fonctions à connaître :

- Melting (fondre) : consiste à transformer les données d'un format "large" à un format long. On doit donner à melt une variable d'identifiants et une liste de variables de mesure. Si on ne spécifie rien, melt considère les variables de type integer ou factor comme des identifiants, et les autres comme des mesures. Toutes les variables de mesure doivent être du même type (factor, numeric...), sinon elles ne pourraient être regroupées dans une même colonne d'un data frame.

Les valeurs manquantes peuvent être codées avec un NA, ou bien par suppression de la ligne correspondante. Le choix se fait avec l'argument na.rm.

```
> library(reshape)
> data(french_fries)
> head(french_fries,4)
> ffm <- melt(french_fries, id=1:4, na.rm=TRUE)
> str(ffm)
```

- Casting (mouler) : permet de réaliser l'opération inverse.

```
> data(smiths)
> smithsm <- melt(smiths, id=c("subject", "time"))
> cast(smithsm)
> cast(smithsm, time + subject ~ variable)
```

les variables time et subject sont conservées en colonnes dans le data frame résultat. Chaque modalité de la variable "variable" est transformée en une colonne.

```
> cast(ffm, treatment ~ ., length)
> cast(ffm, . ~ treatment, length)
> cast(ffm, treatment ~ subject, length)
> # ---- valeurs moyennes pour chaque traitement par individu
> cast(ffm, treatment ~ variable | subject, mean)
> # ---- on peut ajouter des marges aux valeurs obtenues
> cast(ffm, treatment ~ rep, sum, margins=TRUE)
> # ---- On peut agréger avec des fonctions renvoyant plusieurs valeurs
> cast(ffm, treatment ~ ., summary)
> # ---- On peut passer en argument un vecteur de fonction
> cast(ffm, treatment ~ rep, c(min, max))
```

Les lignes ci-dessus sont à rapprocher des tableaux croisés sous tableur.

### 2.5.2 Fusion

Il est très souvent nécessaire de fusionner des tables. Ainsi pour calculer les différents volume de la table arbres, il sera nécessaire de chercher le type et le numéro de tarif dans la table tarif.

La fusion se fait grâce à la fonction merge.

```
> arbres <- merge(arbres, tarif, by.x = c("NumDisp","Essence"), by.y = c("Disp","Essence"), all.x = TRUE)
```

La fonction `merge` nécessite moins d'arguments si les colonnes qui servent de clé à la fusion portent le même nom. Le script suivant montre comment modifier l'intitulé d'une colonne d'un `data.frame`.

```
> # Modification des noms afin de faciliter les fusions
> names(tarif)[1] <- "NumDisp"
> arbres <- merge(arbres, tarif, by = c("NumDisp","Essence"), all.x = T)
```

Vous pouvez changer automatiquement tout ou une partie des noms à condition que la règle soit simple.

```
> anciensNoms <- names(arbres)
> names(arbres) <- paste(anciensNoms, "_1", sep="")
```

### 2.5.3 Aggrégation

#### Aggregate

Elle peut être faite grâce à la fonction `aggregate` qui fait partie du package de base. L'argument `na.rm=T` signifie que les valeurs absentes sont négligées.

```
> Placettes <- aggregate(Arbres[,c("Gha","VSchHa","ValCha","Gainha","ValPha","TotalCarbha")],
+ by = list(Arbres$Placette,Arbres$Cycle,Arbres$NumDisp), FUN = sum, na.rm=T)
> names(Placettes)[1:3] <- c("Placette","Cycle","NumDisp")
```

La deuxième instruction est nécessaire car la fonction `aggregate` n'affecte pas automatiquement leur nom aux colonnes qui servent à faire le regroupement.

#### SummaryBy

La fonction `SummaryBy` du package `doBy` est intéressante à connaître. Dans le script suivant sont calculés les moyennes, variance et nombre de valeurs des variables `Weight` et `Feed` par classes de `Evit` et `Cu`.

```
> library(doBy)
> data(dietox)
> dietox12 <- subset(dietox,Time==12)
> summaryBy(Weight + Feed ~ Evit + Cu, data=dietox12, FUN=c(mean,var,length))
```

La fonction `SummaryBy` retranscrit automatiquement leur nom aux colonnes qui servent à faire le regroupement.

#### By

la fonction `By` est équivalente à la fonction `aggregate`. Le `data.frame` est éclaté en sous tableaux correspondant à chaque modalité du facteur retenu, puis la fonction est appliquée à chacun d'eux. Elle renvoie un objet de classe "by".

```
> attach(iris)
> Moy <- by(Sepal.Length, Species, mean)
> liste <- list(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width)
> names(liste) <- names(iris)[-5]
> lapply(liste, mean)
```

```
$Sepal.Length
[1] 5.843333
```

```
$Sepal.Width
[1] 3.057333
```

```
$Petal.Length
[1] 3.758
```

```
$Petal.Width
[1] 1.199333
```



```
> lapply(liste, quantile, probs = c(0, .33, 0.66, 1))
```

```
$Sepal.Length
  0%   33%   66%  100%
4.300 5.400 6.234 7.900
```

```
$Sepal.Width
  0%   33%   66%  100%
 2.0   2.9   3.2   4.4
```

```
$Petal.Length
  0%   33%   66%  100%
1.000 2.087 4.834 6.900
```

```
$Petal.Width
  0%   33%   66%  100%
0.100 0.668 1.600 2.500
```

Remarque : pour `by` le second argument est soit un facteur ou une liste de facteurs, alors que pour `aggregate` les facteurs sont obligatoirement passés sous forme de liste, même s'il n'y en a qu'un.

### **data.table**

Le package `data.table` offre une alternative très très intéressante sur des gros tableaux. Sa syntaxe s'appuie sur l'utilisation classique des tableaux du type `A[lig,col]`. Les `data.table` sont des `data.frame`. Par conséquent on peut leur appliquer les opérateurs classiques (`summary`, `str`, `head`, ...).

Les scripts ci-après donnent un aperçu des possibilités.

```
> DT = data.table(x=rep(c("a","b","c"),each=3), y=c(1,3,6), v=1:9)
> # ----- Extraction simple
> DT[2]                      # 2nd row
> DT[,v]                     # v column (as vector)
> DT[,list(v)]               # v column (as data.table)
> # ----- Création d'un ou plusieurs index
> setkey(DT,x)               # Une seule clé.
> setkey(DT,x,y)             # Deux clés
> setkeyv(DT,c("x","y"))     # Idem
> # ----- Extraction lignes
> DT["a"]                   # recherche binaire (rapide)
> DT[x=="a"]                 # vector scan (slow)
> DT[J("a",3:6)]            # join 4 rows (2 missing)
> DT[J("a",3:6),nomatch=0]   # remove missing
> DT[J("a",3:6),roll=TRUE]   # rolling join (lof)
> DT[!"a"]                   # toutes sauf
> DT[!2:4]                   # all rows other than 2:4
> DT[!J("b",3)]              # same result but much faster
> DT[,m:=mean(v),by=x][]     # add new column by reference by group
> # NB: postfix [] is shortcut to print()
> # ----- Gestion données
> DT[,z:=42L]                # add new column by reference
> DT[,z:=NULL]               # remove column by reference
> DT["a",v:=42L]             # subassign to existing v column by reference
> DT["b",v2:=84L]            # subassign to new column by reference (NA padded)
> DT[,.SD[which.min(v),by=x][] # nested query by group
> # ----- Aggregation
> DT[,sum(v),by=x]           # keyed by
```

```

> DT[,sum(v),by=y]           # ad hoc by
> DT["a",sum(v)]            # j for one group
> DT[c("a","b"),sum(v)]     # j for two groups
> DT[,sum(v),by=list(y%%2)] # Somme de la colonne v pour les valeurs paires ou impaires de y
> DT[,.SD[2],by=x]          # 2nd row of each group
> DT[,tail(.SD,2),by=x]     # last 2 rows of each group
> DT[,lapply(.SD,sum),by=x]  # apply through columns by group
> DT[,list(MySum=sum(v), MyMin=min(v), MyMax=max(v)), by=list(x,y%%2)] # by 2 expressions
> # ----- Fusion
> X = data.table(c("b","c"),foo=c(4,2))
> DT[X]                     # join
> DT[X,sum(v)]              # join and eval j for each row in i
> DT[X,mult="first"]        # first row of each group
> DT[X,sum(v)*foo]          # join inherited scope
> DT[,sum(v),x][V1<20]      # compound query
> DT[,sum(v),x][order(-V1)] # ordering results

```

### Que choisir ?

```

> library(doBy)
> EssCat <- summaryBy(Gha + VolHa ~ NumDisp + Placette + Essence + Cat, data=arbres,
+                     FUN= sum, na.rm=T, keep.names=T)
> Ess <- summaryBy(Gha + VolHa ~ NumDisp + Placette + Essence, data=EssCat,
+                  FUN= sum, na.rm=T, keep.names=T)
> Placettes <- summaryBy(Gha + VolHa ~ NumDisp + Placette, data=Ess,
+                         FUN= sum, na.rm=T, keep.names=T)
> Dispositifs <- summaryBy(Gha + VolHa ~ NumDisp, data=Placettes, FUN= c(mean,sd), na.rm=T)
> # -----
> # Autre possibilité plus rapide
> library(data.table)
> Arbres <- data.table(arbres)
> TabEssCat <- Arbres[, list(Gha=sum(Gha, na.rm=T),VolHa=sum(VolHa, na.rm=T)),
+                         by=c("NumDisp","Placette","Essence","Cat")]
> TabEss <- TabEssCat[, list(Gha=sum(Gha, na.rm=T),VolHa=sum(VolHa, na.rm=T)),
+                         by=c("NumDisp","Placette","Essence")]
> TabPlac <- TabEss[, list(Gha=sum(Gha, na.rm=T),VolHa=sum(VolHa, na.rm=T)),
+                         by="NumDisp,Placette"]
> TabDisp <- TabPlac[, list(GhaMoy=mean(Gha, na.rm=T),GhaSd=sd(Gha, na.rm=T),
+                           VolHaMoy=mean(VolHa, na.rm=T),VolHaSd=sd(VolHa, na.rm=T)), by=c("NumDisp")]

```

La syntaxe de `summaryBy` est intéressante. Les `data.table` sont très intéressants pour des gros tableaux. Pour des tableaux de 1 million de lignes et des opérations assez simples comme une somme ou une moyenne, le ratio de temps d'exécution entre `data.table`, `ddply`, `tapply` et `aggregate` est respectivement de 1 - 20 - 25 - 120. Ces ratios peuvent augmenter selon le type d'opérations.

10+ times faster than `tapply()` 100+ times faster than `==` 500+ times faster than `DF[i,j]<-value`

### 2.5.4 Concaténer des vecteurs/matrices

La fonction `rbind` permet de fusionner des lignes, la fonction `cbind` des colonnes. Cette dernière ne peut fonctionner que si les deux tableaux ont exactement le même nombre de lignes, et n'a de sens que si les lignes sont dans le même ordre.

```

> x=1:10
> y=x^2
> rbind(x,y)

```

```

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x      1    2    3    4    5    6    7    8    9   10
y      1    4    9   16   25   36   49   64   81   100

```

```
> cbind(x,y)
```

```

      x    y
[1,]  1    1
[2,]  2    4
[3,]  3    9
[4,]  4   16
[5,]  5   25
[6,]  6   36
[7,]  7   49
[8,]  8   64
[9,]  9   81
[10,] 10  100

```

Le vecteur y correspond au carré terme à terme du vecteur x. On peut généraliser le procédé à d'autres fonctions (sqrt, abs, sin, cos, tan, exp, log, log10, gamma, ...). Elles seront appliquées à chacun des termes du vecteur ou de la matrice.

## 2.6 Calculs

### 2.6.1 Fonctions disponibles

Les fonctions qui suivent retournent un scalaire	
sum(x)	somme des éléments de x
prod(x)	produit des éléments de x
mean(x)	moyenne
sd(x)	écart-type
cov(x), cor(x)	covariance ou corrélation
max(x)	maximum
min(x)	minimum
abs(x)	valeur absolue
sqrt(x)	racine carrée
round, floor, signif	arrondi
log(x, base)	log dans la base spécifiée
length(x)	longueur du vecteur
dim()	dimension de la matrice
ncol(), nrow()	nombre de colonnes et de lignes
Les fonctions qui suivent retournent un vecteur	
cumsum()	sommes cumulées
cumprod()	produits cumulés
diff()	différences successives
rank(x)	rang des éléments de x
match(x, y)	retourne un vecteur de même longueur que x contenant les éléments de x qui sont dans y (NA sinon)
which(x == a)	retourne un vecteur des indices de x pour lesquels l'opération de comparaison est vraie (TRUE), dans cet exemple l'égalité
unique	si x est un vecteur ou un data.frame, retourne un objet similaire mais avec les éléments dupliqués supprimés
order, sort	tri des données
rev(sort(x))	tri décroissant
rank	fournit les rangs
subset(x, ...)	retourne une sélection de x en fonction de critères. Si x est un data.frame, l'option select permet de préciser les variables à sélectionner (ou à éliminer à l'aide du signe -)

Remarque : Il est toujours très utile d'aller voir dans l'aide, les arguments des différentes fonctions. Par exemple la fonction `mean` possède un argument `trim` qui permet de retirer du calcul un certain pourcentage des valeurs extrêmes, ce qui permet de calculer une moyenne robuste.

### 2.6.2 Tri

`sort(x)` trie les éléments `ts` de `x` et retourne le vecteur triés. L'option `decreasing = TRUE` permet de trier dans le sens décroissant. `Order` réalise un tri multiple hiérarchisé sur un ensemble de vecteurs. Exemple : tri de la table `arbres` par placette et espèce.

```
> arbres[order(arbres$Placette, arbres$Essence), ]
```

La fonction `order` renvoie un vecteur contenant les anciens numéros de lignes triés.

```
> x <- runif(10)
> y <- 1:10
> (m <- cbind(x,y))
```

```
      x  y
[1,] 0.6726671 1
[2,] 0.3452254 2
[3,] 0.8786242 3
[4,] 0.4002297 4
[5,] 0.9442145 5
[6,] 0.2655087 6
[7,] 0.6731080 7
[8,] 0.5369807 8
[9,] 0.1471393 9
[10,] 0.5823971 10
```

```
> (z <- m[order(x),])
```

```
      x  y
[1,] 0.1471393 9
[2,] 0.2655087 6
[3,] 0.3452254 2
[4,] 0.4002297 4
[5,] 0.5369807 8
[6,] 0.5823971 10
[7,] 0.6726671 1
[8,] 0.6731080 7
[9,] 0.8786242 3
[10,] 0.9442145 5
```

Autres exemples

```
> attach(mtcars)
> newdata <- mtcars[order(mpg),] # trié par mpg
> newdata <- mtcars[order(mpg, cyl),] # trié par mpg et cyl
> newdata <- mtcars[order(mpg, -cyl),] # trié par mpg et cyl (descendant)
> detach(mtcars)
```

### 2.6.3 Gestion des données manquantes

Par les besoins du cours, le script suivant génère des données manquantes.

```
> x <- rnorm(10,0,1)
> x[c(3,4,6)] <- NA
```

Elles sont représentées par NA. L'indétermination se propage systématiquement dans les calculs. La plupart des fonctions courantes possèdent un paramètre `na.rm` qui permet de ne pas tenir compte des valeurs manquantes lors du calcul demandé. Exemple :

```
> mean(x)

[1] NA

> mean(x, na.rm = TRUE)
```

```
[1] -0.2830401
```

On peut aussi écrire

```
> mean(x[-which(is.na(x))])

[1] -0.2830401
```

### Repérer les données manquantes

```
> x <- rnorm(10,0,1)
> x[c(3,4,6)] <-NA
> which(is.na(x)) # On les retrouve

[1] 3 4 6
```

### Supprimer les données manquantes

Supprimer les données manquantes de tout le tableau

```
> x <- na.omit(x)
```

### L'instruction `complete.cases`

Elle permet de ne pas prendre en compte les lignes où il y a des données manquantes.

```
> str(airquality)

'data.frame':      153 obs. of  6 variables:
 $ Ozone   : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R : int 190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind    : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp    : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month   : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day     : int  1 2 3 4 5 6 7 8 9 10 ...

> str(airquality[complete.cases(airquality), ])

'data.frame':      111 obs. of  6 variables:
 $ Ozone   : int  41 36 12 18 23 19 8 16 11 14 ...
 $ Solar.R : int 190 118 149 313 299 99 19 256 290 274 ...
 $ Wind    : num  7.4 8 12.6 11.5 8.6 13.8 20.1 9.7 9.2 10.9 ...
 $ Temp    : int  67 72 74 62 65 59 61 69 66 68 ...
 $ Month   : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day     : int  1 2 3 4 7 8 9 12 13 14 ...
```

Une autre solution consiste à remplacer les valeurs manquantes par la moyenne de la variable.

```
> head(airquality)
```

```
      Ozone Solar.R Wind Temp Month Day
1       41     190  7.4   67     5   1
2       36     118  8.0   72     5   2
3       12     149 12.6   74     5   3
4       18     313 11.5   62     5   4
5      NA      NA 14.3   56     5   5
6       28      NA 14.9   66     5   6
```

```
> head(apply(airquality, 2, function(x) ifelse(is.na(x), mean(x, na.rm = TRUE), x)))
```

```
      Ozone Solar.R Wind Temp Month Day
[1,] 41.00000 190.0000  7.4   67     5   1
[2,] 36.00000 118.0000  8.0   72     5   2
[3,] 12.00000 149.0000 12.6   74     5   3
[4,] 18.00000 313.0000 11.5   62     5   4
[5,] 42.12931 185.9315 14.3   56     5   5
[6,] 28.00000 185.9315 14.9   66     5   6
```

La fonction `apply` possède 3 arguments : le tableau de données, une option indiquant si le traitement doit être fait en ligne, en colonne ou les deux, et enfin la fonction retenue.

Il est aussi possible de créer un nouveau vecteur ne contenant pas les données manquantes

```
> y <- x[!is.na(x)]
```

## 2.6.4 Opérations sur les Matrices/Vecteurs

### Opérations élémentaires

Les opérations  $+$   $*$   $-$   $/$  entre 2 vecteurs ou matrices de même dimension sont des opérations terme à terme. Si les vecteurs ne sont pas de même longueur, le plus court est complété automatiquement.

```
> x =c(1:5)
> y = c(1:2)
> x + y
```

```
[1] 2 4 4 6 6
```

### Calculs matriciels

Le produit matriciel est obtenu avec	<code>eigen</code>
Calcul des valeurs/vecteurs propres	<code>det</code>
Calcul du déterminant	<code>t(A)</code>
Transposée de la matrice A	<code>chol(X)</code> retourne R telle que $X = R'R$ où R est une matrice triangulaire supérieure et R' est la transposée de R.
Décomposition de Choleski	<code>svd(X)</code> retourne (U,D,V) telles que $X = UDV'$ où U et V sont orthogonales et D est diagonale.
Décomposition svd	<code>solve(A)</code>
Inverse de la matrice A	<code>solve(A,b)</code>
x tel que $Ax = b$	

La fonction `sweep` permet de diviser (multiplier, soustraire etc...) chacune des colonnes (ou ligne) par un élément d'un vecteur. L'exemple suivant permet de réduire la matrice.

```
> sweep(mat,2,ecartype,FUN='/')
```

## 2.7 Récurrence

Chaque fois que l'on souhaite exécuter une tâche identique.

### 2.7.1 Boucles et tests

**ifelse**

```
> t$V$Sch <- ifelse (t$Sch=="SchR",
+                   5/70000*(8+t$Num)*(t$Diam-5)*(t$Diam-10),
+                   5/90000*(8+t$Num)*(t$Diam-5)*t$Diam)
```

**if else**

Syntaxe générale :

```
> if (condition) {
+ .... } else
+ { ...}
```

**for**

Syntaxe générale :

```
> for (i in vecteur) {
+ .... }
```

- break permet une sortie anticipée des boucles
- next permet d'aller à la boucle suivante

Exemple : création automatique de 10 vecteurs.

```
> for(i in 1:10) assign(paste("data",i,sep=""),runif(10))
```

Exemple : dessin des arbres d'un dispositif AFI.

```
> # ---- Choix
> Num <- 1
> MonCycle <- 1
> # ---- Coordonnées de tous les arbres
> tab <- subset(arbres, Type=="A" & NumDisp== Num & Cycle== MonCycle,
+             c(Placette,Essence,Qual,Azimut,Distance,Diam1,Diam2))
> tab$Diam <- (tab$Diam1+ tab$Diam2)/2
> tab$X <- tab$Distance * sin(tab$Azimut)
> tab$Y <- -tab$Distance * cos(tab$Azimut)
> tab$Placette <- as.factor(tab$Placette)
> # ---- Dessin
> pdf(file = paste("../PDF/DessinArbresDisp",Num,"Cycle",MonCycle,".pdf",sep=""), width = 9, height = 9)
> Xmax <- floor(max(max(tab$X),-min(tab$X))/5 + 0.5) * 5
> Ymax <- floor(max(max(tab$Y),-min(tab$Y))/5 + 0.5) * 5
> Taille <- max(Xmax,Ymax)
> par(mfrow=c(4,3))
> par(mar = c(2, 2, 2, 1))
> for (k in 1:10) {
+   tab1 <- subset(tab, tab$Placette== k)
+   plot(tab1$X,tab1$Y,pch=19, xlim=c(-Taille,Taille), ylim=c(-Taille,Taille),
+        cex=tab1$Diam/10,col="Green")
+ }
> dev.off()
```

On peut choisir d'envoyer les résultats à l'écran ou directement dans un PDF. L'instruction pdf permet d'enregistrer directement dans ce format et de choisir les dimensions de la fenêtre.

De même dans un script on peut envoyer les résultats dans une nouvelle fenêtre (quartz() sous MacOS, windows() sous windows ou X11() sous linux).

### Les boucles while

Syntaxe générale :

```
> while(condition) {
+ instructions
+ }
```

## 2.7.2 Les fonctions apply, tapply, sapply, lapply, mapply, eapply

Elles évitent d'écrire des boucles. Elles s'appliquent à des matrices ou à des data.frame. Elles dépendent des objets que l'on manipule. Elles génèrent des types d'objets différents en sortie.

### apply

Permet d'appliquer, par ligne et/ou par colonne une même fonction.

Syntaxe générale : apply(X, MARGIN, FUN, ...)

- X est un tableau ou une matrice.
- MARGIN indique si l'action doit être appliquée sur les lignes (1), sur les colonnes (2) ou les deux (c(1,2)).
- FUN est la fonction qui sera utilisée.
- ... sont d'éventuels arguments supplémentaires pour FUN.

```
> M <- matrix(rnorm(20,5,2),nrow=5)
> apply(M,2,mean) # moyenne par colonne
```

```
[1] 5.204071 4.779303 5.535600 5.637132
```

```
> apply(M,2, which.min) # position de la valeur minimale
```

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

La fonction utilisée peut être multiple.

```
> apply(M, 2, function(x) c(mean(x), sd(x), max(x)))
```

La fonction utilisée qui peut être créée pour répondre à des besoins spécifiques.

```
> data(iris)
> # on va fabriquer une fonction qui, pour chaque ligne, nous donnera la somme de Sepal.Length, Sepal.W
>
> masomme <- function(monvec){
+ return(sum(as.numeric(monvec[1:4])))
+ # le as.numeric permet de passer outre la transformation en caractères
+ }
> lasomme <- apply(data.frame(iris), FUN=masomme, MARGIN=1)
> head(cbind(iris, lasomme))
> # on rajoute une colonne avec le résultat et on regarde le début du jeu de données
```

La fonction peut s'appliquer à des matrices de plusieurs dimensions.

```
> M <- array(1:32, dim = c(4,4,2))
> apply(M, 1, sum) # somme des valeurs des 2ème et 3ème dimensions
```



```
[1] 120 128 136 144
> apply(M, c(1,2), sum) # somme des valeurs de la 3ème dimension
```

```
      [,1] [,2] [,3] [,4]
[1,]    18    26    34    42
[2,]    20    28    36    44
[3,]    22    30    38    46
[4,]    24    32    40    48
```

Alternatives : fonctions colMeans, rowMeans, colSums, rowSums.

### sapply ou lapply

Ces deux fonctions s'appliquent à des listes, l'argument MARGIN ne sert plus à rien. La fonction sapply renvoie un vecteur donc plus facile à manipuler.

```
> attach(iris)
```

The following object(s) are masked from 'iris (position 3)':

```
Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species
```

```
> liste <- list(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)
> names(liste) <- names(iris)[-5]
> lapply(liste, mean)
```

```
$Sepal.Length
[1] 5.843333
```

```
$Sepal.Width
[1] 3.057333
```

```
$Petal.Length
[1] 3.758
```

```
$Petal.Width
[1] 1.199333
```

```
> sapply(liste, mean)
```

```
Sepal.Length Sepal.Width Petal.Length Petal.Width
    5.843333    3.057333    3.758000    1.199333
```

```
> lapply(liste, quantile, probs = c(0, .33, 0.66, 1))
```

```
$Sepal.Length
 0%  33%  66% 100%
4.300 5.400 6.234 7.900
```

```
$Sepal.Width
 0%  33%  66% 100%
 2.0  2.9  3.2  4.4
```

```
$Petal.Length
 0%  33%  66% 100%
1.000 2.087 4.834 6.900
```

```
$Petal.Width
 0%  33%  66% 100%
0.100 0.668 1.600 2.500
```

**tapply**

Cette fonction s'applique à un data.frame. Elle peut éviter d'avoir à créer des sous-populations. Elle nécessite trois arguments : un vecteur, un facteur et une fonction. Utile quand on souhaite avoir des résultats par modalité d'un facteur.

```
> tapply(Sepal.Length, Species, mean)
```

```
      setosa versicolor  virginica
      5.006      5.936      6.588
```

Une autre façon de procéder consiste à décomposer un vecteur en liste avec la fonction `split()`, puis à utiliser la fonction `lapply`.

**eapply**

Il est possible sous R de créer son propre environnement.

```
> # a new environment
> e <- new.env()
> # two environment variables, a and b
> e$a <- 1:10
> e$b <- 11:20
> # mean of the variables
> eapply(e, mean)
```

**vapply**

C'est une fonction proche de `sapply` avec comme différence la nécessité de fournir la valeur par défaut.

```
> x <- list(a = 1, b = 1:3, c = 10:100)
> vapply(x, FUN = length, FUN.VALUE = 0)
```

```
a  b  c
1  3 91
```

```
> l <- list(a = 1:10, b = 11:20)
> l.fivenum <- vapply(l, fivenum, c(Min.=0, "1st Qu."=0, Median=0, "3rd Qu."=0, Max.=0))
> l.fivenum
```

```
      a      b
Min.   1.0 11.0
1st Qu. 3.0 13.0
Median  5.5 15.5
3rd Qu. 8.0 18.0
Max.   10.0 20.0
```

La fonction `fivenum` renvoie les 5 valeurs classiques.

**mapply**

Applique la fonction pour le premier élément des différents objets, puis pour le deuxième, etc.

```
> mapply(sum, 1:5, 1:5, 1:5)
```

```
[1]  3  6  9 12 15
```

```
> mapply(rep, 1:4, 4:1)
```

```
[[1]]
[1] 1 1 1 1
```

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

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

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

### rapply

```
> myFun <- function(x){
+   if (is.character(x)){
+     return(paste(x,"!",sep=""))
+   }
+   else{
+     return(x + 1)
+   }
+ }
> #A nested list structure
> l <- list(a = list(a1 = "Boo", b1 = 2, c1 = "Eeek"),
+         b = 3, c = "Yikes",
+         d = list(a2 = 1, b2 = list(a3 = "Hey", b3 = 5)))
> rapply(l,myFun)
```

a.a1	a.b1	a.c1	b	c	d.a2	d.b2.a3	d.b2.b3
"Boo!"	"3"	"Eeek!"	"4"	"Yikes!"	"2"	"Hey!"	"6"

### Eléments de choix

- apply : quant on veut appliquer une ou plusieurs fonctions aux lignes ou aux colonnes d'une matrice.
- lapply : quant on veut appliquer une fonction à tous les éléments d'une liste et avoir en retour une liste.
- sapply : quant on veut appliquer une fonction à tous les éléments d'une liste et avoir en retour un vecteur.
- vapply : idem sapply mais le fait de fournir la valeur par défaut que doit retourner la fonction accélère le code.
- mapply : Applique la fonction pour le premier élément des différents objets, puis pour le deuxième, etc.
- rapply : quant on souhaite appliquer une fonction à des listes imbriquées.
- tapply : quant on souhaite appliquer un calcul à des sous-populations.

### 2.7.3 La fonction summary

Elle fournit en standard un grand nombre d'informations. Il existe une autre version de summary dans le package Hmisc.

```
> library(Hmisc)
> options(digits=3)
> # Création d'un jeu de données'
> sex <- factor(sample(c("m","f"), 500, rep=TRUE))
> age <- rnorm(500, 50, 5)
> treatment <- factor(sample(c("Drug","Placebo"), 500, rep=TRUE))
> df <- data.frame(npatt=factor(1:500), sex, age, treatment)
> summary(age ~ sex)
```

```
age      N=500
```

```
+-----+-----+
|      | |N| age|
+-----+-----+
|sex    |f|259|50 |
|      |m|241|50 |
+-----+-----+
|Overall| |500|50 |
+-----+-----+
```

```
> g <- function(x){
+   c(smean.sd(x), median(x), quantile(x,prob=c(0.25,0.75)))}
> summary(age~factor(sex),fun=g)
```

```
age      N=500
```

```
+-----+-----+-----+-----+-----+-----+
|      | |N| Mean|SD | |25% |75% |
+-----+-----+-----+-----+-----+-----+
|factor(sex)|f|259|50 |5.26|50.2|46.5|53.0|
|          |m|241|50 |5.29|50.0|46.2|53.9|
+-----+-----+-----+-----+-----+-----+
|Overall   | |500|50 |5.27|50.2|46.4|53.4|
+-----+-----+-----+-----+-----+-----+
```

## 2.7.4 Boucles ou Fonctions récurrentes ?

Dans les dernières versions de R les temps d'exécution sont similaires. Attention : si trop de boucles imbriquées, le temps d'exécution peut devenir très long.

## 2.8 Simulation

Générer 20 nombres compris entre 0 et 100 arrondis à 1 chiffre après la virgule.

```
> round(runif(20,0,100), digits=1)
```

```
[1]  4.8 50.0 75.2  5.2 28.7 33.6  9.7 51.2 91.9 21.0 25.5 68.3 87.2 29.6 79.1
[16] 25.1 61.3 10.0 39.4 92.5
```

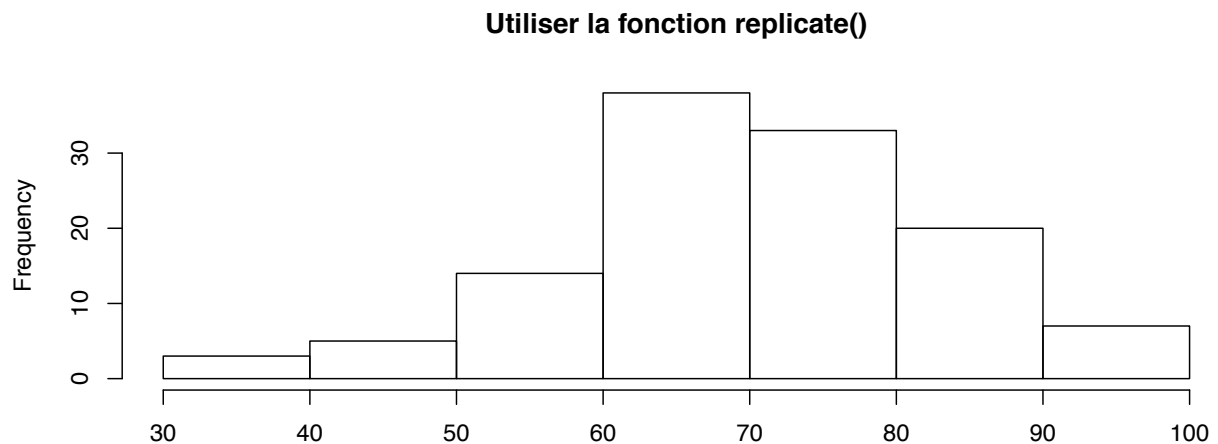
### Bootstrap

Tirage au hasard sans remise de 10 individus dans un tableau de données. L'échantillon ne retient que les colonnes 1 à 3, 5 et 6.

```
> data(swiss) #Swiss Fertility and Socioeconomic Indicators 1888
> don <- swiss
> echant <- sample(1:dim(don)[1], size=10)
> don[echant,c(1:3,5:6)]
```

La fonction replicate permet de générer plusieurs jeux de données d'une même série d'opérations.

```
> data(swiss) #Swiss Fertility and Socioeconomic Indicators 1888
> don <- swiss
> hist(replicate(10,
+   {
+     echant<-sample(1:dim(don)[1], size=12)
+     don[echant,1]
+   } ), main="Utiliser la fonction replicate()", xlab="")
```



## 2.9 Gestion des chaînes de caractères

### 2.9.1 Les chaînes de caractères existantes

```
> letters[1:5]
```

```
[1] "a" "b" "c" "d" "e"
```

```
> LETTERS[1:5]
```

```
[1] "A" "B" "C" "D" "E"
```

```
> month.name[1:5]
```

```
[1] "January" "February" "March" "April" "May"
```

```
> month.abb[1:5]
```

```
[1] "Jan" "Feb" "Mar" "Apr" "May"
```

### 2.9.2 Les fonctions de base

La fonction `cat` permet la concaténation et l'affichage.

```
> x <- choose(6, 0:6)
```

```
> cat("Les combinaisons parmi 6 elements sont : ", x, sep=" ")
```

```
Les combinaisons parmi 6 elements sont : 1 6 15 20 15 6 1
```

Remarque : La fonction `choose` permet d'avoir le nombre de combinaisons.

La fonction `paste` permet l'écriture dans une chaîne de caractères.

```
> num <- 1
```

```
> paste("dataset", num, ".dat", sep="")
```

```
[1] "dataset1.dat"
```

**Autres fonctions**

abbreviate	automatise les abréviations
nbchar	nombre de caractères d'une chaîne
make.unique()	ajoute des suffixes pour que tous les éléments d'un vecteur soit différents.
charmatch()	renvoie la position d'une chaîne de caractères dans une liste.
substr(x, debut, fin)	extraction de caractères
grep(expr, chaine)	renvoie 1 (ou la position dans le vecteur si chaine est un vecteur) si expr est dans chaine, rien sinon.
gsub, sub(expr1, expr2, chaine)	substitue expr2 à expr1 dans chaine (première occurrence pour sub, toutes pour gsub)
strsplit(chaine, expr)	renvoie une liste avec les éléments de chaîne découpés en fonction de expr.

**Exemples**

```
> text <- "Ceci est un chêne"
> vect <- c(rep("Chêne",4), rep("Hêtre",4))
> substr(text,2,4) # donne "eci"
> grep("chêne", text)
> grep("Chêne", vect)
> grep("er$",month.name)
> strsplit(text," ")
> unlist(strsplit(text," ")) # permet de récupérer un vecteur
> paste(unlist(strsplit(text," ")), collapse="-")
```

Certains caractères sont particuliers.

^	début de ligne
\$	fin de ligne
.	n'importe quel caractère
<i>ABC</i>	A ou B ou C

**2.9.3 Librairie stringr**

Détection présence d'une chaîne de caractères dans une autre chaîne.

```
> library(stringr)
> codes <- c("mg1g2kh", "g1h", "mg3","g1","h")
> codeg1 <- str_detect(codes, "g1")
> sum(codeg1)
```

```
[1] 3
```

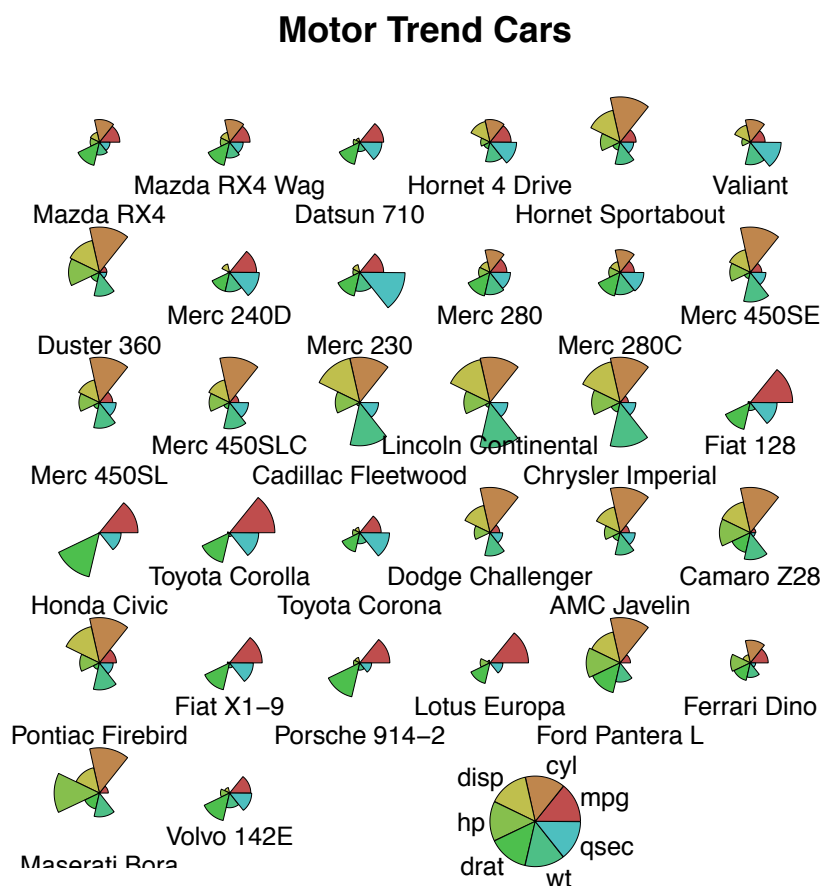
# Chapitre 3

## Graphiques

Les instructions `demo(graphics)`, `demo(plotmath)`, `demo(image)`, `demo(persp)` permettent de voir quelques exemples. Il existe sur internet des sites qui permettent également de récupérer des idées sur des modes de représentations et de télécharger les scripts correspondants<sup>1</sup>.

Voici un exemple de graphique récupéré sur le site dont l'adresse figure en note de bas de page.

```
> palette(rainbow(12, s = 0.6, v = 0.75))
> stars(mtcars[, 1:7], len = 0.8, key.loc = c(10, 1.8), cex=0.8,
+       main = "Motor Trend Cars", draw.segments = TRUE)
```



NB : La fonction `stars` possède de nombreux arguments, en particulier "location" qui permet d'indiquer la position de chaque radar.

---

1. <http://gallery.r-enthusiasts.com>

La création de graphique repose sur de nombreux packages et en particulier :

- package base (Ross Ihaka)
- package grids (Paul Murrel)
- package lattice (Deepayan Sarkar)
- package ggplot2 (Hadley Wickman)

Il existe trois types de fonctions.

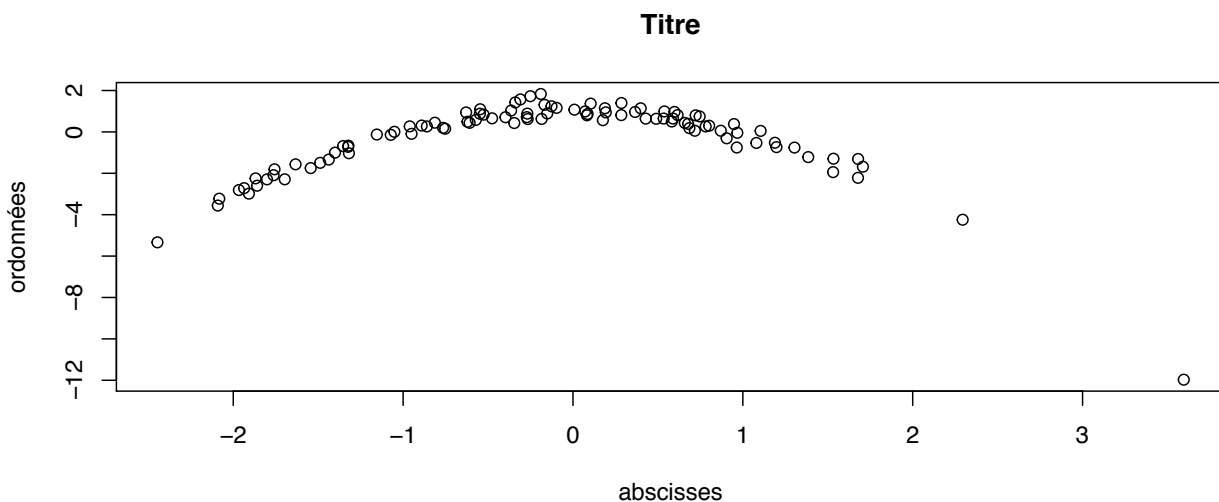
- fonctions graphiques de haut niveau : exemple `plot`
- fonctions graphiques de bas niveau : permettent d'ajouter ou de modifier des éléments à un graphique existant (exemple `points`). Par exemple `abline` ajoute une droite et `axis` permet de définir les axes.
- fonctions interactives : permettent de récupérer des informations sur un graphique existant.

## 3.1 Les bases

### 3.1.1 Un exemple simple

Il repose sur la fonction `plot` qui sera plus largement présentée au paragraphe 3.2.

```
> n <- 100
> x <- rnorm(n)
> y <- 1 - x^2 + 0.3*rnorm(n)
> plot(y ~ x, xlab="abscisses", ylab="ordonnées", main="Titre")
```



### 3.1.2 Paramètres des graphes

La fonction `par()` renvoie la liste de tous les paramètres affectés à un graphique. Exemples :

- `par(mfrow = c(1,2))` permet d'avoir 2 graphiques côte à côte sur la même ligne.
- `par(mar = c(6, 4, 4, 2))` : change les valeurs des marges pour le device ouvert. Les valeurs par défaut sont 5.1, 4.1, 4.1, 2.1 (bottom, left, top, right). L'unité est la ligne.
- `par("cex")` : lit le paramètre `cex` qui permet de modifier la taille de la police (en pourcentage de la taille par défaut).
- `par(cex = 0.5)` : modifie le paramètre.
- `par(new=TRUE)` permet de dessiner le nouveau graphique sur l'ancien.

La liste complète des paramètres modifiables, triée par ordre alphabétique, est la suivante :

```
> sort(names(par(no.readonly = TRUE)))
```



```

[1] "adj"      "ann"      "ask"      "bg"       "bty"      "cex"
[7] "cex.axis" "cex.lab"  "cex.main" "cex.sub"  "col"      "col.axis"
[13] "col.lab"  "col.main" "col.sub"  "crt"      "err"      "family"
[19] "fg"       "fig"      "fin"      "font"     "font.axis" "font.lab"
[25] "font.main" "font.sub" "lab"      "las"      "lend"     "lheight"
[31] "ljoin"    "lmitre"   "lty"      "lwd"      "mai"      "mar"
[37] "mex"      "mfcoll"   "mfg"      "mfrow"    "mgp"      "mkh"
[43] "new"      "oma"      "omd"      "omi"      "pch"      "pin"
[49] "plt"      "ps"       "pty"      "smo"      "srt"      "tck"
[55] "tcl"      "usr"      "xaxp"     "xaxs"     "xaxt"     "xlog"
[61] "xpd"      "yaxp"     "yaxs"     "yaxt"     "ylbias"   "ylog"

```

Il y a également 5 paramètres graphiques non modifiables "cin" "cra" "csi" "cxy" "din"

```
> par("din")
```

```
[1] 7 7
```

### Commentaires

- ann : Par défaut les graphiques sont annotés (noms abscisse, ordonnée, titre), sauf si on demande explicitement qu'ils ne le soient pas.
- new : permet de superposer plusieurs figures.
- xlog ou ylog : échelle logarithmique.
- xpd : permet de déborder de la région utile ou même de la région de la figure.
- font, font.axis, font.lab, font.main et font.sub : contrôlent la police.
- las : orientation des étiquettes.
- pch : type de points.
- mfrow ou mfcoll : permet de subdiviser l'écran.
- mfg : après un découpage avec mfrow ou mfcoll, permet de choisir la sous-figure.
- family : contrôle la police utilisée, mais cela ne marche pas sur tous les périphériques.
- xaxs ou yaxs : permet de neutraliser l'extension de 4 % de xlim et ylim. exemple xaxs="i" ou yaxs="r" (i pour internal et r pour regular).
- adj : contrôle l'ajustement des chaînes de caractères.
- fin, pin : dimensions de la figure et de la zone utile.

```
> par("fin"); par("pin")
```

```
[1] 7 7
```

```
[1] 5.76 5.16
```

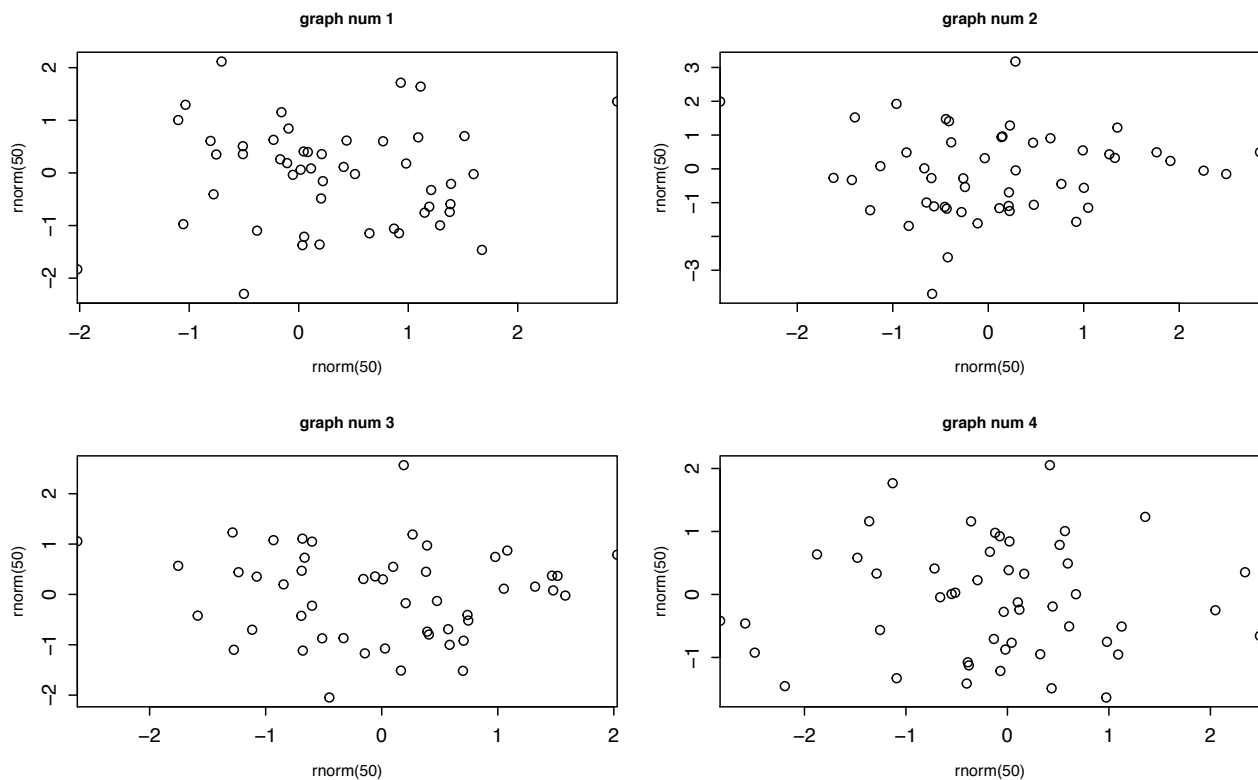
- mar : la taille de la région utile est contrôlée par la taille des marges exprimée en ligne. La valeur par défaut est  $\text{dec}(5, 4, 4, 2) + 0.1$ . Cela signifie que par exemple en bas, 5 lignes (une ligne pour les graduations (ticks), une ligne pour les étiquettes des graduations, une ligne vide, une ligne de légende, une ligne vide, plus une bordure vide d'un dixième de ligne tout autour).
- oma : permet d'ajouter des marges externes.

```

> par(oma = c(0, 0, 3, 0), mfrow = c(2, 2), mex = 0.7)
> for (i in 1:4) plot(rnorm(50), rnorm(50), main = paste("graph num", i), cex.lab=0.8, cex.main=0.8,
> mtext("Un titre commun aux quatre graphiques", side = 3, line = 1, outer = TRUE, cex = 1.2, font=4,
> mfrow = c(1, 1)

```

### Un titre commun aux quatre graphiques



### 3.1.3 Mise en forme

Option	Description
axes=TRUE ou FALSE	Si TRUE, les axes et le cadre sont tracés
type="n","p","l","b","h","s"...	Précise le type de graphe dessiné. type="n" supprime le graphe
col="blue", col.axis, col.main...	Précise la couleur du graphe, des axes, du titre...
bg="red"	Précise la couleur du fond
xlim=c(0,10), ylim=c(0,20)	Précise les limites des axes
xlab="axe x", ylab="axe y"	Précise les annotations des axes
main="titre"	Précise le titre du graphe
sub="sstitre"	Précise le sous-titre du graphe
bty="n","o"...	Contrôle comment le cadre est tracé. bty="n" supprime le cadre
cex=1.5, cex.axis, cex.main...	Contrôle la taille des caractères
font=1, font.axis...	Précise la police du texte
las="0"	Contrôle l'orientation des annotations des axes
lty="1"	Contrôle le type de lignes tracées
lwd=1.5	Contrôle la largeur des lignes
pch="+","o"...	Contrôle le type de symbole utilisé pour le tracé des points
ps=1.5	Contrôle la taille en points du texte et des symboles
tck, tcl	Précise la longueur des graduations sur les axes
mfc=c(3,2), mfrow=c(3,2)	Partitionne le graphe en 3 lignes et 2 colonnes. Les figures sont remplies colonnes par colonnes ou lignes par lignes

### 3.1.4 Graphes multiples

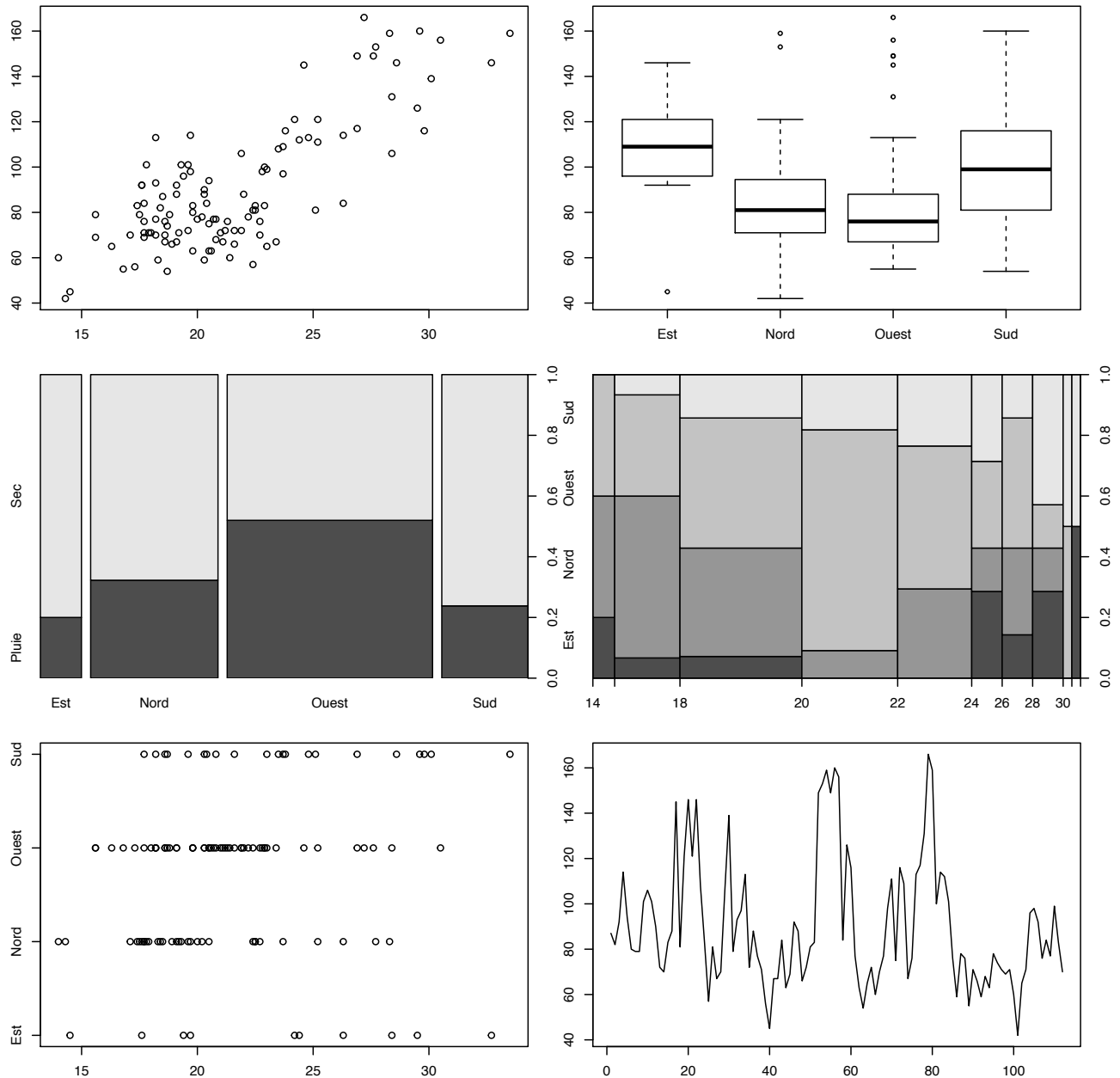
```
> xyplot(Sepal.Length + Sepal.Width ~ Petal.Length + Petal.Width | Species,
+       data = iris, scales = "free", layout = c(2, 2),
+       auto.key = list(x = .6, y = .7, corner = c(0, 0)))
> xyplot(Murder ~ Population | state.region, data = states,
```

```
+      groups = state.name,
+      panel = function(x, y, subscripts, groups) {
+          ltext(x = x, y = y, labels = groups[subscripts], cex=1,
+              fontfamily = "HersheySans")
+      })
```

### Paramètre mfrow

`par(mfrow = c(2,3))` : les graphes seront tracés sur 2 lignes et 3 colonnes par ligne.  
`par(mfcol = c(2,3))` : les graphes seront tracés sur 2 lignes et 3 colonnes par colonne.

```
> # ----- Importation des données
> tab <- read.table("../Data/ozone.txt", h=T)
> # ----- Mise en page
> par(mfrow = c(3, 2))
> par(mar = c(3, 2, 1, 2))
> # ----- Dessin
> plot(maxO3 ~T12, data= tab)
> plot(maxO3 ~vent, data= tab)
> plot(pluie ~vent, data= tab)
> plot(vent ~T12, data= tab)
> plot(tab$T12 , tab$vent, yaxt="n")
> axis(side=2, at=1:4, labels=levels(tab$vent))
> plot(tab$maxO3, type="l")
> par(mfrow = c(1,1))
```



Remarques :

La fonction `plot` adapte son comportement au type de données.

La fonction `axis` permet de renommer les libellés des graduations.

Ecrire d'un titre général à plusieurs graphiques

```
> op <- par(mfrow = c(2, 2), oma=c(0,0,3,0))
> for (i in 1:4) plot(runif(20), runif(20), main=paste("random plot (",i,")",sep=''))
> par(op)
> mtext("Titre général", side=3, line=1.5, font=2, cex=2, col='red')
```

Fonction `split.screen`

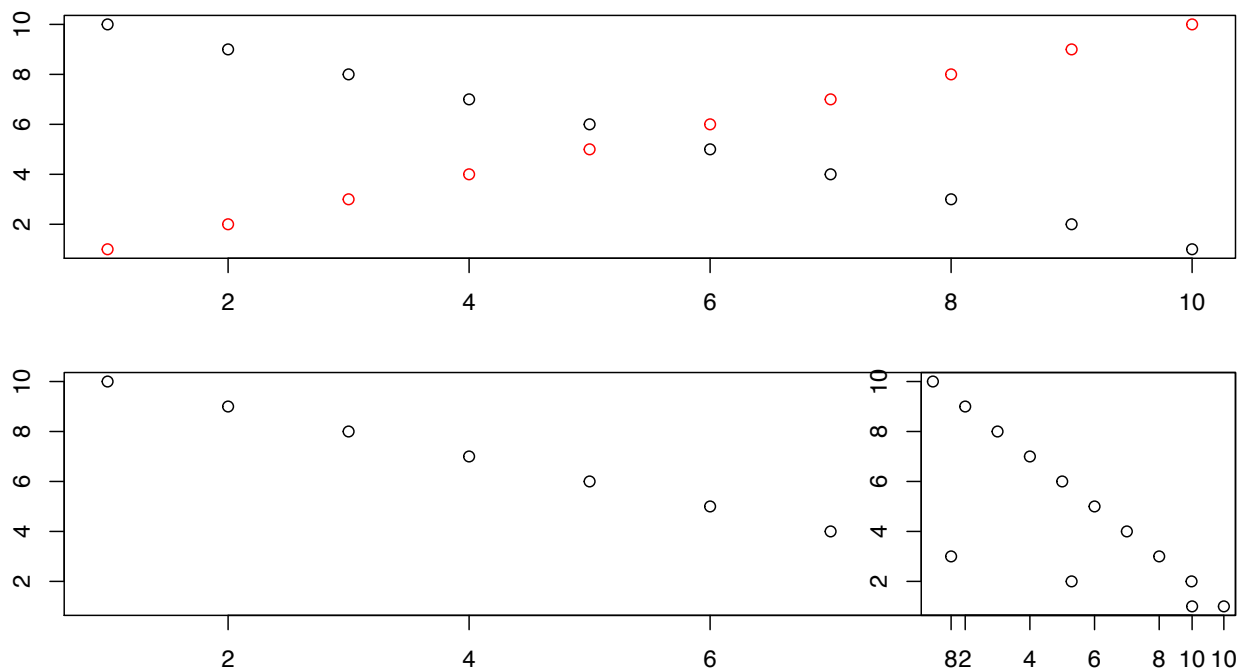
```
> par(mar = c(2, 2, 2, 2))
> split.screen(c(2,1)) # Sépare l'écran en 2 lignes
```

```
[1] 1 2
```

```
> split.screen(c(1,3), screen = 2) # Sépare l'écran n°2 en 3 parties
```

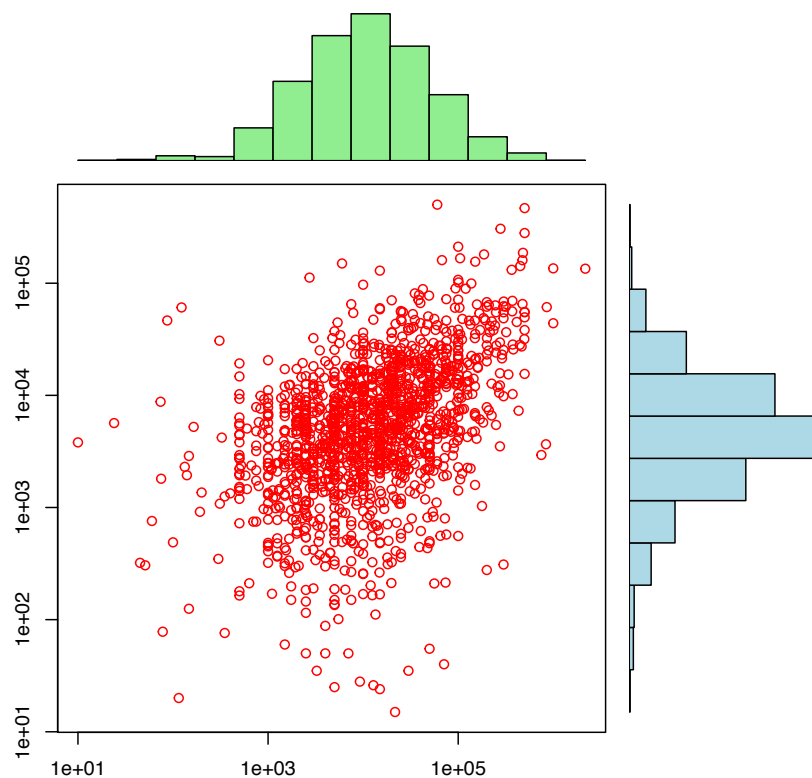
```
[1] 3 4 5
```

```
> screen(1) # Active l'écran 1
> plot(10:1)
> screen(5) # Active l'écran 5
> plot(10:1)
> erase.screen() # Efface le dernier écran actif
> screen(2) # Active l'écran 2
> plot(10:1)
> screen(1, FALSE) # Retour à l'écran 1, mais sans l'effacer
> plot(1:10, axes=FALSE, lty=2, ylab="", col="red") # surimpression
> close.screen(all = TRUE) # Sortie du mode split-screen
```



### Fonction layout

```
> library(evd)
> data(lossalae)
> x <- lossalae$Loss
> y <- lossalae$ALAE
> xhist <- hist(log(x), plot=FALSE)
> yhist <- hist(log(y), plot=FALSE)
> top <- max(c(xhist$counts, yhist$counts))
> nf <- layout(matrix(c(2,0,1,3),2,2, byrow=TRUE), c(3,1), c(1,3), TRUE)
> par(mar=c(3,3,1,1))
> plot(x, y, xlab="", ylab="", log="xy", col="red")
> par(mar=c(0,3,1,1))
> barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0,col="light green")
> par(mar=c(3,0,1,1))
> barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE,col="light blue")
```



```
> plot.new()
> plot(cars,type="p",col="red",lwd=2)
> par(new=TRUE)
> par(fig=c(0.1, 0.7, 0.55, 0.98))
> plot(cars,type="p",col="green",lwd=2)
```

La fonction `layout` partitionne le graphique actif en plusieurs parties sur lesquelles sont affichés les graphes successivement. Elle permet de faire des graphiques de taille différentes

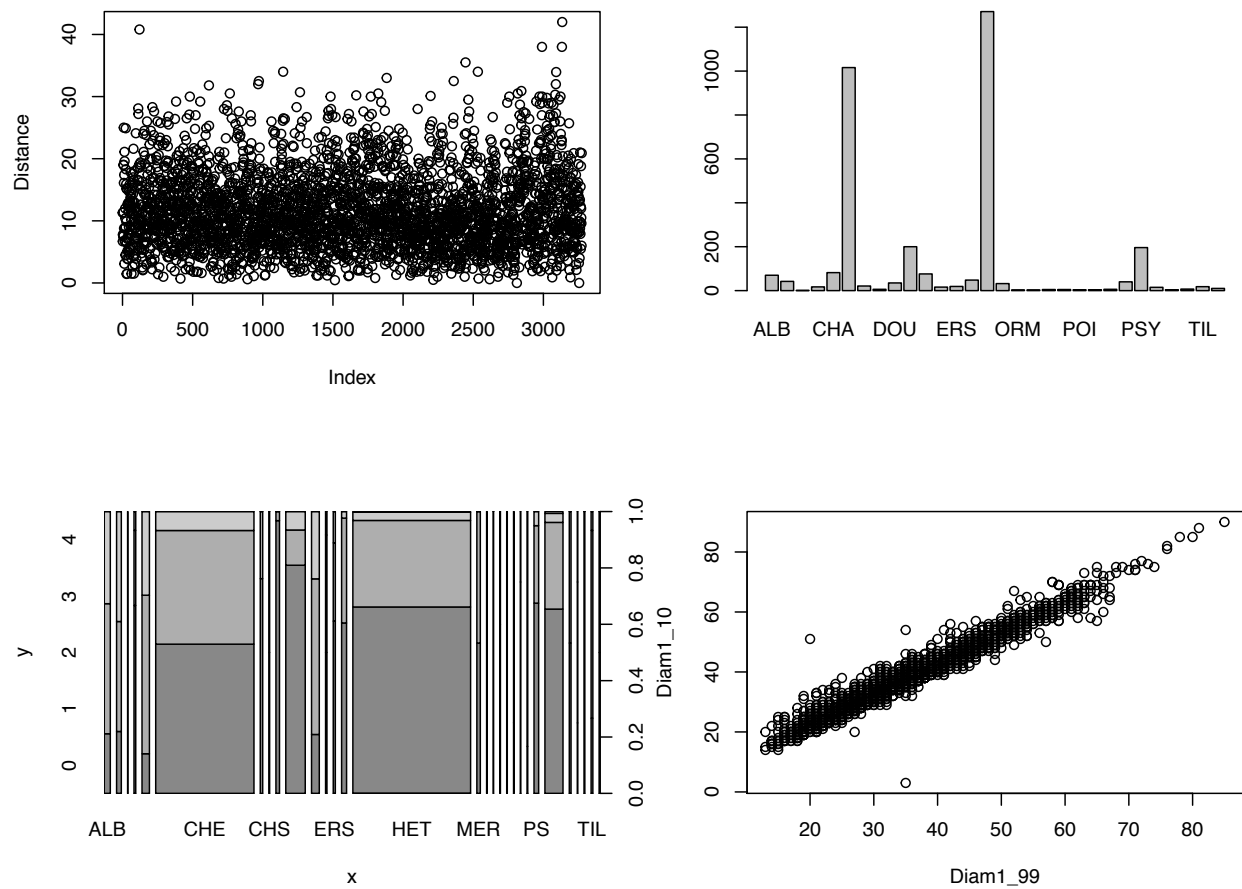
```
> mat <-matrix(c(1,1,2,3), nrow=2, ncol=2, byrow=TRUE)
> layout(mat)
> plot(1:10,10:1,pch=0)
> plot(rep(1,4),type="l")
> plot(c(2,3,-1,0),type="b")
```

## 3.2 Fonction plot

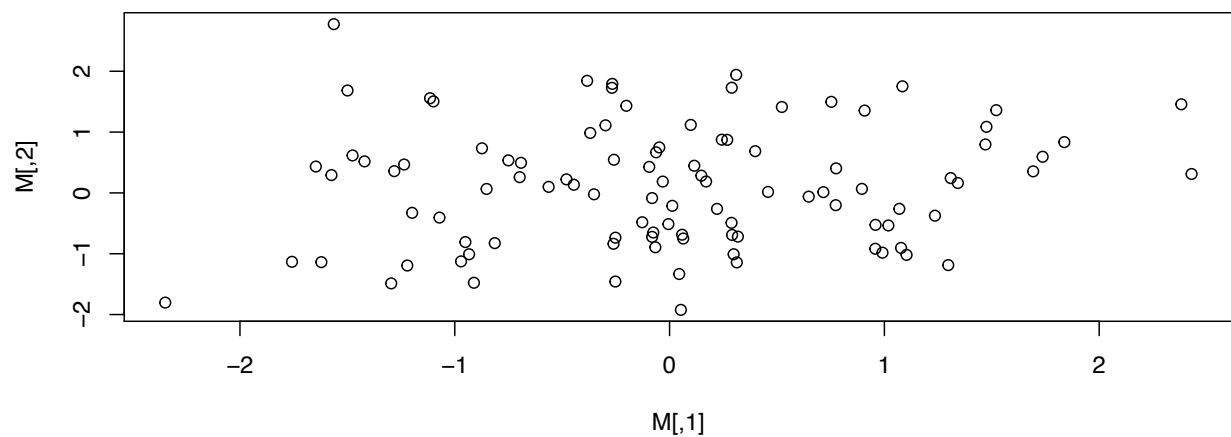
### 3.2.1 Quelques exemples

Elle s'adapte automatiquement aux différents types de données : quantitatives, qualitatives ou mixtes.

```
> attach(arbres)
> par(mfrow = c(2,2))
> plot(Distance) # représentation dans l'ordre des données
> plot(Essence) # histogramme du nombre d'individus par essence
> plot(Essence, factor(Statut_08)) # histogramme du nombre d'individus par essence et statut
> plot(Diam1_10 ~ Diam1_99) # nuage de points
> par(mfrow = c(1,1))
```

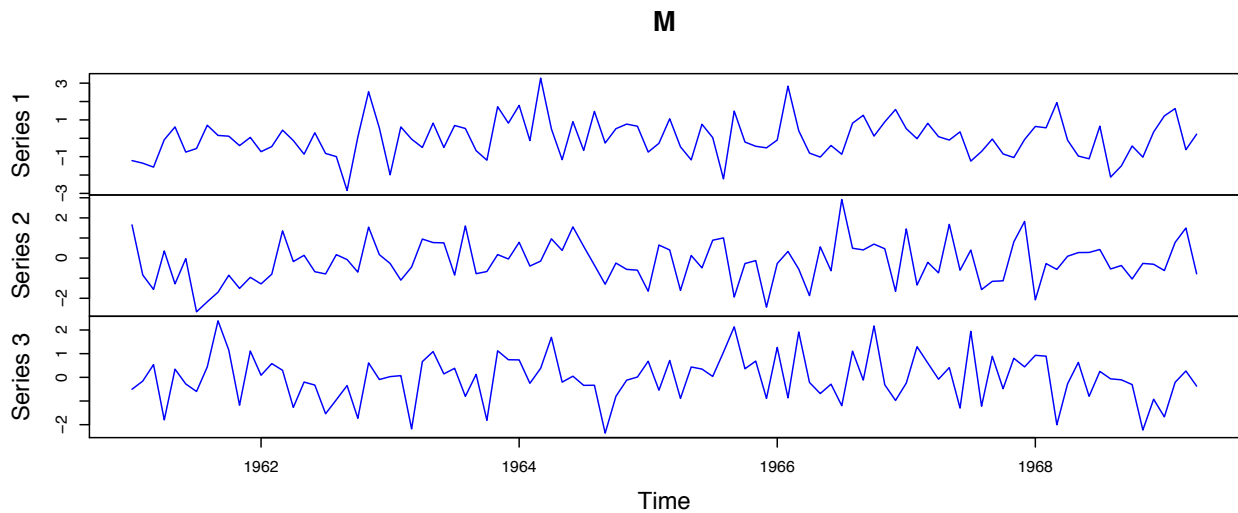


```
> M <- matrix(rnorm(300), 100, 3)
> plot(M)
```

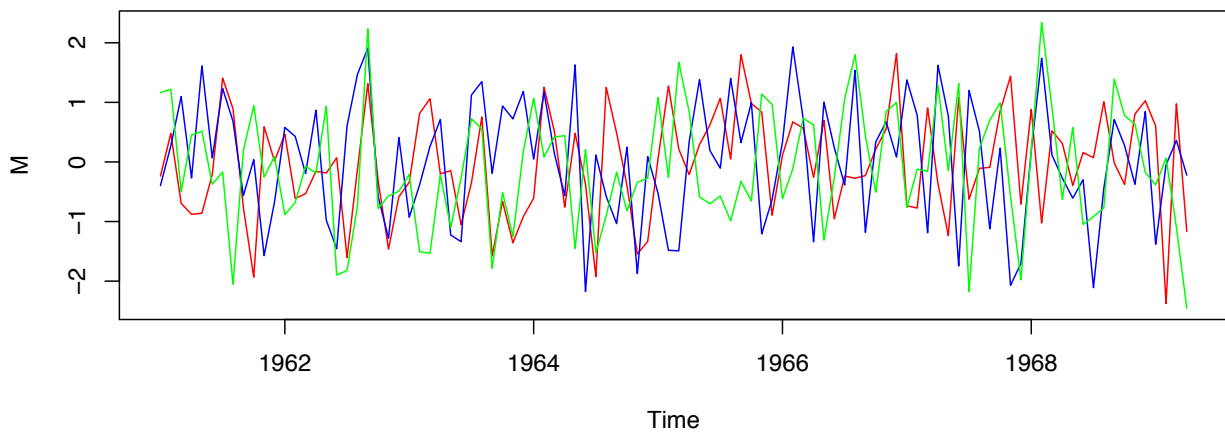


la fonction `ts` permet de créer des séries temporelles (time-series).

```
> M <- ts(matrix(rnorm(300), 100, 3), start=1961, frequency=12)
> plot(M, col="blue")
```



```
> M <- ts(matrix(rnorm(300), 100, 3), start=1961, frequency=12)
> plot(M, plot.type="single", col=c("red","blue","green"))
```



Le script ci-après décompose les différentes éléments constitutifs d'un graphique.

```
> plot.new() # Ouverture d'un nouveau graphique
> plot.window(xlim=c(0,1), ylim=c(5,10), asp=0.1) # Limites du graphique et ratio
> abline(a=6, b=3)
> axis(1)
> axis(2)
> title(main="The Overall Title")
> title(xlab="An x-axis label")
> title(ylab="A y-axis label")
> box()
```



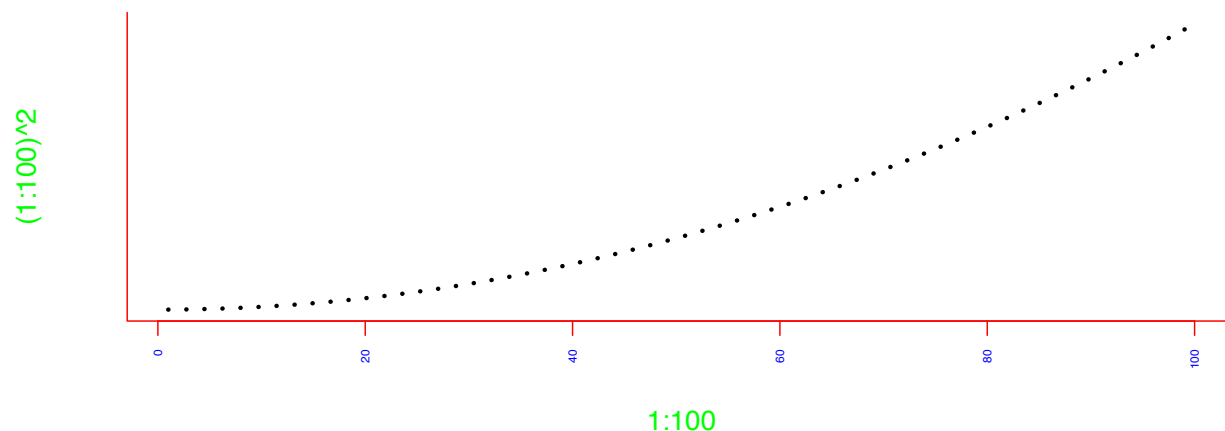
### 3.2.2 Les options

Une liste d'options est fournie dans l'aide en ligne (il suffit de taper `par()`). Parmi les options les plus utiles :

cex	taille des caractères à l'intérieur du graphique par rapport à la taille par défaut
cex.axis	taille des abscisses par rapport à la taille par défaut
bg	couleur du fond
pch	type de dessin. Exemple <code>pch=20</code> point plein. <code>pch = 1</code> = rond vide, <code>pch = *</code>
col	permet de jouer sur les couleurs et la transparence des couleurs
type	'p' pour points, 'l' pour lignes, 'b' pour both, 'c' for the lines part alone of "b", "o" for both 'overplotted', "h" for 'histogram' like (or 'high-density') vertical lines, "s" for stair steps, "S" for other steps, "n" for no plotting.
lty	line type. "blank", "solid", "dashed", "dotted", "dotdash", "longdash", ou "twodash"
xaxp	<code>xaxp = c(0, 50, 10)</code> numérote l'axe des x de 0 à 50 avec 10 ticks (idem avec <code>yaxp</code> ).
log	<code>log = "x"</code> logarithmique selon x ou bien <code>log = "xy"</code> pour avoir les deux axes logarithmiques.
bty	contrôle la forme du cadre : carré par défaut ("o"), en L ("l"), en U ("u"), en C ("c"), en 7 ("7") ou en crochet ("]")

Le code suivant trace le graphe avec des lignes (`type`), en pointillés (`lty`), d'épaisseur 3 (`lwd`), sans graduation de l'axe des y (`yaxt`), avec des axes en rouge (`fg`), seulement en bas et à gauche (`bty`), avec des étiquettes de graduation perpendiculaires à l'axe (`las`), un label d'axe 1.5 fois plus grand que normal (`cex.lab`), des étiquettes de graduation 2 fois plus grand que normal (`cex.axis`), une couleur de label d'axe verte (`col.lab`) et un couleur des graduations bleue (`col.axis`).

```
> plot(1:100, (1:100)^2, type = "l", lty = "dotted", lwd = 3, yaxt = "n", fg = "red", bty = "l",
+      las = 2, cex.lab = 1.2, cex.axis = 0.5, col.lab = "green", col.axis = "blue")
```



#### Modification du quadrillage

```
> plot(seq(1, 10), seq(11, 20), tck = 1) # dessine une grille.
> plot(seq(1, 10), seq(11, 20), tck = 0.02) # ticks intérieurs (en fraction de la zone de plot)
> plot(seq(1, 10), seq(11, 20), tck = -0.02) # ticks extérieurs (en fraction de la zone de plot)
```

#### Modification des marques































l'option `pch` permet de modifier les motifs. Ce peut être un entier de 1 à 25, ou bien n'importe quel caractère entre guillemets.

```
> px <- c(rep(0.5,5),rep(1.5,5),rep(2.5,5),rep(3.5,5),rep(4.5,5),rep(5.5,5))
> py <- rep(5:1,6)
> plot(px,py, type="n", xlim= c(0,6), ylim=c(0,5), xlab="", ylab="", axes=F, ann=FALSE)
```

```

> abline(v = 0:6, lty=2,col='gray')
> abline(h = 0:6, lty=2,col='gray')
> sigc <- paste(0:25,":",sep="")
> sigc <- c(sigc,"@:", "A:", ":", "+:")
> text(px-0.3, py-0.5,sigc,cex=1,col="black")
> dessin <- c(0:25,"@", "A", ":", "+")
> points(px[1:26], py[1:26]-0.5, pch=0:25, cex=2, col="blue", bg="red")
> points(px[27:30], py[27:30]-0.5, pch=c("@", "A:", ":", "+"),cex=2, col="blue")

```

0: 	5: 	10: 	15: 	20: 	25: 
1: 	6: 	11: 	16: 	21: 	@: 
2: 	7: 	12: 	17: 	22: 	A: 
3: 	8: 	13: 	18: 	23: 	*: 
4: 	9: 	14: 	19: 	24: 	+: 

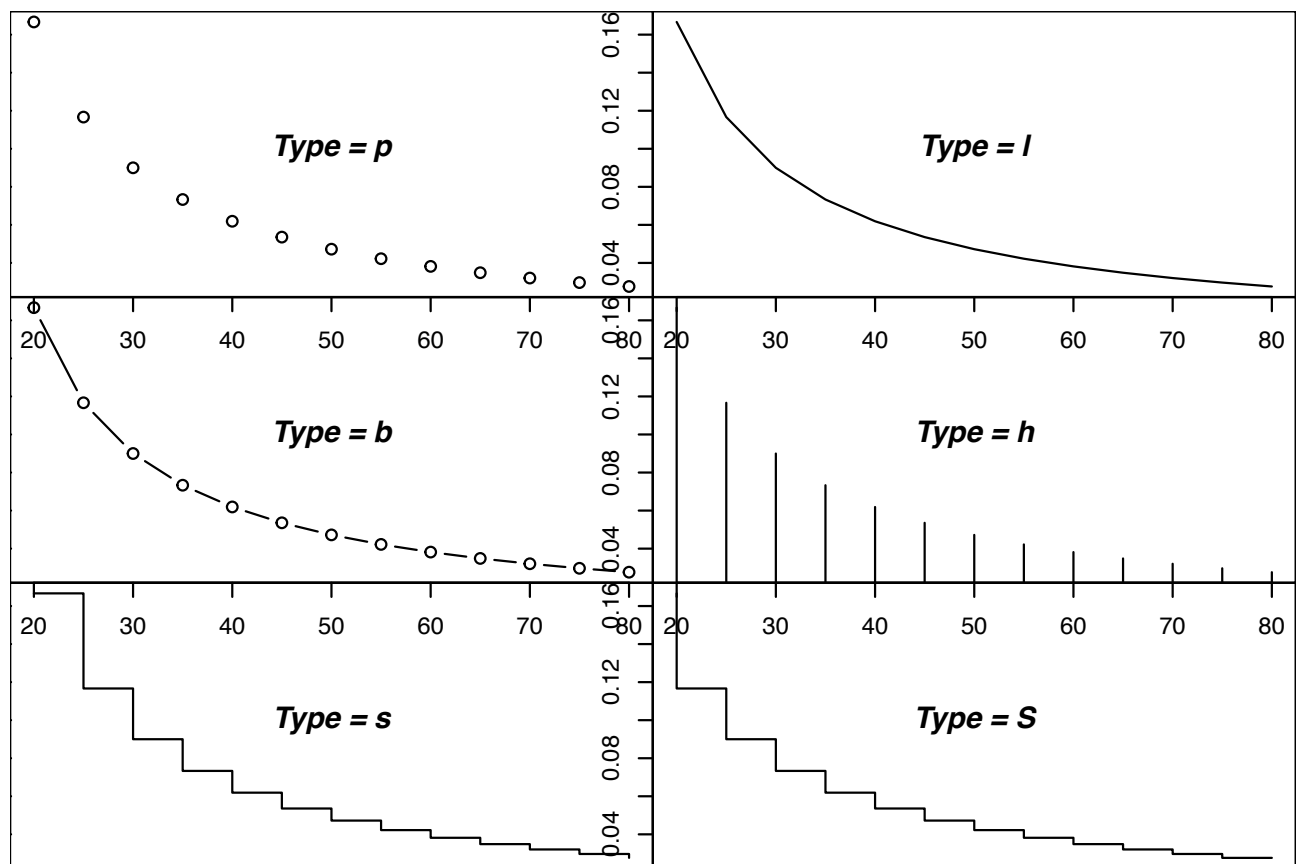
## Modification du type

Affichage des principaux types de graphiques

```

> # -----Accroissement relatif en volume
> diam <- seq(20,80,5)
> yschr <- (2*diam - 15)/(diam-5)/(diam-10)
> par(mfrow = c (3,2))
> par(mar = c(0, 0, 0, 0))
> plot(yschr ~ diam, type="p")
> text(50,0.1,"Type = p", font=4, cex=1.2)
> plot(yschr ~ diam, type="l")
> text(50,0.1,"Type = l", font=4, cex=1.2)
> plot(yschr ~ diam, type="b")
> text(50,0.1,"Type = b", font=4, cex=1.2)
> plot(yschr ~ diam, type="h")
> text(50,0.1,"Type = h", font=4, cex=1.2)
> plot(yschr ~ diam, type="s")
> text(50,0.1,"Type = s", font=4, cex=1.2)
> plot(yschr ~ diam, type="S")
> text(50,0.1,"Type = S", font=4, cex=1.2)
> par(mfrow = c (1,1))
> par(mar=c(5,4,4,4))

```



### Graphiques intercatifs

Les fonctions interactives (exemple `identify`) permettent aux opérateurs d'extraire ou d'apporter de l'information à un graphique.

```
> plot(Diam1_99, Diam1_10)
> identify(Diam1_99, Diam1_10, row.names(arbres), cex=0.8, plot=T)
```

Il suffit de cliquer sur un point du graphique pour que le numéro de ligne apparaisse. Un clic droit sur le graphique permet d'arrêter l'opération (sous MacOS appuyer sur la touche `esc`).  
Idem mais on fait apparaître l'essence.

```
> identify(Diam1_99, Diam1_10, Essence, , cex=0.8, col="red", plot=T)
```

## 3.3 Fonctions graphiques secondaires

La fonction `plot` est une fonction graphique de haut niveau. Comme tout graphique on peut ajouter des infos avec des fonctions graphiques de bas niveau comme par exemple la fonction `title` ou `points`. On peut aussi ajouter une courbe avec les instructions `line`, `curve`, `abline`, ...

### 3.3.1 Liste des fonctions secondaires

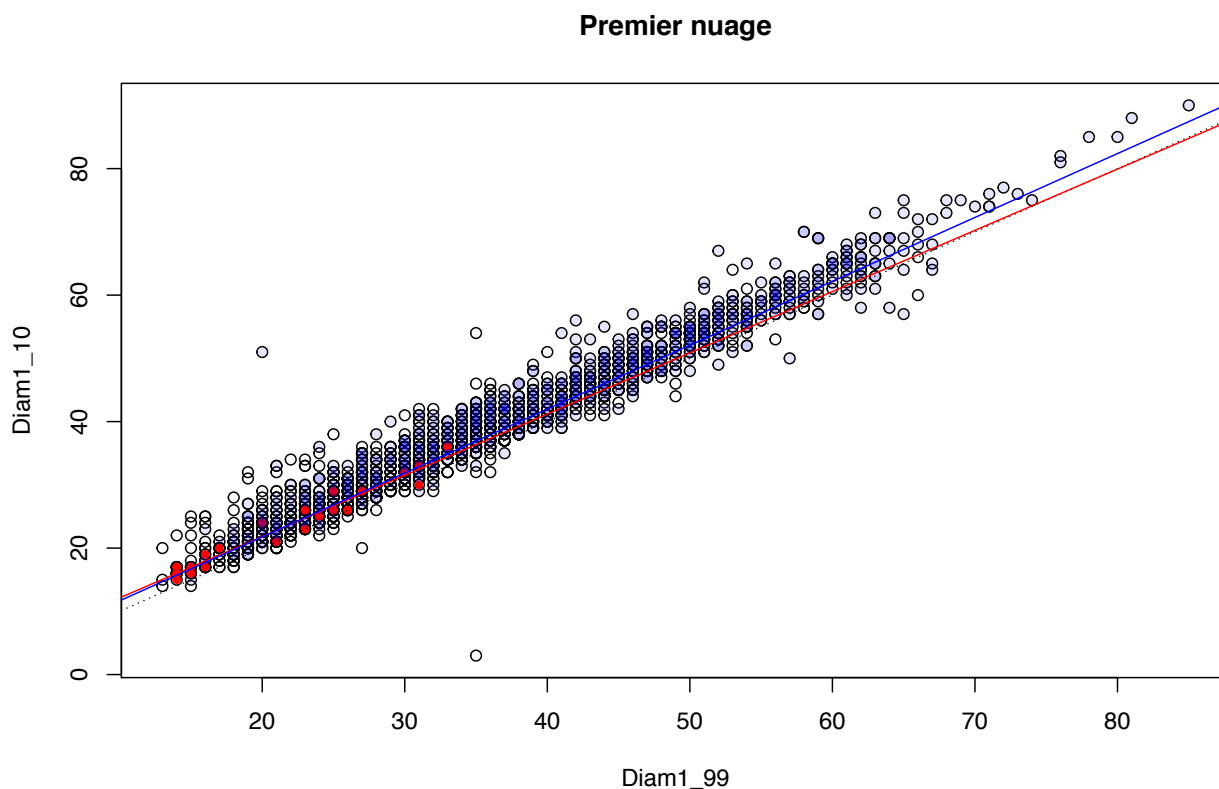
Les fonction suivantes sont dites secondaires car elles agissent sur un graphique existant.

Fonction	Description
<code>abline(a,b)</code>	trace la droite $y = bx+a$ . Pour tracer une droite horizontale (resp. verticale), on pourra utiliser <code>abline(h=)</code> (resp. <code>abline(v=)</code> ).
<code>points(x,y,...)</code>	ajoute des points.
<code>lines(x,y,...)</code>	relie de nouveaux points.
<code>text(x,y,labels,...)</code>	ajoute le texte défini par <code>labels</code> au point de coordonnées $(x,y)$ .
<code>mtext(text,size=3,line=0,...)</code>	ajoute le texte défini par <code>text</code> dans la marge (du bas si <code>side=1</code> , de gauche si <code>side=2</code> , du haut si <code>side=3</code> , de droite si <code>side=4</code> ) et à un nombre de lignes du cadre spécifiée par <code>line</code> .
<code>legend(x,y,legend,...)</code>	ajoute une légende au point de coordonnées $(x,y)$ .

### 3.3.2 Exemples de fonctions secondaires

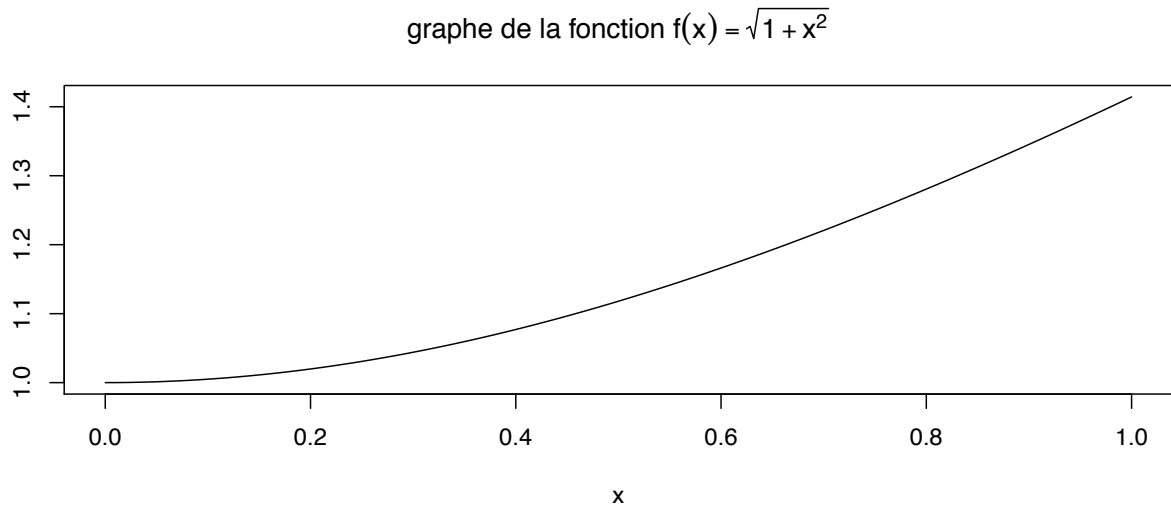
#### Droite de régression

```
> plot(Diam1_10 ~ Diam1_99)
> title("Premier nuage")
> points(Diam1_99[Essence=="ALT"],Diam1_10[Essence=="ALT"], col="red", pch=20)
> points(Diam1_99[Essence=="HET"],Diam1_10[Essence=="HET"], col=rgb(0, 0, 1, 0.1), pch=20)
> abline(0,1, lty=3)
> abline(coef = lm(Diam1_10[Essence=="ALT"] ~ Diam1_99[Essence=="ALT"])$coef, col="red")
> abline(coef = lm(Diam1_10[Essence=="CHE"] ~ Diam1_99[Essence=="CHE"])$coef, col="blue")
```



La fonction `expression` permet la notation mathématique.

```
> curve(sqrt(1+x^2), ylab="")
> title(main=expression("graphe de la fonction f"(x) == sqrt(1+x^2)))
```



Pour cet exemple, plutôt que la fonction `plot`, il est plus simple d'utiliser la fonction `curve`.

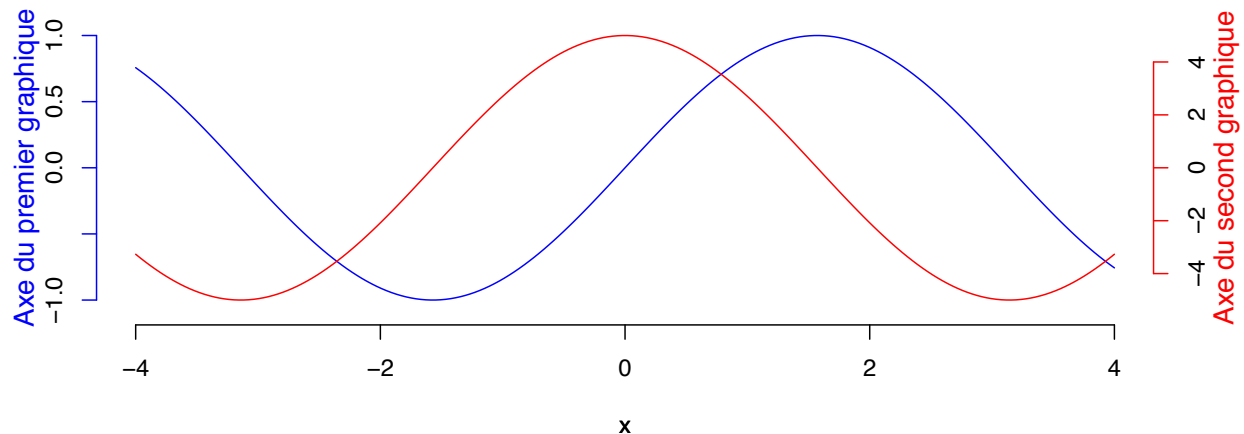
### Affichage légende

```
> x <- c(0:1000)*pi/100
> plot(x, cos(x), xlim=c(0,pi), ylim=c(-1,1), type="l", ylab="", font=3)
> lines(x, sin(x), lty=2)
> abline(-1, 1, lty=3)
> legend(1, -0.5, c("cos(x)", "sin(x)", "x-1"), lty = c(1, 2, 3), y.intersp=0.3)
```

### Graphiques multi-ordonnées

Le graphique suivant permet de dessiner un graphique avec 2 échelles indépendantes pour les ordonnées.

```
> par(mar=c(5,4,4,4))
> x=seq(-4,4,length=200)
> y1 = sin(x) ; y2 = 5*cos(x)
> ylim <- c(-1.1, 1.1)
> plot(x, y1, col="blue", type="l", ylim=ylim, axes=FALSE, ylab="")
> axis(1)
> axis(2, col="blue")
> ylim <- c(-5.5, 5.5)
> par(new=TRUE)
> plot(x, y2, col="red", type="l", ylim=ylim, axes=FALSE, ylab="")
> axis(4, col="red")
> mtext("Axe du premier graphique", 2, line=2, col="blue", cex=1.2)
> mtext("Axe du second graphique", 4, line=2, col="red", cex=1.2)
```



### 3.4 Les autres fonctions graphiques

#### 3.4.1 Les différents types de graphes

type	description
<code>plot(x)</code>	Trace le graphe des valeurs de x ordonnées sur l'axe des abscisses
<code>plot(x,y)</code>	Trace le graphe de y en fonction de x
<code>sunflowerplot(x,y)</code>	Idem mais les points superposés sont dessinés sous forme de fleurs dont le nombre de pétales correspond au nombre de points
<code>hist(x,freq=T)</code>	Trace un histogramme de x
<code>pie(x)</code>	Trace un graphe en camembert
<code>barplot(x)</code>	Trace un diagramme en barre (ou bande) des valeurs de x
<code>boxplot(x)</code>	Trace le graphe en boîtes et moustaches de x
<code>stripplot(x)</code>	Trace le graphe des valeurs de x sur une ligne
<code>interaction.plot(f1,f2,x,fun=mean)</code>	Trace le graphe des moyennes de x en fonction des valeurs des facteurs f1 (sur l'axe des abscisses) et f2 (plusieurs graphes)
<code>coplot()</code>	Trace le graphe bivarié de x et y pour chaque valeur de z (ou un petit intervalle de valeurs de z)
<code>matplot(x,y)</code>	Trace le graphe bivarié de la 1ère colonne de x contre la 1ère colonne de y, la 2ème colonne de x contre la 2ème colonne de y..
<code>pairs(x)</code>	Si x est une matrice ou un data.frame, tracer tous les graphes bivariés entre les colonnes de x
<code>plot.ts(x)</code>	Trace le graphe d'une série temporelle x en fonction du temps
<code>ts.plot(x)</code>	idem
<code>qqnorm(x)</code>	Trace les quantiles de x en fonction de ceux attendus d'une loi normale
<code>qqplot(x,y)</code>	Trace les quantiles de y en fonction de ceux de x
<code>contour(x,y,z)</code>	Trace des courbes de niveau
<code>filled.contour(x,y,z)</code>	Idem mais les aires entre les contours sont colorées, voir aussi <code>image(x,y,z)</code>
<code>persp(x,y,z)</code>	Idem mais en 3D, voir <code>demo(persp)</code>
<code>symbols(x,y,...)</code>	Dessiner aux coordonnées données par x et y des symboles (étoiles, cercles, boxplots...)
<code>arrows</code>	idem mais des flèches

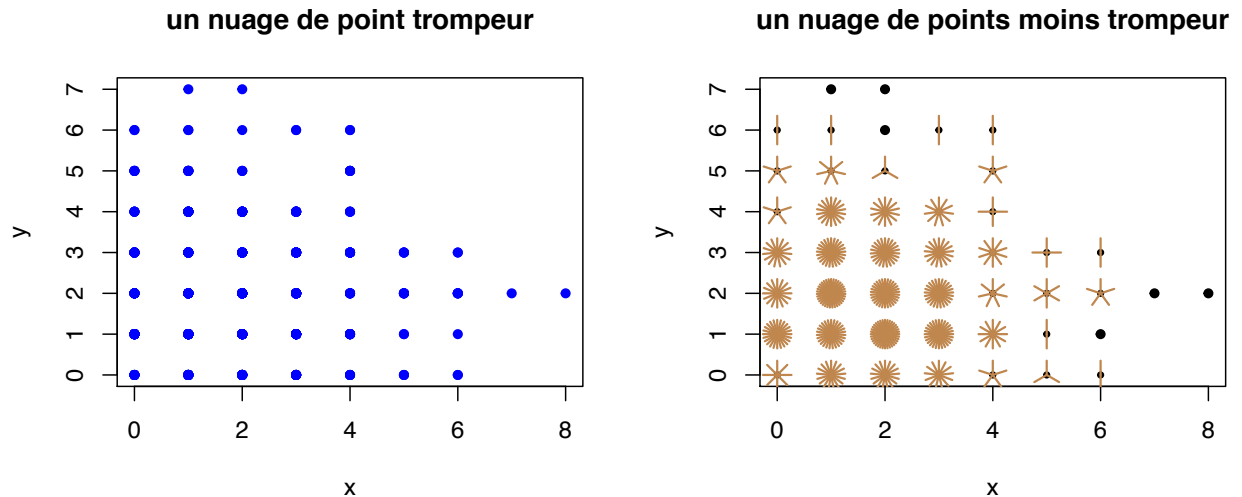
#### 3.4.2 Exemples

##### Gestion des points superposés

```

> n = 500
> x = rpois(n, lambda = 2)
> y = rpois(n, lambda = 2)
> par(mfrow = c(1,2))
> plot(x, y, pch = 19, cex=0.8, col = "blue", main = "un nuage de point trompeur")
> sunflowerplot(x, y, pch = 19, size=1/10, main = "un nuage de points moins trompeur")
> par(mfrow = c(1,1))

```

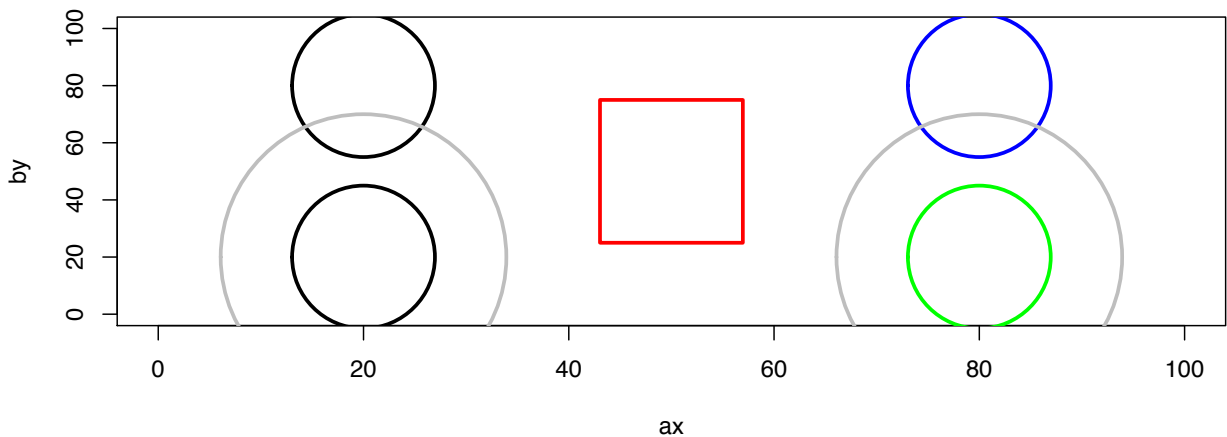


### Dessin

```

> # coordonnées du centre des cercles
> ax <- c(20,20,20,80,80,80)
> by <- c(80,20,20,80,20,20)
> s <- c(1,1,2,1,1,2) # taille
> couleur <- c("black","black","grey","blue","green","grey") # couleur
> #on trace des objets sur un graphique
> symbols(ax,by,circles=s, inches=1, fg=couleur, xlim=c(0,100), ylim=c(0,100), lwd=2.5)
> symbols(50,50,squares=10, inches=1, fg="red", add=TRUE, lwd=2.5)

```



### 3.4.3 Exemples : graphiques de distribution-comparaison

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

Le premier graphique trace le vecteur numérique  $x$  contre les *Normal order scores* (graphique de probabilité normale).

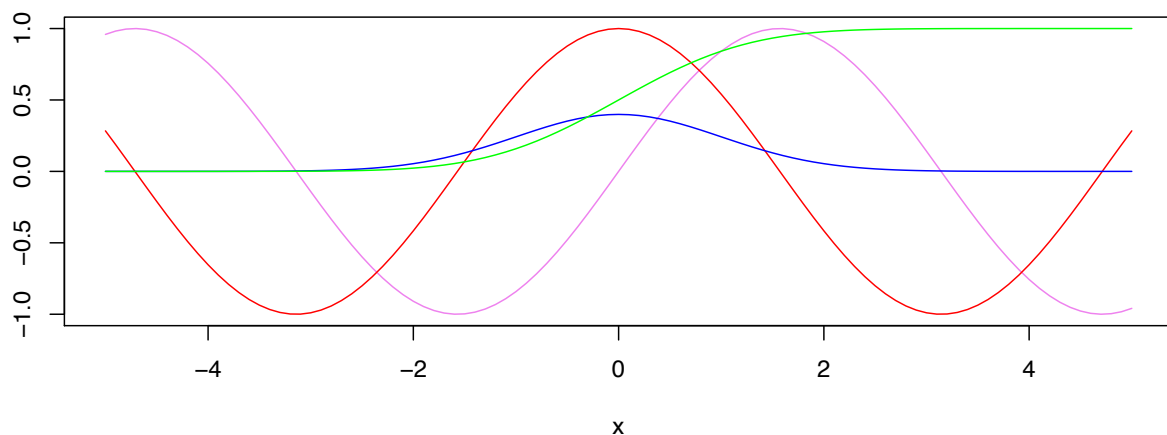
Le deuxième rajoute une ligne droite passant par la distribution et les quartiles de données.

Le troisième forme trace des quantiles de  $x$  contre ceux de  $y$  afin de comparer leurs distributions respectives.

### 3.4.4 Dessiner une fonction

On peut dessiner, avec la fonction `curve`, toutes les fonctions dont on connaît l'équation.

```
> x <- seq(-5,5,length=100)
> plot(x, x/5, type="n", ylab="")
> curve(sin(x), add = TRUE, col = "violet")
> curve(cos(x), add = TRUE, col = "red")
> curve(dnorm(x), add = TRUE, col = "blue")
> curve(pnorm(x), add = TRUE, col = "green")
```



### 3.4.5 Légende

La commande `legend` permet d'ajouter une légende au dessin.

```
> x <- seq(-6,6,length=200)
> y <- sin(x^2)
> z <- cos(x)
> plot(y~x, type='l', lwd=3, ylab='expression(sin(x^2))', xlab='angle', main="titre")
> abline(h=0,lty=3)
> abline(v=0,lty=3)
> lines(z~x, type='l', lwd=3, col='red')
> legend(-6,-1, yjust=0,c("Sinus^2", "Cosinus"), lwd=3, lty=1, col=c(par('fg'), 'red'),)
```

## 3.5 Périphériques graphiques

Par défaut, un graphique est affiché de manière temporaire dans une fenêtre. Le transfert dans un traitement de texte peut se faire par simple copier-coller. Il est également possible d'envoyer le dessin directement sous un



format PDF, JPEG ou autre.

La liste des périphériques disponibles s'obtient en tapant ?device.

Par exemple

Type	Description
pdf	Write PDF graphics commands to a file
postscript	Writes PostScript graphics commands to a file
quartz	The graphics device for the Mac OS X native Quartz 2d graphics system

Ces périphériques se ferment avec `dev.off()`.

```
> pdf("../graphics/nomFichier.pdf") # Ouvre un terminal pdf
> plot(1:10,1:10) # ...ou quelque chose de plus sophistiqué ...
> dev.off() # Ferme le terminal pdf et sauvegarde le fichier
```

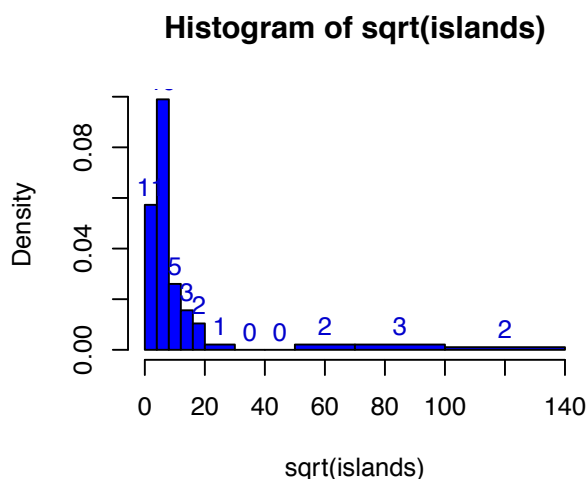
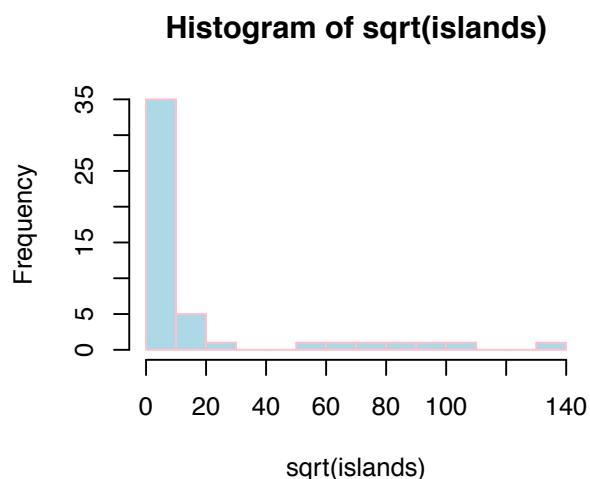
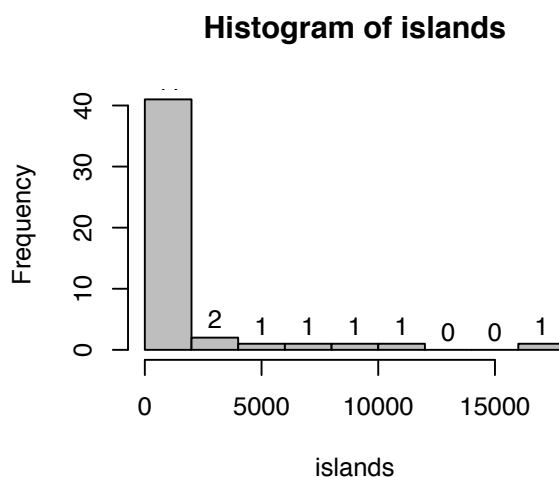
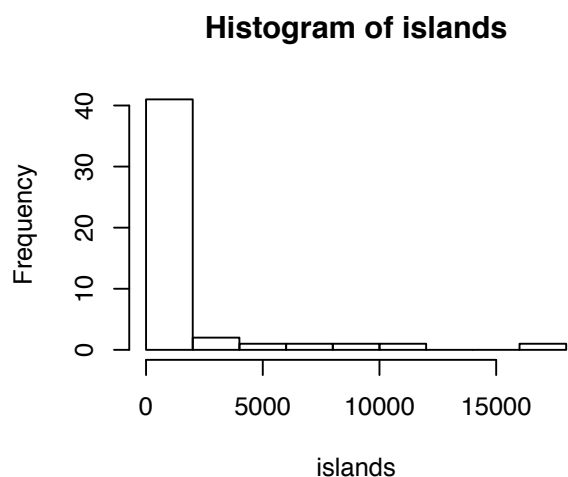
Attention : il faut que le dossier `graphics` existe !

## 3.6 Gestion couleur

## 3.7 Histogramme

Pour tracer un histogramme, la commande de base est la fonction `hist` qui contient un certain nombre de paramètres.

```
> par(mfrow=c(2, 2))
> hist(islands)
> hist(islands, col="gray", labels = TRUE)
> hist(sqrt(islands), breaks = 12, col="lightblue", border="pink")
> r <- hist(sqrt(islands), breaks = c(4*0:5, 10*3:5, 70, 100, 140), col='blue1')
> text(r$mids, r$density, r$counts, adj=c(.5, -.5), col='blue3')
> par(mfrow=c(1, 1))
```



### 3.8 Camembert

```
> ess <- rep(c("CHS","CHP"),5)
> Diam <- runif(10, 20, 80)
> Gha <- pi*Diam^2/40000
> tab <- data.frame(ess, Diam, Gha)
> pie(summary(as.factor(ess))) # correct
```

### 3.9 Fonction boxplot

```
> boxplot(mpg ~ cyl, data=mtcars, main="Car Milage Data",
+         xlab="Number of Cylinders", ylab="Miles Per Gallon")

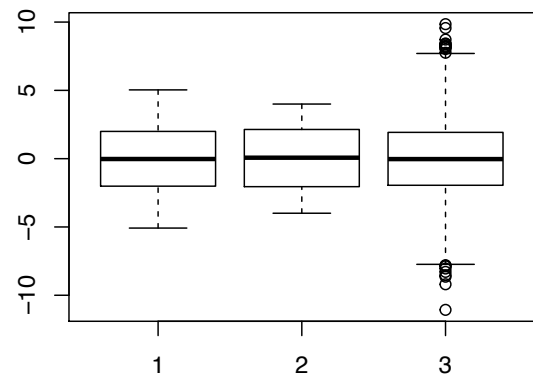
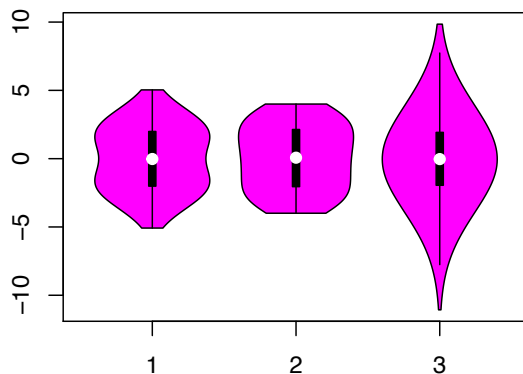
> boxplot(len~supp*dose, data=ToothGrowth, notch=TRUE,
+         col=(c("gold","darkgreen")),
+         main="Tooth Growth", xlab="Suppliment and Dose")
```

### 3.9.1 Fonction vioplot

```
> library(vioplot)
> x1 <- mtcars$mpg[mtcars$cyl==4]
> x2 <- mtcars$mpg[mtcars$cyl==6]
> x3 <- mtcars$mpg[mtcars$cyl==8]
> vioplot(x1, x2, x3, names=c("4 cyl", "6 cyl", "8 cyl"),
+       col="gold")
> title("Violin Plots of Miles Per Gallon")
```

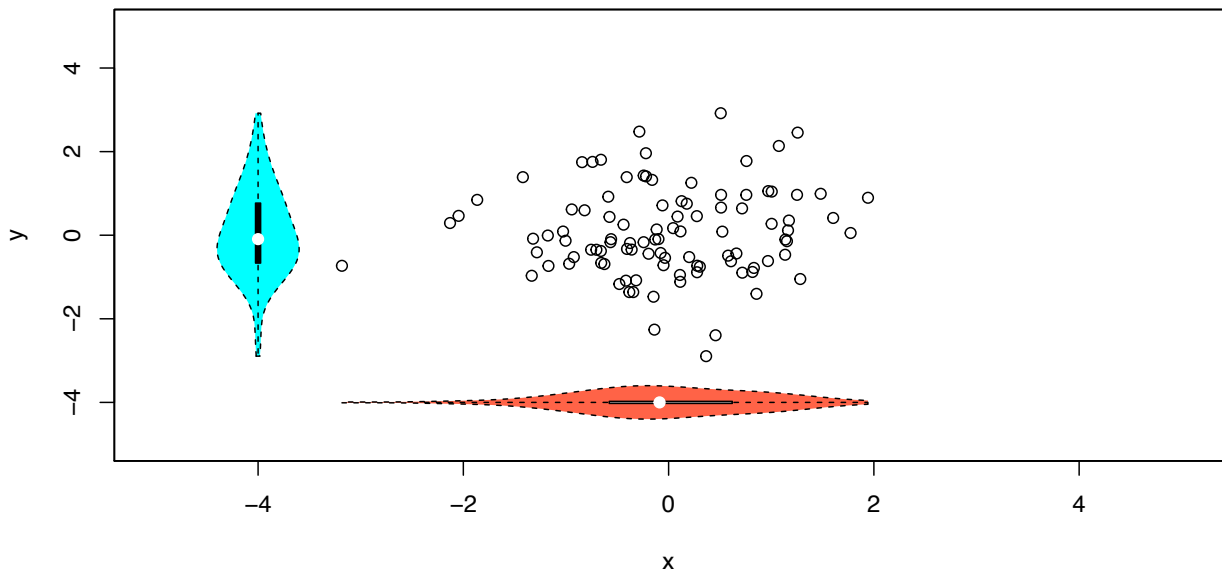
### 3.9.2 Comparaison vioplot/boxplot

```
> par(mfrow=c(1,2))
> bimodal <- c(rnorm(1000,-2,1),rnorm(1000,2,1))
> uniform <- runif(2000,-4,4)
> normal <- rnorm(2000,0,3)
> vioplot(bimodal,uniform,normal)
> boxplot(bimodal,uniform,normal)
> par(mfrow=c(1,1))
```



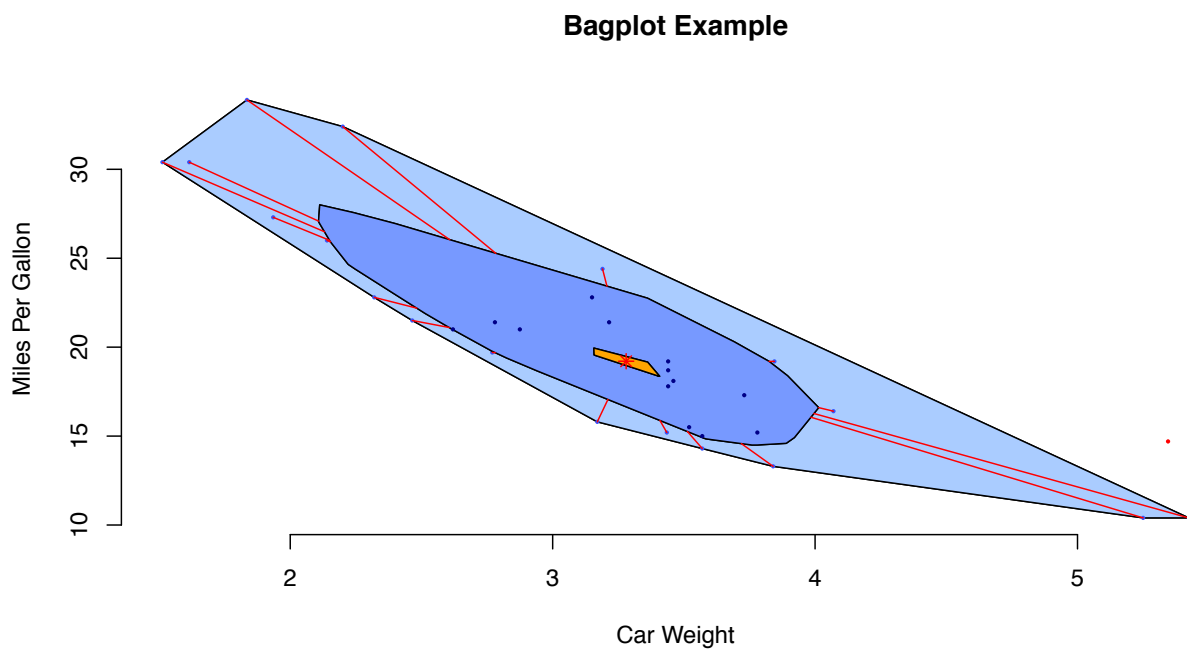
### 3.9.3 Apport ?

```
> x <- rnorm(100)
> y <- rnorm(100)
> plot(x, y, xlim=c(-5,5), ylim=c(-5,5))
> vioplot(x, col="tomato", horizontal=TRUE, at=-4, add=TRUE, lty=2, rectCol="gray")
> vioplot(y, col="cyan", horizontal=FALSE, at=-4, add=TRUE, lty=2)
```



### 3.9.4 bagplot : un boxplot en 2D

```
> library(aplpack)
> attach(mtcars)
> bagplot(wt,mpg, xlab="Car Weight", ylab="Miles Per Gallon",
+   main="Bagplot Example")
```



## 3.10 Les graphiques treillis

Ils permettent d'étudier les relations entre certaines variables, conditionnées par d'autres.

graph_ type	description	formula examples
barchart	bar chart	$x \sim A$ or $A \sim x$
bwplot	boxplot	$x \sim A$ or $A \sim x$
cloud	3D scatterplot	$z \sim x*y \mid A$
contourplot	3D contour plot	$z \sim x*y$
densityplot	kernal density plot	$x \mid A \sim B$
dotplot	dotplot	$x \mid A$
histogram	histogram	$x$
levelplot	3D level plot	$z \sim y*x$
parallel	parallel coordinates plot	data frame
spiom	scatterplot matrix	data frame
striplot	strip plots	$A \sim x$ or $x \sim A$
xyplot	scatterplot	$y \sim x \mid A$
wireframe	3D wireframe graph	$z \sim y*x$

```
> library(lattice)
> attach(mtcars)
```

The following object(s) are masked from 'mtcars (position 4)':

```
am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt
```

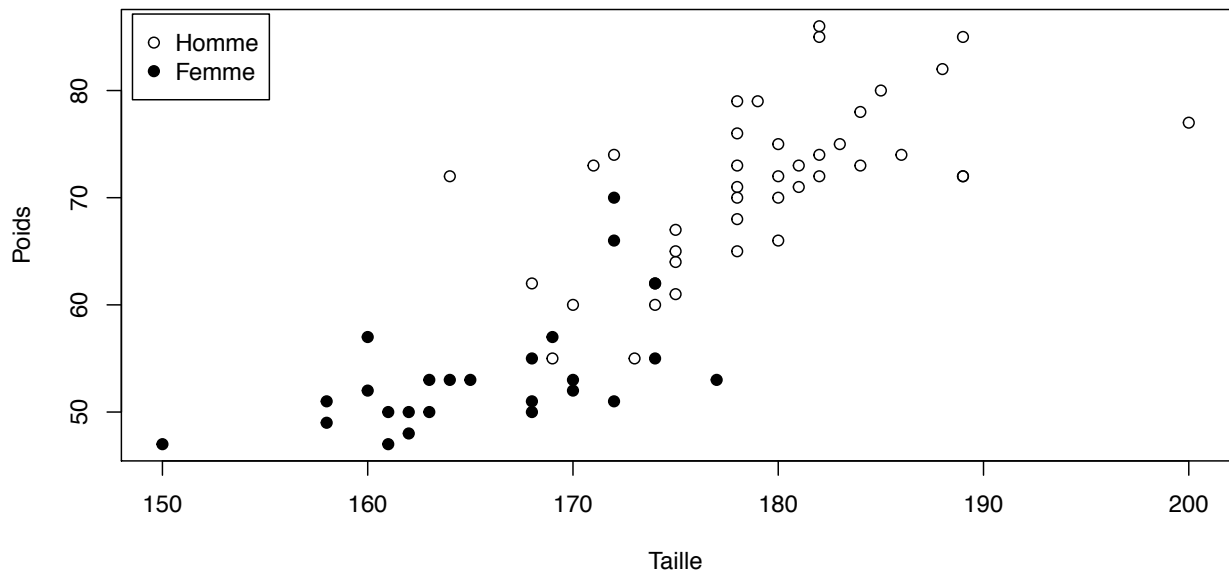
```
> densityplot(~mpg, main="Density Plot", xlab="Miles per Gallon")
> cyl.f <- factor(cyl, levels=c(4,6,8), labels=c("4cyl","6cyl","8cyl"))
> densityplot(~mpg|cyl.f, main="Density Plot by Number of Cylinders",
+             xlab="Miles per Gallon")
> densityplot(~mpg|cyl.f, main="Density Plot by Numer of Cylinders",
+             xlab="Miles per Gallon", layout=c(1,3))
```

## 3.11 Exemples

### 3.11.1 Nuages de points

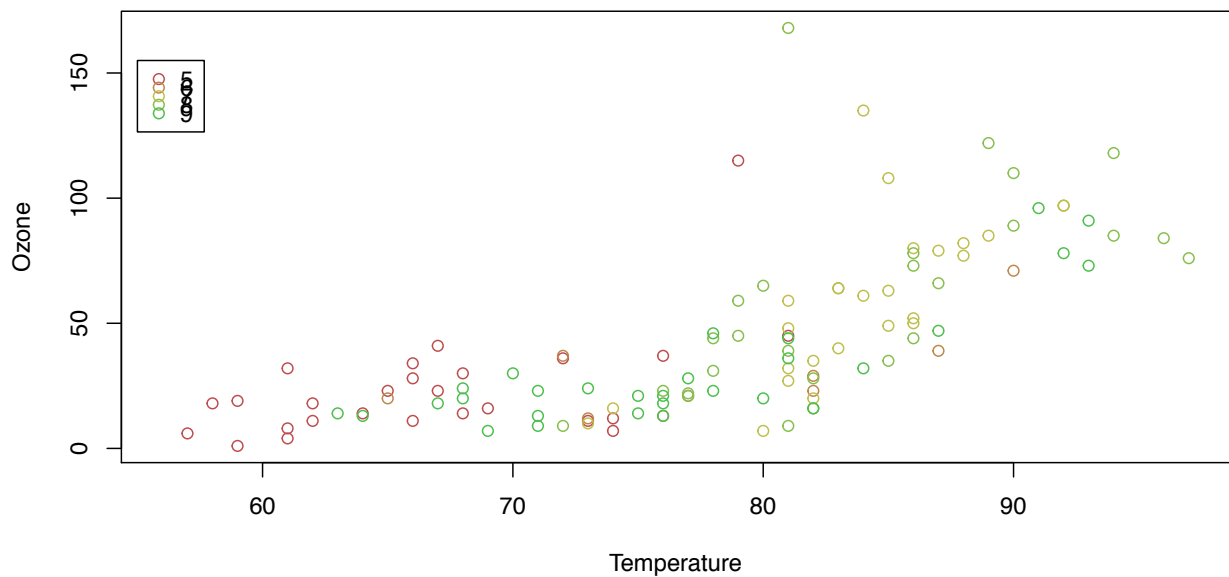
Nuage 2 populations

```
> tab <- read.table("http://pbil.univ-lyon1.fr/R/donnees/t3var.txt", h = TRUE)
> plot(poi ~ tai, data=tab, pch = ifelse(sexe == "h", 1, 19), xlab = "Taille", ylab = "Poids")
> legend("topleft", inset = 0.01, c("Homme", "Femme"), pch = c(1, 19))
```



### Nuage plusieurs populations

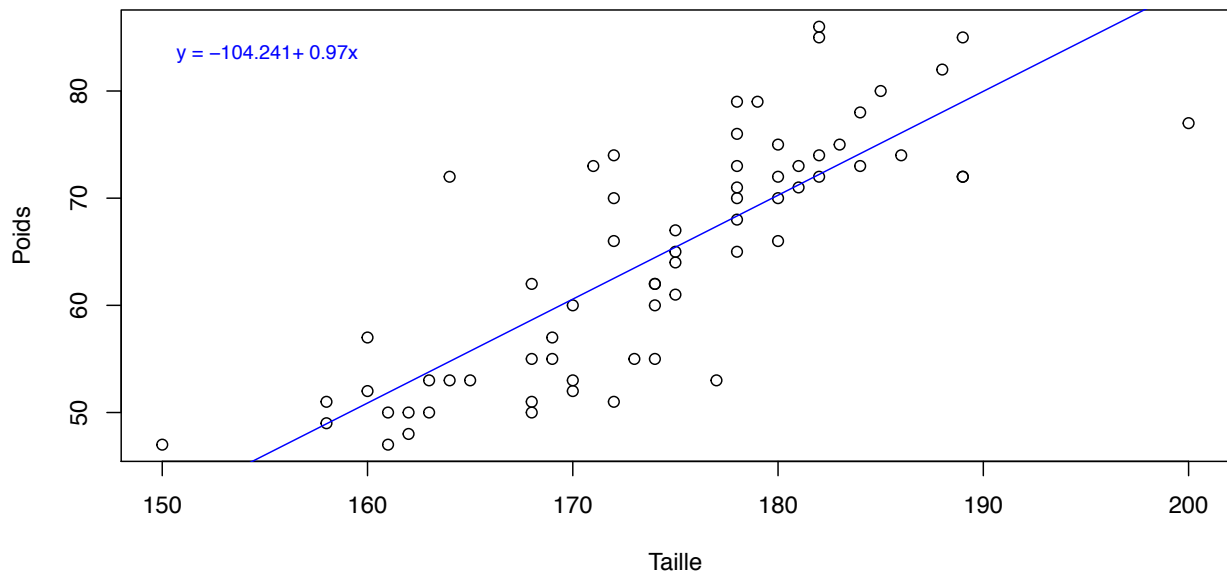
```
> plot(Ozone ~ Temp, data=airquality, xlab="Temperature", ylab="Ozone", type="n")
> month <- sort(unique(airquality$Month))
> for(i in month)
+ points(airquality$Temp[airquality$Month==i], airquality$Ozone[airquality$Month==i], col=(i-4))
> legend(55, 155, month, col=(month-4), pch=1, y.intersp=0.3)
```



La librairie ggplot permet de faire ce graphique de manière plus rapide.

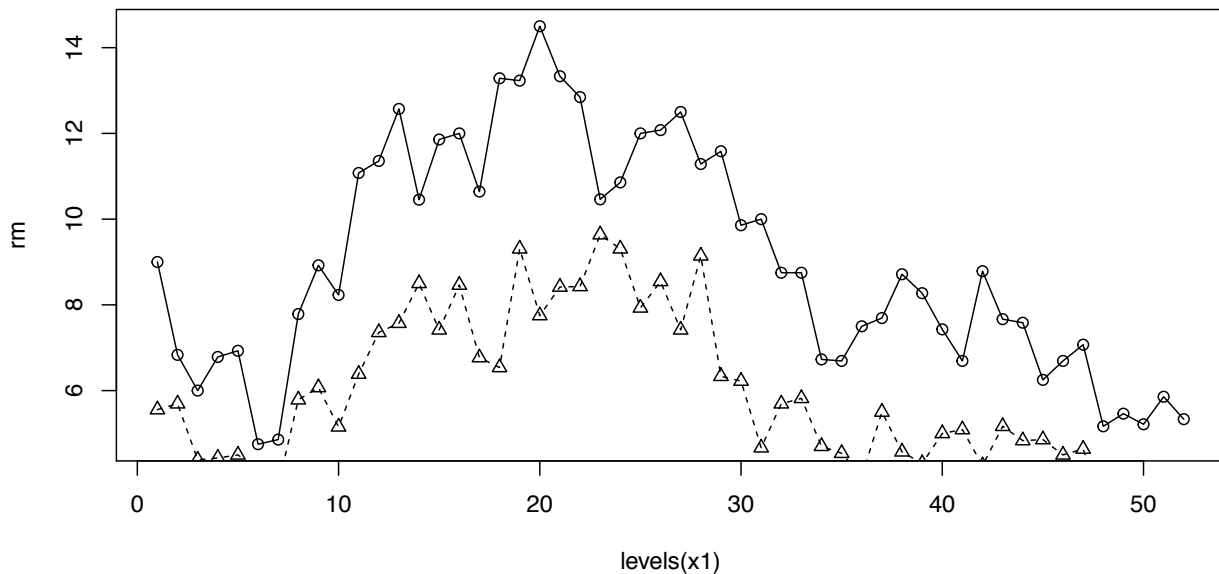
### regression

```
> tab <- read.table("http://pbil.univ-lyon1.fr/R/donnees/t3var.txt", h = TRUE)
> plot(poi ~ tai, data=tab, xlab = "Taille", ylab = "Poids")
> fit <- lm(poi ~ tai, tab)
> abline(fit, col="blue")
> text(min(tab$tai), max(tab$poi)*0.97, pos=4, cex=0.8, col="blue",
+      paste("y = ",round(fit$coefficients[1],3),"+ ", round(fit$coefficients[2],3), "x", sep=""))
```



### 3.11.2 Graphique 2 séries

```
> # -----Import des données
> tab <- read.table("http://pbil.univ-lyon1.fr/R/donnees/ecrin.txt", h = TRUE)
> y <- tab[,4]
> x1 <- factor(tab[,2])
> x2 <- factor(tab[,3])
> levels(x2) <- c("Ma", "So")
> # ----- Calcul des moyennes par semaines et pour matin et soir
> ym <- tapply(y, list(x1, x2), mean)
> rm <- ym[, 1]
> rs <- ym[, 2]
> # ----- Dessin
> plot(levels(x1), rm, pch = 1)
> points(levels(x1), rs, pch = 2)
> lines(levels(x1), rm, lty = 1)
> lines(levels(x1), rs, lty = 2)
```



### 3.11.3 Eclater nuage par rapport à une variable z

```
> MonCoplot <-function(n,a,b) {
+ # -----Importation des données
+ tab <- read.table("http://pbil.univ-lyon1.fr/R/donnees/ecrin.txt", h = TRUE)
+ # ----- Eclater la variable y par rapport à la colonne n°b
+ y.list <- split(tab,tab[,b])
+ # ----- Dessin
+ nuage <- function(x) {
+   plot(x[,a], x[,n], ylim = c(0, 18), xlim = c(0, 53), xlab = "", ylab = "", type = "n")
+   grid(col = grey(0.7), lty = 3)
+   points(x[,a], x[,n], pch = 20,cex = 1, col="blue")
+   lines (lowess (x[,a], x[,n],f=0.5), col="red")
+   text(0, 16, unique(x[,b]), cex = 1.5, pos = 4)
+ }
+ par(mfrow = c(4, 4))
+ par(mar = c(3, 2, 1, 1))
+ lapply(y.list, nuage)
+ }
```



# Chapitre 4

## Package ggplot2

### 4.1 Introduction

Développé par Wilkinson en 2007, le package ggplot2 adapte à R le principe de la grammaire de Wilkinson (2005) par l'utilisation de layers (calques) qui peuvent s'utiliser comme des objets (assignation possible). Ce package dépend principalement des packages reshape et plyr. Il nécessite la librairie ggplot2.

```
> library(ggplot2)
```

Intérêt principal de ggplot2 par rapport au module de base (plot) : réduction de la longueur des codes dès que l'on veut dépasser le simple graphique, que l'on veut par exemple faire des graphiques par sous-populations ou bien tout simplement rajouter une légende.

Elle possède deux fonctions graphiques de base, qplot (pour quick plot) et ggplot dont la syntaxe générale est respectivement

```
qplot(x, y, data=data)
ggplot(data, aes(x, y)) + layers.
```

Ces deux fonctions qplot ou ggplot conduisent aux mêmes résultats. Voici par exemple la même figure générée par chacune des deux fonctions.

```
> qplot(clarity, data=diamonds, fill=cut, geom="bar")
> ggplot(diamonds, aes(clarity, fill=cut)) + geom_bar()
```

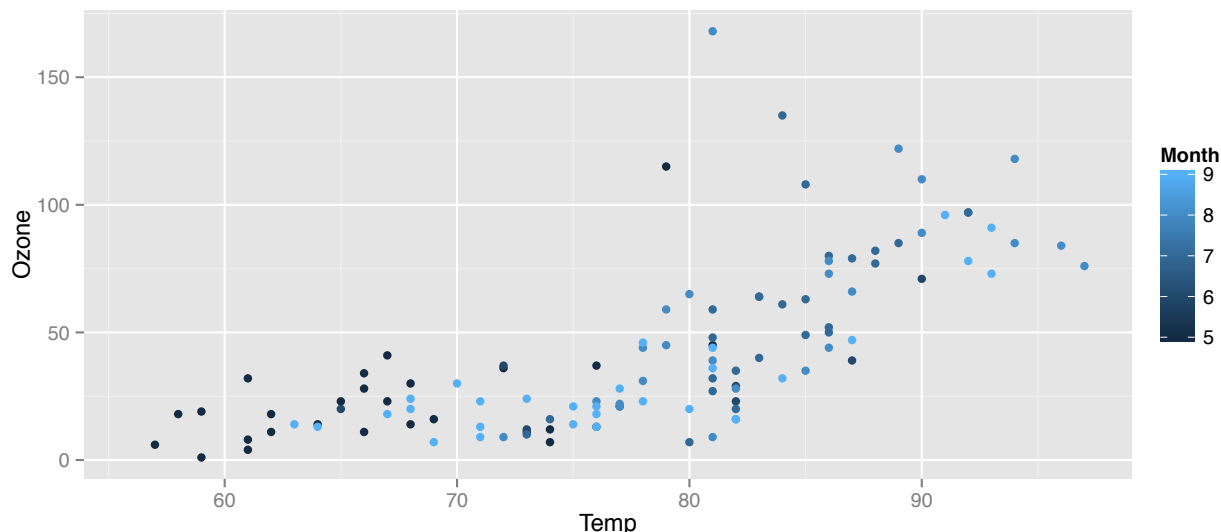
Les principaux layers sont les suivants :

data	données brutes
mapping	projection graphique
geom	objets géométriques (points, lignes, polygones, etc.)
stat	transformation statistique (histogramme, modèle, etc.)
scale	espace esthétique (couleurs, formes, tailles, axes, légendes)
coord	système de coordonnées (axes, grilles)
facet	subdivision (syn. lattice, trellis)

### 4.2 qplot()

Rapide, qplot fournit des graphiques plus simplement que la fonction plot du package de bases. A titre d'exemple, la ligne de code ci-après remplace celles du paragraphe 3.11.1.

```
> qplot(Temp, Ozone, data=airquality, colour=Month)
```



Ci-dessous, quelques exemples à tester.

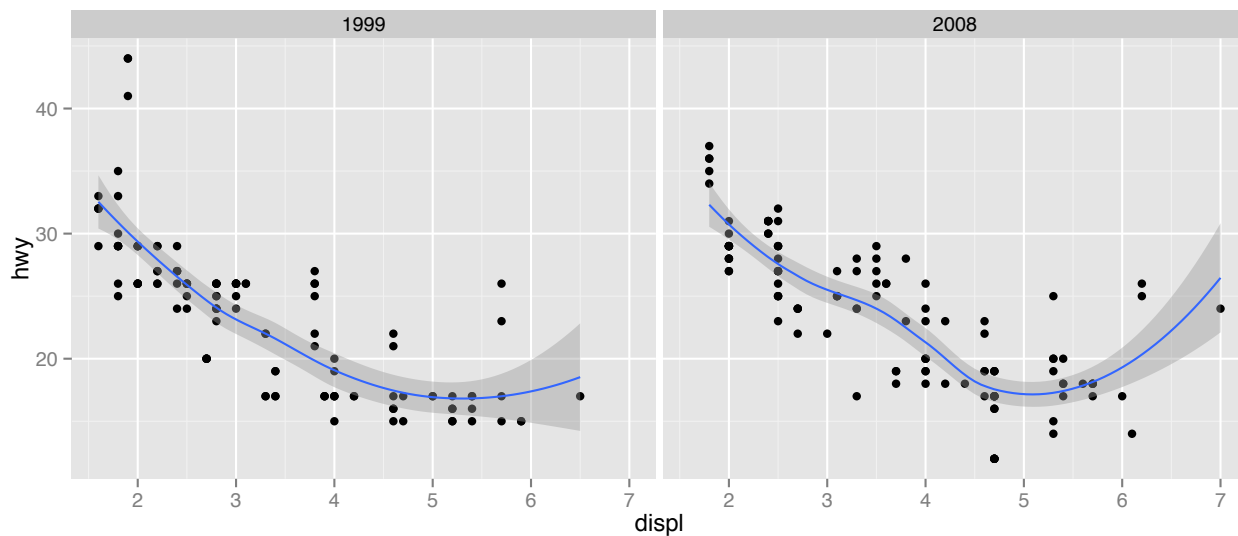
```
> qplot(cut, data = diamonds)
> qplot(cut, data = diamonds, weight = carat)
> qplot(cut, data = diamonds, weight = price)
> qplot(mpg, wt, data = mtcars, color=cyl)
> qplot(mpg, wt, data = mtcars, color=factor(cyl))
> qplot(mpg, wt, data=mtcars, size=cyl, color=factor(am))
> qplot(mpg, wt, data=mtcars, size=factor(cyl), color=factor(am))
> qplot(mpg, wt, data=mtcars, facets= ~ am) # le symbole ~ est nécessaire
> qplot(mpg, wt, data=mtcars, facets= vs ~ am)
> qplot(factor(cyl), data = mtcars) # Si seulement x = histogramme
> qplot(factor(cyl), data = mtcars, geom="bar") # Idem
> qplot(y = mpg, data = mtcars) # Si seulement y = nuage de point selon ordre d'apparition
> qplot(y = mpg, data = mtcars) + geom_point() # Idem
> qplot(factor(cyl), wt, data = mtcars)
> qplot(factor(cyl), wt, data = mtcars, geom="boxplot")
> qplot(factor(cyl), wt, data = mtcars, geom=c("boxplot", "jitter"))
> qplot(factor(cyl), wt, data = mtcars, geom=c("jitter", "boxplot"))
> qplot(mpg, wt, data=mtcars, facets= ~ am) + geom_smooth(method = "lm")
> qplot(mpg, wt, data = mtcars, color=cyl) + geom_smooth(method = "lm")
> qplot(factor(cyl), data=mtcars, geom="bar", fill=factor(cyl))
> qplot(factor(cyl), data=mtcars, geom="bar", fill=factor(gear)) # Plus intéressant
> qplot(reorder(class, hwy), hwy, data = mpg)
> qplot(table, data = diamonds, binwidth = 0.1) + xlim(50, 70) + ylim(0, 50)
> qplot(table, price, data = diamonds, geom = "boxplot", group=round(table))
```

Remarques : L'option "geom" indique comment représenter les données. Il existe de nombreuses options<sup>1</sup>. Par exemple l'option facets permet d'écarter le graphique en sous-populations. Sa syntaxe est la suivante :

- facets = facteur + tilde produira un graphique multiple en colonnes
- facets = tilde + facteur produira un graphique multiple en lignes

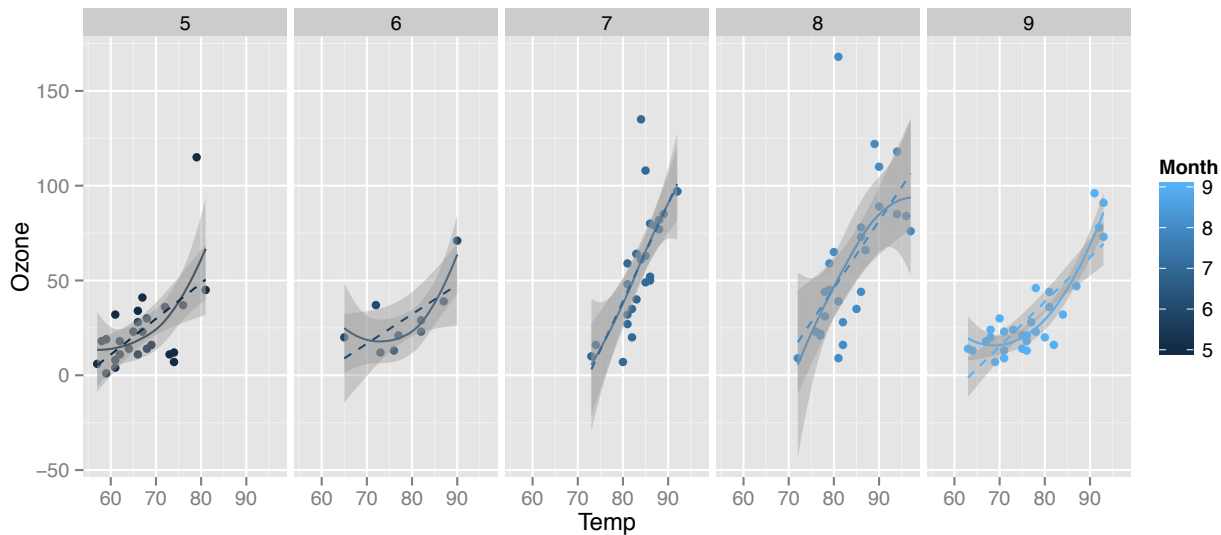
```
> qplot(displ, hwy, data = mpg, geom = c("point", "smooth"), facets = . ~ year)
```

1. <http://docs.ggplot2.org/current/>



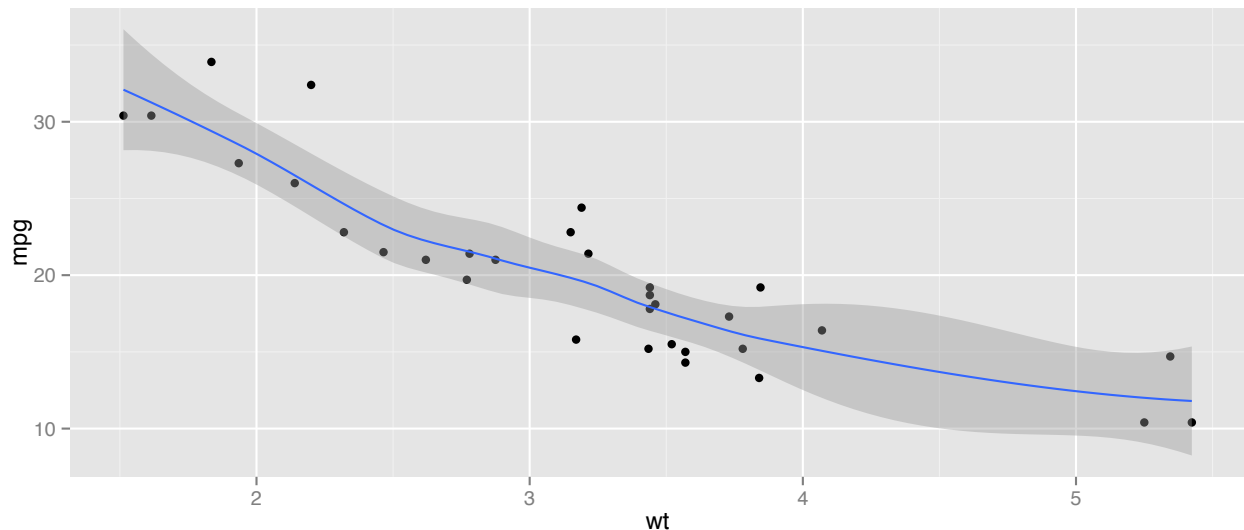
Voici un graphique plus évolué. La courbe de régression sera d'autant plus lissée que le paramètre `span` est élevé.

```
> qplot(Temp, Ozone, data=airquality, colour=Month, geom=c("point","smooth"),
+       facets=~ Month, span=2, facet_grid) + geom_smooth(method="lm",linetype=2)
```



Un layer séparé permet de mieux gérer ses paramètres. Par exemple les méthodes de régression disponibles sont : `lm` (modèle linéaire), `glm` (modèle linéaire généralisé), `gam` (modèle additif généralisé), `loess` (régression polynomiale), `rlm` (régression linéaire robuste)

```
> qplot(wt, mpg, data = mtcars) + stat_smooth(method="loess", se=T)
```

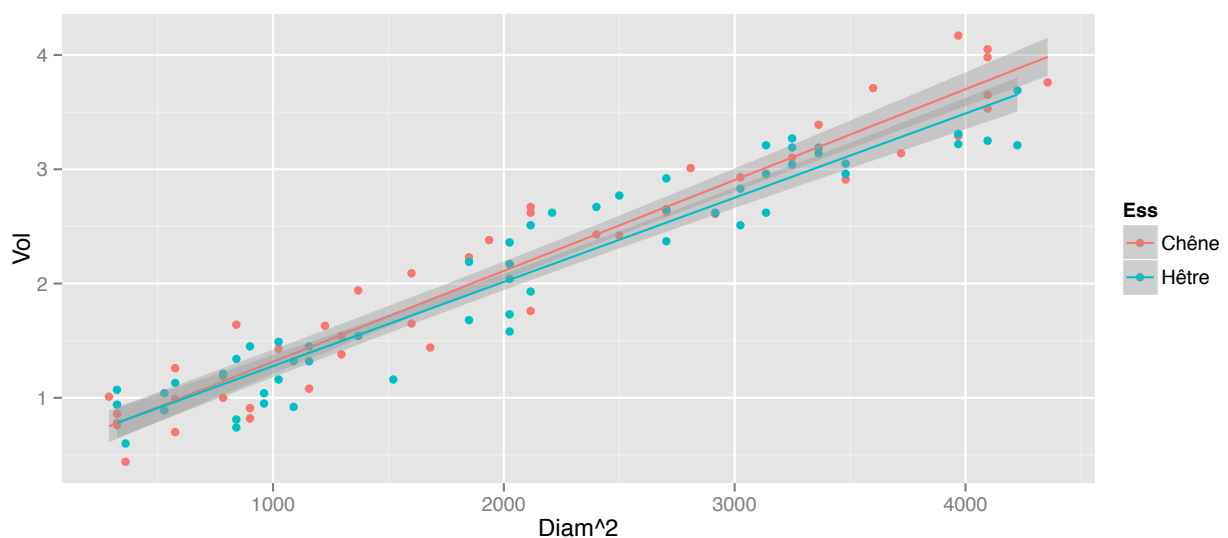


Il est également possible de faire d'autres types de régression

```
> qplot(wt, mpg, data = mtcars) + stat_smooth(method="lm", se=T, formula = y ~ poly(x, 2))
> library(mgcv)
> qplot(carat, price, data = diamonds, geom = c("point", "smooth"),
+       method = "gam", formula = y ~ s(x))
> qplot(carat, price, data = diamonds, geom = c("point", "smooth"),
+       method = "gam", formula = y ~ s(x, bs = "cs"))
> library(splines)
> qplot(wt, mpg, data=mtcars, geom=c("point", "smooth"), method="lm", formula = y ~ ns(x,5))
> library(MASS)
> qplot(wt, mpg, data=mtcars, geom=c("point", "smooth"), method="rlm")
```

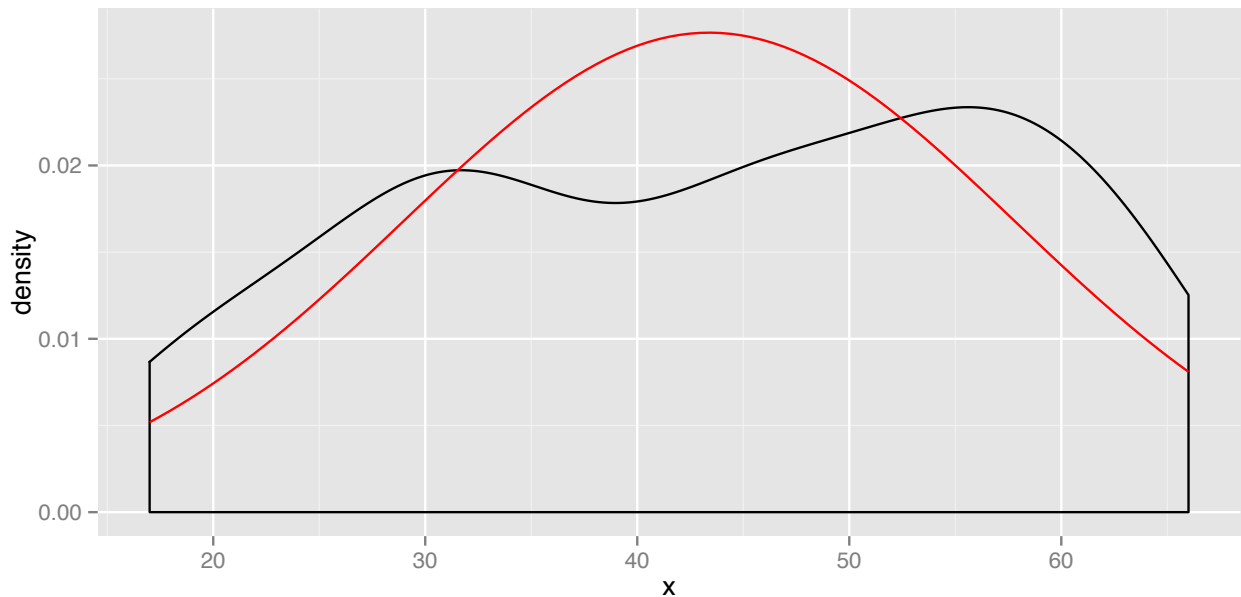
qplot accepte des transformations de variables

```
> t <- read.csv("../Data/Tarif.csv", h = T, sep = ";", dec=",")
> p <- qplot(Diam^2, Vol, data = t, color = Ess)
> p + geom_smooth(aes(group=Ess), method="lm")
```



On peut ajouter une fonction

```
> x <- t$Diam
> qplot(x, geom="density") + stat_function(fun = dnorm, colour="red",
+     arg = list(mean = mean(x), sd = sd(x)))
```



Comparaison variable continue ou discrète

```
> # continuous scale vs. discrete scale
> qplot(wt, mpg, data=mtcars, colour=cyl) # erreur car pas de valeur 5 ou 7 pour cyl
> qplot(wt, mpg, data=mtcars, colour=factor(cyl))
```

## 4.3 ggplot

Par rapport à `qplot`, `ggplot` possède deux arguments, les données et l'aesthetic mapping (paramètres esthétiques de la projection, doivent être inclus dans la fonction `aes()`). Cette objet peut ensuite être mis en forme par des layers.

### 4.3.1 Premiers pas

#### 4.3.2 Différentes formulations

```
> n <- 10
> df <- data.frame(gp = factor(rep(letters[1:3], each = n)), y = rnorm(3*n),
+     op = rep(c("Chêne", "Hêtre"), 15))
> # Première formulation
> ggplot(df, aes(x = gp, y = y)) + geom_point()
> # Résultat équivalent
> ggplot() + geom_point(data = df, aes(x = gp, y = y))
```

La deuxième formulation présente l'avantage de séparer la création du graphique, des données. Elle est sans doute plus lisible.

Quelques exemples à tester :

```
> library(plyr)
> n <- 100
> df <- data.frame(gp = factor(rep(letters[1:3], each = n)), y = rnorm(3*n),
+     op = rep(c("Chêne", "Hêtre"), 1.5*n))
```

```

> ds <- ddpoly(df, .(gp), summarise, mean = mean(y), sd = sd(y))
> p <- ggplot()
> p + geom_point(data = df, aes(x = gp, y = y))
> p + geom_point(data = df, aes(x = gp, y = y), position = "jitter")
> p <- ggplot(data = df, aes(x = gp, y = y))
> p + geom_jitter(position = position_jitter(width = .2), aes(colour = op))
> ggplot(data = df, aes(x = gp, y = y)) +
+   geom_jitter(position = position_jitter(width = .2)) +
+   geom_point(data = ds, aes(x = gp, y = mean), colour = 'red', size = 3) +
+   geom_errorbar(data = ds, aes(x = gp, y = mean, ymin = mean - sd/n^0.5,
+     ymax = mean + sd/n^0.5), colour = 'red', width = 0.4)
> # Sans avoir besoin d'utiliser le package pylr
> ggplot(data = df, aes(x = gp, y = y)) +
+   geom_jitter(position = position_jitter(width = .2)) +
+   stat_summary(fun.y=mean, geom="point", shape=20, size=4, color="red")

```

### 4.3.3 Mise en forme

#### Thèmes

Le thème contrôle l'apparence globale (police, couleurs). On peut par exemple utiliser un thème existant. La modification va impacter tous les graphiques suivants.

```

> qplot(mpg, wt, data = mtcars)
> theme_set(theme_bw())
> qplot(mpg, wt, data = mtcars)
> theme_set(theme_grey())
> qplot(mpg, wt, data = mtcars)

```

On peut aussi définir un nouveau thème.

```

> old <- theme_update(panel.background = element_rect(colour = "red"))
> qplot(mpg, wt, data = mtcars)
> theme_set(old)
> theme_get()

```

#### Modification étiquettes

```

> p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
> p + labs(title = "New plot title")
> p + labs(x = "New x label")
> p + xlab("New x label")
> p + ylab("New y label")
> p + ggtitle("New plot title")
> ggplot(mtcars, aes(factor(gear), mpg)) + geom_point(aes(color=wt)) +
+   xlab("Mon abscisse") + ylab("Mon ordonnée")

```

#### Gestion légende

```

> # changing name of legend (bug: in labs you must use "colour", "color" doesn't work)
> p <- qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point")
> p + labs(colour="Ma légende")
> p + labs(colour="Ma légende") + labs(title = "Pac man")
> # removing legend
> qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point")
> + scale_color_discrete(legend=FALSE)
> qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point") +

```

```

+   opts(legend.position="none")
> # Autre façon de supprimer la légende
> df <- data.frame(cond = factor( rep(c("A","B"), each=200) ),
+                   rating = c(rnorm(200),rnorm(200, mean=.8)))
> ggplot(df, aes(x=cond, y=rating, fill=cond)) + geom_boxplot() + guides(fill=FALSE)
> # moving legend to another place
> qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point") +
+   opts(legend.position="left")
> # changing labels on legend
> qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point") +
+   scale_colour_discrete(name="Legend for cyl", breaks=c("4","6","8"), labels=c("four","six","eight"))
> # reordering breaks (values of legend)
> qplot(mpg, wt, data=mtcars, colour=factor(cyl), geom="point") +
+   scale_colour_discrete(name="Legend for cyl", breaks=c("8","4","6"))

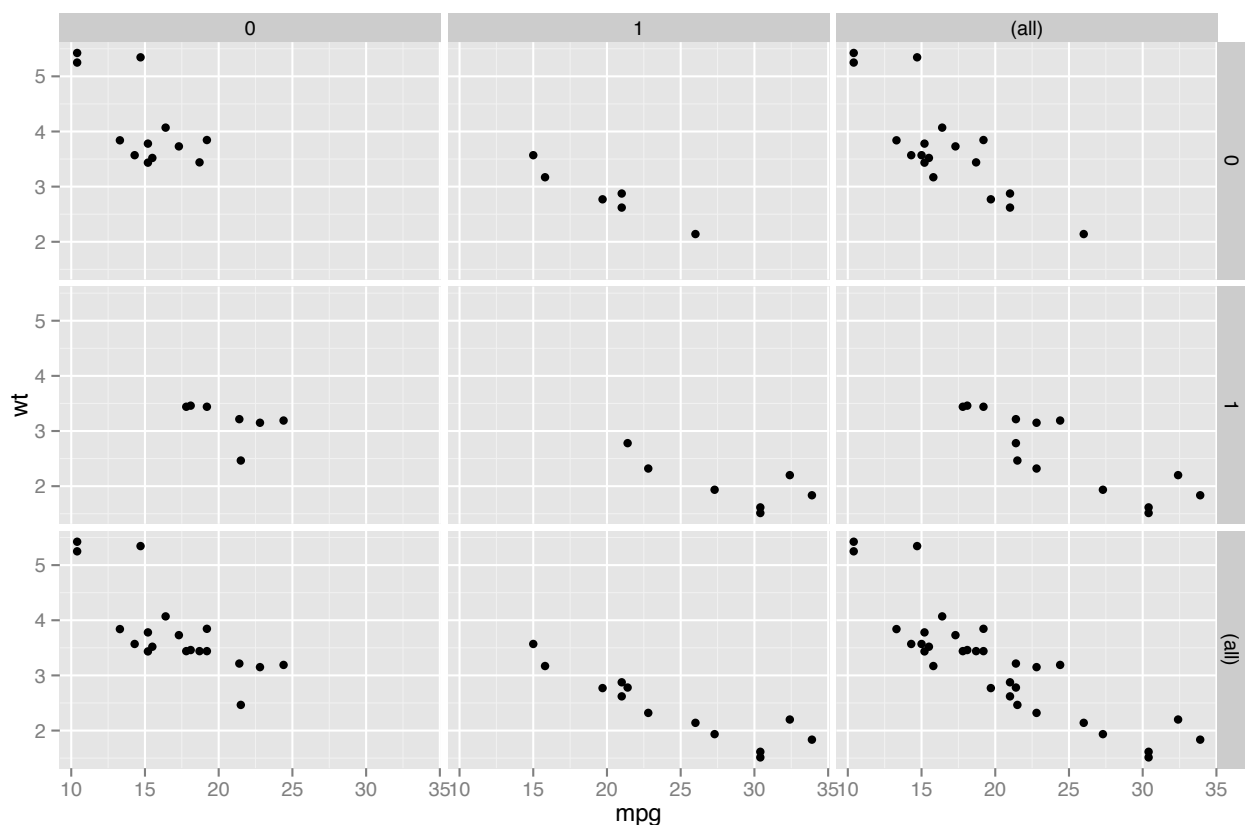
```

#### 4.3.4 Sous-ensembles

```

> p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
> p + facet_grid(vs ~ am, margins=TRUE)
> p + facet_grid( ~ vs + am)

```

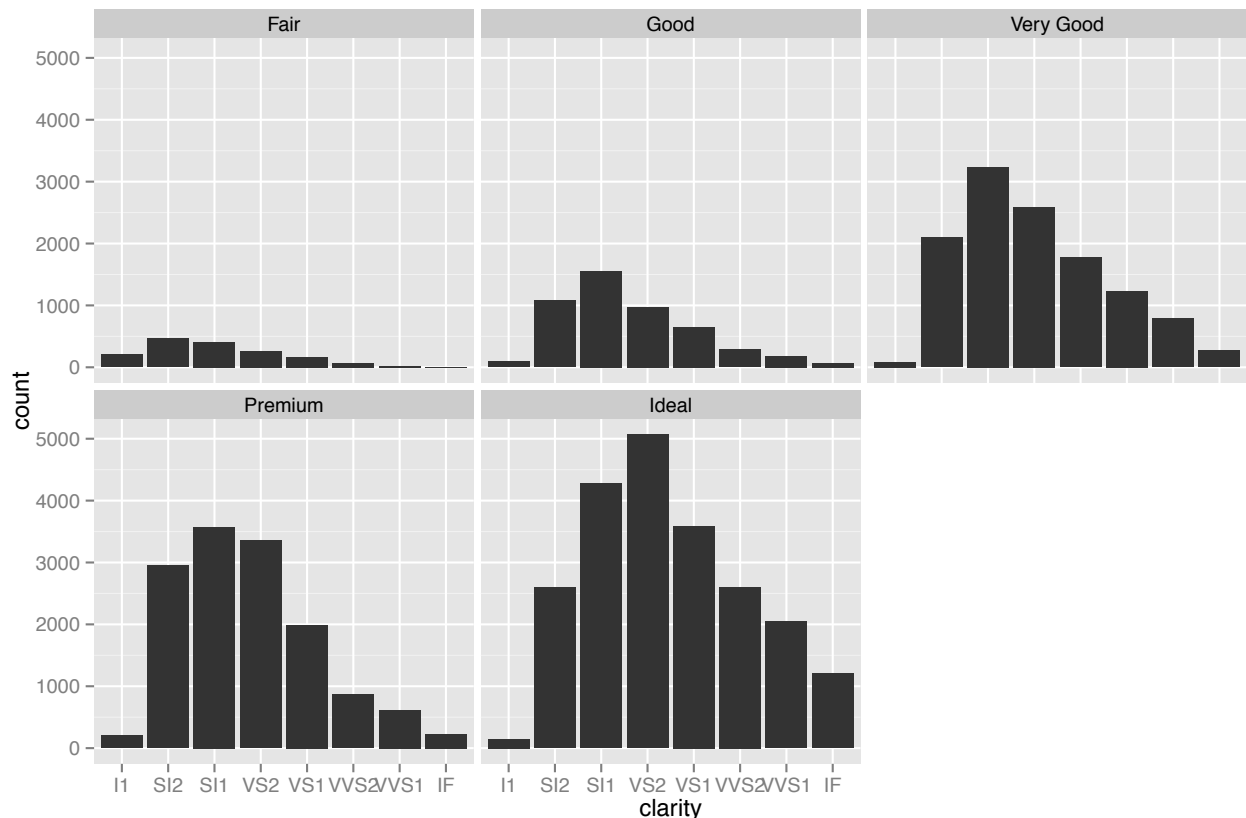


```

> mt <- ggplot(mtcars, aes(mpg, wt, colour = factor(cyl))) + geom_point()
> mt + facet_grid(vs ~ am, scales = "free", space="free", labeller = label_both)
> d <- ggplot(diamonds, aes(carat, price, fill = ..density..)) +
+   xlim(0, 2) + stat_binhex(na.rm = TRUE) + theme(aspect.ratio = 1)
> d + facet_wrap(~ color)
> d + facet_wrap(~ color + cut)

```

```
> ggplot(diamonds, aes(clarity)) + geom_bar() + facet_wrap(~ cut)
```



### 4.3.5 Nuages de points

```
> ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) + geom_point()
> ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) + geom_line()
> ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) + layer(geom="line")
> ggplot(mpg, aes(displ, hwy, colour = factor(year))) + geom_point(shape = 2)
> ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) + geom_point() + geom_smooth(method = "lm")
> ggplot(mpg, aes(displ, hwy)) + geom_text(label=mpg$manufacturer, size=5)
> # Modification des variables de départ
> ggplot(diamonds, aes(log(carat), log(price))) + geom_point()
> ggplot(diamonds, aes(carat, price)) + geom_point() + coord_trans(x = "log10", y = "log10")
> ggplot(diamonds, aes(carat, x * y * z)) + geom_point()
> # Transparence
> ggplot(diamonds, aes(log(carat), log(price))) + geom_point(alpha = 0.1)
> ggplot(diamonds, aes(log(carat), log(price), alpha = carat)) + geom_point()
> ggplot(diamonds, aes(log(carat), log(price))) +
+   geom_point(alpha = diamonds$carat/max(diamonds$carat))
> # Noter la différence entre les deux écritures suivantes
> ggplot(mtcars, aes(wt, mpg, color=qsec, size=3)) + geom_point()
> ggplot(mtcars, aes(wt, mpg, color=qsec)) + geom_point(size=3)
> # Représentation de plus de 2 variables
> ggplot(mtcars, aes(wt, mpg, color=factor(cyl), size=qsec)) + geom_point()
> ggplot(mtcars, aes(wt, mpg, color=factor(carb), size=qsec, shape=factor(cyl))) + geom_point()
> ggplot(mtcars, aes(wt, mpg, color=factor(carb), size=qsec)) + geom_point(shape=1)
> # Trois écritures produisant le même résultat
> ggplot(mtcars, aes(wt, mpg)) + geom_point() + geom_smooth()
> qplot(wt, mpg, data=mtcars) + geom_point() + geom_smooth()
```



```

> qplot(wt, mpg, data=mtcars, geom=c("point", "smooth"))
> ggplot(mtcars, aes(wt, mpg)) + geom_point() + geom_smooth(se=FALSE, span=0.2)
> ggplot(mtcars, aes(wt, mpg)) + geom_point() + geom_smooth(se=FALSE, span=1)
> ggplot(mtcars, aes(wt, mpg, color=factor(cyl))) + geom_point() + geom_smooth()
> qplot(wt, mpg, data=mtcars, geom=c("point", "smooth"), method="lm")
> qplot(mpg, wt, data=mtcars, facets=cyl~., geom=c("point", "smooth")) + coord_flip()
> p <- ggplot(mtcars, aes(x=wt, y=mpg, label=rownames(mtcars))) + geom_point()
> p + geom_text(hjust=-0.1, vjust=0.1, size=4, angle = 10)
> p + geom_text(aes(size=wt)) + scale_size(range=c(3,6))
> p + geom_text(vjust=1, aes(label = paste(wt, "^(", cyl, ")", sep = "")), parse = TRUE)
> p + annotate("text", label = "plot mpg vs. wt", x = 4, y = 30, size = 6, colour = "red")
> p <- ggplot(mtcars, aes(x=wt, y=mpg, label=rownames(mtcars)))
> p + geom_text(aes(family=c("serif", "mono")[am+1], fontface=(am+1)*4))
>

```

La fonction `geom_point` possède des arguments : `alpha`, `colour`, `fill`, `shape`, `size`

### Comparaison `qplot` / `ggplot`

```

> df <- data.frame(cond = factor( rep(c("A","B"), each=200) ),
+                       rating = c(rnorm(200),rnorm(200, mean=.8)))
> # Même graphique, mais écriture directe de l'abscisse pour ggplot
> qplot(df$rating, binwidth=.5)
> ggplot(df, aes(x=rating)) + geom_histogram(binwidth=.5)
> # Meilleure séparation des éléments du graphique
> ggplot(df, aes(x=rating)) + geom_histogram(binwidth=.5, colour="black", fill="white")

```

### 4.3.6 Histogrammes

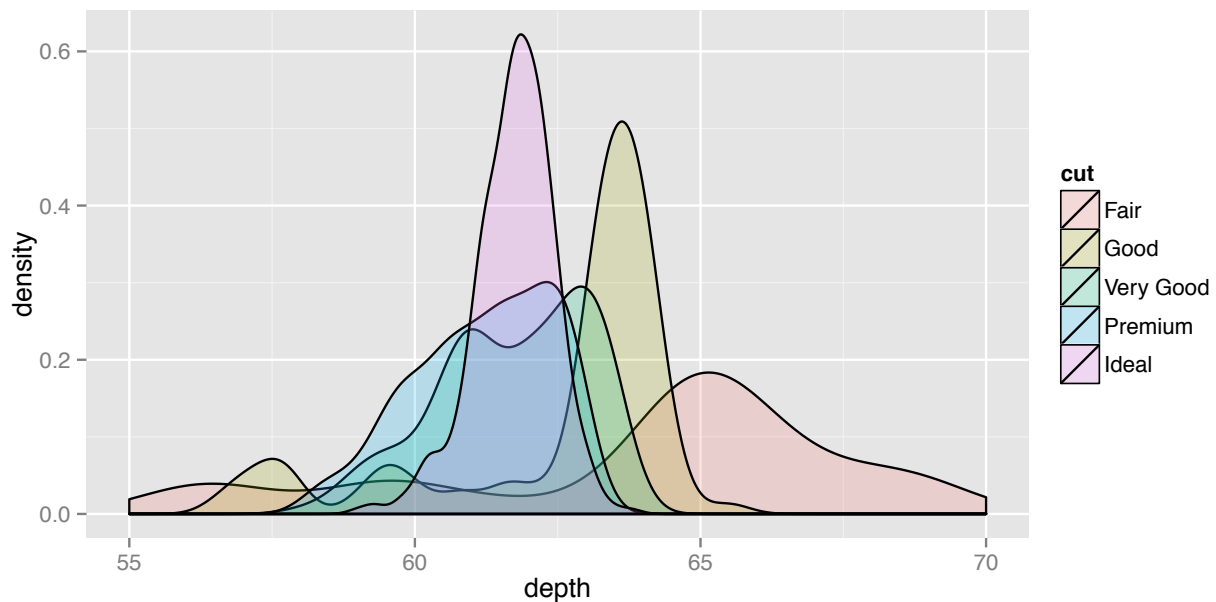
```

> ggplot(mtcars) + geom_histogram(aes(x = mpg), binwidth=3)
> ggplot(mtcars) + geom_histogram(aes(x = mpg), binwidth=3) +
+   geom_freqpoly(aes(x = mpg), binwidth=3, color="red")
> ggplot(mtcars, aes(factor(cyl))) + geom_bar() + coord_flip()
> ggplot(mtcars, aes(factor(cyl), fill=factor(gear))) + geom_bar()
> ggplot(diamonds, aes(clarity, fill=cut)) + geom_bar()

> df <- data.frame(cond = factor( rep(c("A","B"), each=200) ),
+                       rating = c(rnorm(200),rnorm(200, mean=.8)))
> ggplot(df, aes(x=rating)) + geom_density() # Courbe de densité pas terrible
> ggplot(df, aes(x=rating, fill=cond)) + geom_density(alpha=.3) # Mieux
> # Encore mieux : histogram + densité
> ggplot(df, aes(x=rating)) +
+   geom_histogram(aes(y=..density..), # Histogram with density instead of count on y-axis
+                 binwidth=.5,
+                 colour="black", fill="white") +
+   geom_density(alpha=.2, fill="blue") # Overlay with transparent density plot
> ggplot(df, aes(x=rating)) + geom_histogram(binwidth=.5, colour="black", fill="white") +
+   geom_vline(aes(xintercept=mean(rating, na.rm=T)), # Ignore NA values for mean
+             color="red", linetype="dashed", size=1)
> # Histogrammes superposés
> ggplot(df, aes(x=rating, fill=cond)) +
+   geom_histogram(binwidth=.5, alpha=.5, position="identity")
> # Interleaved histograms
> ggplot(df, aes(x=rating, fill=cond)) + geom_histogram(binwidth=.5, position="dodge")
> ggplot(diamonds, aes(x=clarity, fill=cut)) + geom_histogram(position="fill")
> qplot(clarity, data=diamonds, geom="bar", fill=cut, position="fill") # Idem ligne précédente

```

```
> set.seed(6298)
> diamonds_small <- diamonds[sample(nrow(diamonds), 1000), ]
> ggplot(diamonds_small, aes(depth, fill = cut)) + geom_density(alpha = 0.2) + xlim(55, 70)
```



#### 4.3.7 DotPlot

```
> p <- ggplot(mtcars, aes(x = factor(vs), fill = factor(cyl), y = mpg))
> p + geom_dotplot(binaxis = "y", stackdir = "center", position = "dodge")
```

#### 4.3.8 Courbes

```
> ggplot(diamonds, aes(x=clarity, colour=cut)) + geom_polygon(group=cut, position="identity")
> qplot(clarity, data=diamonds, geom="freqpoly", group=cut, colour=cut, position="identity")
> qplot(clarity, data=diamonds, geom="freqpoly", group=cut, colour=cut, position="stack")

> df <- data.frame(x = sort(rnorm(50)), trt = sample(c("a", "b"), 50, rep = TRUE))
> qplot(seq_along(x), x, data = df, geom="step", colour = trt)
```

#### 4.3.9 Camemberts

```
> ggplot(diamonds, aes(x = "", fill = cut)) + geom_bar(width = 1) + coord_polar(theta = "y")
```

#### 4.3.10 Boîtes à moustaches

```
> p <- ggplot(diamonds, aes(x = cut, y = price))
> p + geom_boxplot()
> p + geom_boxplot() + coord_flip()
> p + geom_boxplot(notch = TRUE)
> p + geom_boxplot(notch = TRUE, notchwidth = .3)
> p + geom_boxplot(outlier.colour = "green", outlier.size = 3)
> p + geom_boxplot(aes(fill = cyl))
> p + geom_boxplot(aes(fill = factor(cyl)))
> p + geom_boxplot(aes(fill = factor(am)))
```

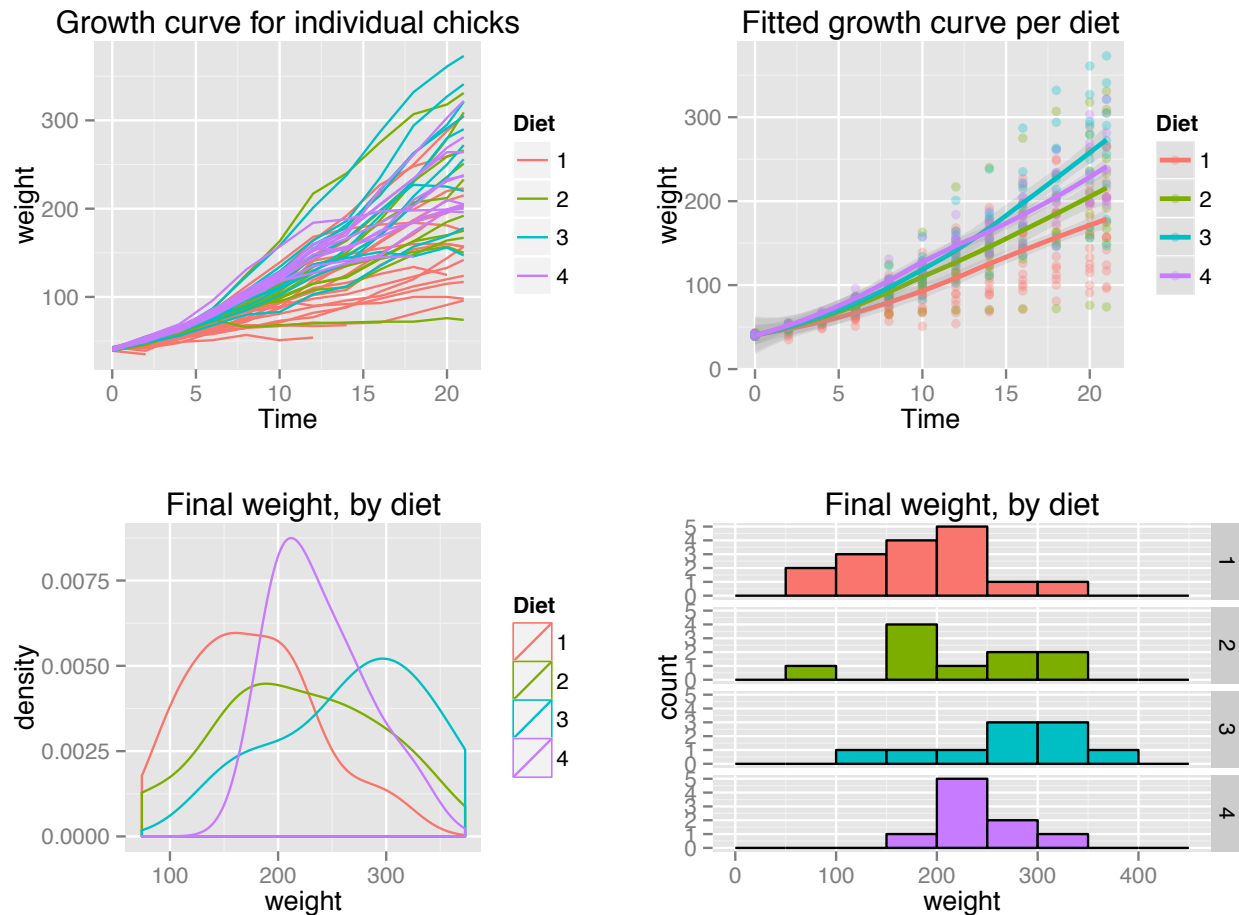
```
> p <- ggplot(movies, aes(y = votes, x = rating, group = round_any(rating, 0.5)))
> p + geom_boxplot() + scale_y_log10() + coord_trans(y = "log10")
> p <- ggplot(mpg, aes(reorder(class, hwy), hwy))
> p + geom_jitter() + geom_boxplot()
```

#### 4.3.11 Dessin de fonctions

```
> # Deux fonctions sur le même graphique
> f <- ggplot(data.frame(x = c(0, 10)), aes(x))
> f + stat_function(fun = sin, colour = "red") + stat_function(fun = cos, colour = "blue")
> # Utilisation d'une fonction personnelle
> test <- function(x) {x ^ 2 + x + 20}
> f + stat_function(fun = test)
```

#### 4.3.12 Graphiques multiples

```
> library(ggplot2)
> library(grid)
> p1 <- ggplot(ChickWeight, aes(x=Time, y=weight, colour=Diet, group=Chick)) +
+   geom_line() + ggtitle("Growth curve for individual chicks")
> p2 <- ggplot(ChickWeight, aes(x=Time, y=weight, colour=Diet)) + geom_point(alpha=.3) +
+   geom_smooth(alpha=.2, size=1) + ggtitle("Fitted growth curve per diet")
> p3 <- ggplot(subset(ChickWeight, Time==21), aes(x=weight, colour=Diet)) +
+   geom_density() + ggtitle("Final weight, by diet")
> p4 <- ggplot(subset(ChickWeight, Time==21), aes(x=weight, fill=Diet)) +
+   geom_histogram(colour="black", binwidth=50) + facet_grid(Diet ~ .) +
+   ggtitle("Final weight, by diet") + theme(legend.position="none")
> grid.newpage()
> layout <- matrix(1:4, 2, 2, byrow=T)
> pushViewport(viewport(layout = grid.layout(nrow(layout), ncol(layout))))
> print(p1, vp = viewport(layout.pos.row = 1, layout.pos.col = 1))
> print(p2, vp = viewport(layout.pos.row = 1, layout.pos.col = 2))
> print(p3, vp = viewport(layout.pos.row = 2, layout.pos.col = 1))
> print(p4, vp = viewport(layout.pos.row = 2, layout.pos.col = 2))
```



### 4.3.13 Librairie plyr

```
> library("plyr")
> ddply(diamonds, "cut", "nrow")
> ddply(diamonds, c("cut", "clarity"), "nrow")
> ddply(diamonds, "cut", summarise, meanDepth = mean(depth))
> ddply(diamonds, "cut", summarise, lower = quantile(depth, 0.25, na.rm=TRUE),
+       median = median(depth, na.rm=TRUE), upper = quantile(depth, 0.75, na.rm=TRUE))
>
```

### 4.3.14 Cartographie

```
> states <- data.frame(map("state", plot=FALSE)[c("x","y")])
> library(mapproj)
> usamap <- qplot(x, y, data=states, geom="path")
> usamap + coord_map(project="lagrange")

crs="+init=epsg :2154")

> library(ggplot2)
> data(diamonds)
> diamonds <- diamonds[sample(nrow(diamonds), 500), ]
> # ---- La fonction quick-plot
> qplot(carat, price, data = diamonds)
> qplot(carat, price, data = diamonds, colour = I("red"), size = I(2), shape = I(15),
```

```
+       alpha = I(1/5))
> qplot(color, price/carat, data = diamonds, geom = "boxplot")
> qplot(carat, price, data = diamonds, geom = c("point","smooth"))

> qplot(mpg, wt, data = mtcars) + geom_smooth(aes(colour = factor(am)), method = "lm") +
+   facet_grid(cyl ~ .)
> p <- ggplot(mpg, aes(displ, hwy, colour = factor(cyl)))
> p + geom_point() + geom_smooth(method = "lm")
> coef <- coef(lm(mpg ~ wt, data = mtcars))
> p <- ggplot(mtcars, aes(wt, mpg))
> p + geom_point() + geom_abline(intercept = coef[1], slope = coef[2], colour = "red",
+                               size = 1.25, linetype = 5)
> library(scales)
> p <- ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()
> p + geom_text(aes(wt, mpg, label = rownames(mtcars)), hjust = -0.1, size = 2.5,
+              color = alpha("blue",0.5)) + xlab("Poids") + ylab("Consommation")
```



# Chapitre 5

## Dessins

### 5.1 Graphiques 3D

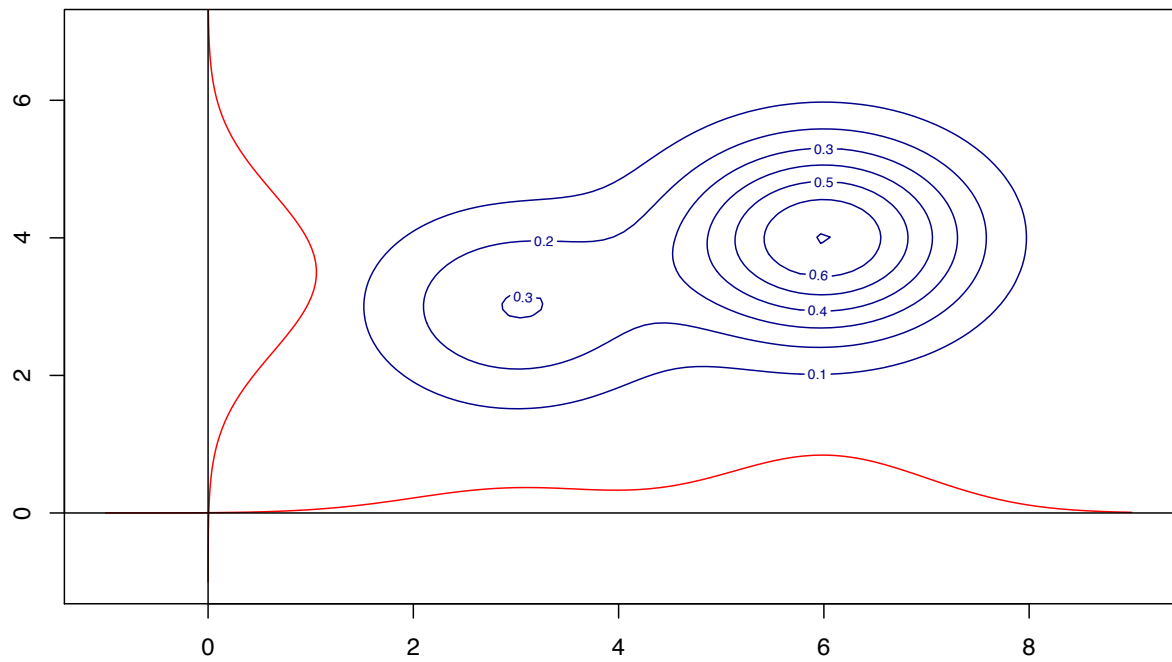
`image(x, y, z, ...)` : trace une grille de rectangles utilisant des couleurs différentes pour représenter la valeur de `z`.

`contour(x, y, z, ...)` : trace des lignes de niveau (contour lines) afin de représenter la valeur de `z`.

`persp(x, y, z, ...)` : trace une surface en 3D.

#### 5.1.1 Courbes de niveau

```
> x <- seq(-1, 9, length = 100)
> y <- seq(-1, 7, length = 100)
> z <- outer(x, y, function(x, y) 0.3 * exp(-0.5 * ((x - 3)^2 + (y - 3)^2)) + 0.7 * exp(-0.5 * ((x - 3)^2 + (y - 3)^2)))
> contour(x, y, z, col = "blue4")
> curve((0.3 * dnorm(x, mean = 3) + 0.7 * dnorm(x, mean = 6)) * 3, -1, 9, col = "red", ylim = c(-1, 7))
> lines((0.5 * dnorm(x, mean = 3) + 0.5 * dnorm(x, mean = 4)) * 3, x, col = "red")
> abline(h = 0)
> abline(v = 0)
```

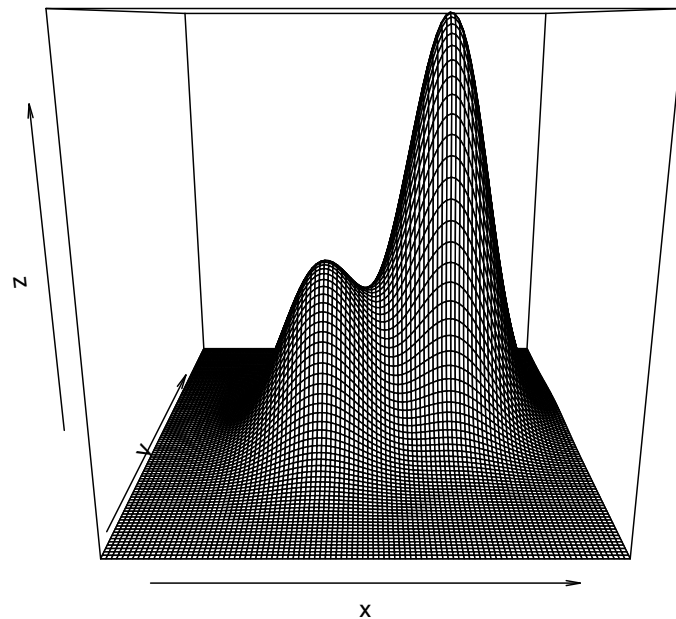


La fonction `outer` évite une double boucle. Elle permet le calcul de la fonction  $f$  pour toutes les combinaisons des variables  $x$  et  $y$ .

### 5.1.2 Représentation en 3D

```
> persp(x, y, z)
```





```
> library(rgl)
> data(volcano)
> z <- 2 * volcano      # Exaggerate the relief
> x <- 10 * (1:nrow(z)) # 10 meter spacing (S to N)
> y <- 10 * (1:ncol(z)) # 10 meter spacing (E to W)
> ylim <- range(z)
> ylen <- ylim[2] - ylim[1] + 1
> colorlut <- terrain.colors(ylen) # height color lookup table
> col <- colorlut[z - ylim[1]+1 ] # assign colors to heights for each point
> rgl.open()
> rgl.surface(x, y, z, color=col, back="lines")
```

La bibliothèque rgl permet de faire tourner la figure. Elle contient également la fonction plot3D

```
> plot3d(x, y, y, col=rainbow(1000), radius=2, type="s")
```

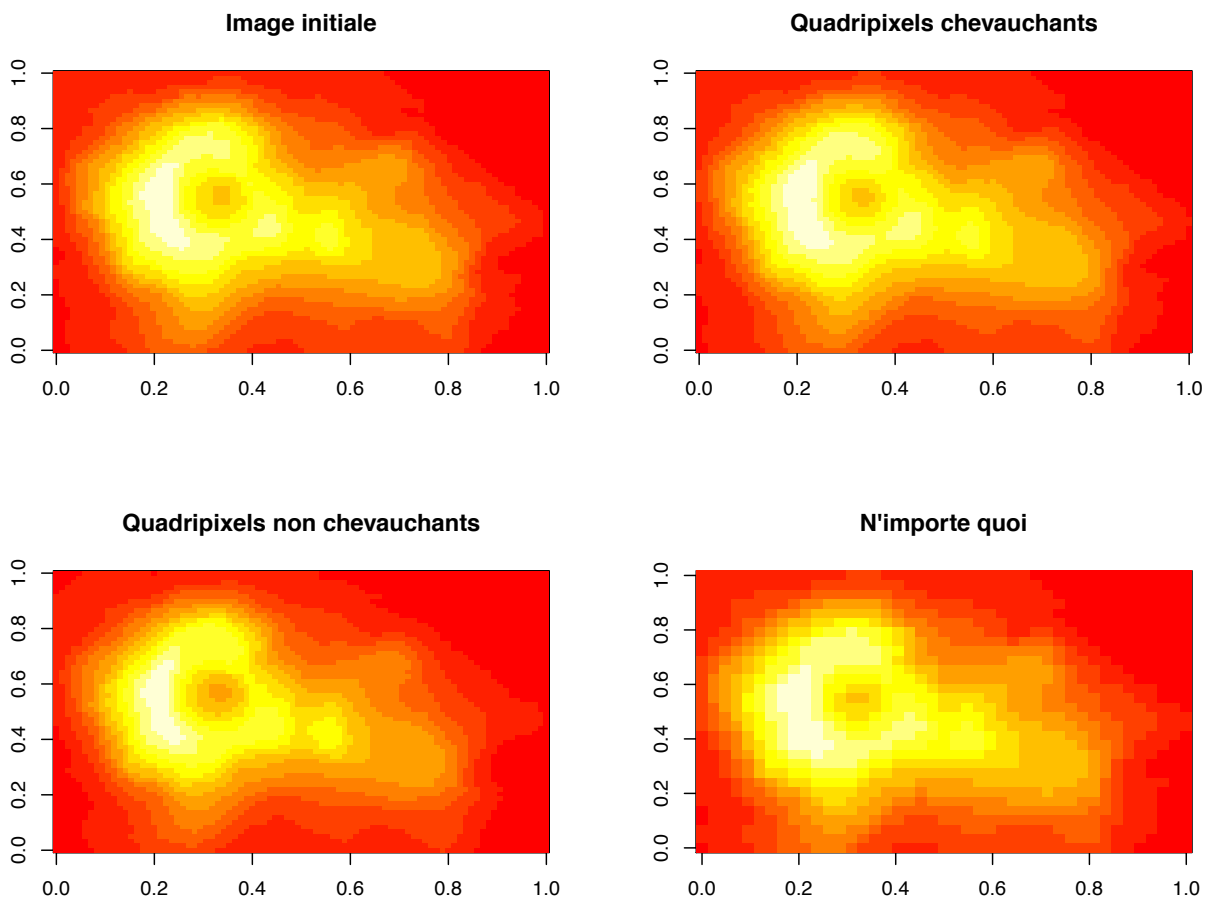
## 5.2 Traitement d'image

```
> data(volcano)
> M <- volcano
> n <- dim(M)[1]
> m <- dim(M)[2]
> M1 <- M [1:(n-1),] [,1:(m-1)]
> M2 <- M [2:n,] [,1:(m-1)]
> M3 <- M [1:(n-1),] [,2:m]
> M4 <- M [2:n,] [,2:m]
> # Avec des quadripixels qui se chevauchent
> M0 <- (M1+M2+M3+M4)/4
```

```

> # Avec des quadripixels qui ne se chevauchent pas
> nn <- floor((n-1)/2)
> mm <- floor((m-1)/2)
> M00 <- (M1*M2+M3-M4)
> # Avec une fonction quelconque
> M000 <- M0 [2*(1:nn),] [,2*(1:mm)]
> # ---- Dessins
> par(mfrow=c(2,2))
> image(M, main="Image initiale")
> image(M0, main="Quadripixels chevauchants")
> image(M00, main="Quadripixels non chevauchants")
> image(M000, main="N'importe quoi")
> par(mfrow=c(1,1))

```



### 5.3 Connexion à Google Map

```

> library(RgoogleMaps)
> MyMap <- GetMap(center=c(48.6932,6.18799), zoom = 19, destfile = "tile1.png", maptype = "satellite")

[1] "http://maps.google.com/maps/api/staticmap?center=48.6932,6.18799&zoom=19&size=640x640&maptype=sate

> PlotOnStaticMap(MyMap)

```





## Chapitre 6

# Analyse des données

### 6.1 Exploration

Pour les facteurs, `table` calcule les effectifs observés pour chaque niveau, ou les combinaisons de niveaux pour plusieurs facteurs.

```
> g1 <- gl(2, 10); g2 <- gl(2, 1, 20); g3 <- gl(2, 2, 20)
> table(g1)
> table(g1, g2)
> table(g1, g2, g3)
> ftable(table(g1, g2, g3))
```

`ftable` (flat table) présente les résultats de `table` sous forme tableaux plats.  
La fonction `summary` est très utile

```
> library(MASS)
> summary(cats)
```

Sex	Bwt	Hwt
F:47	Min. :2.00	Min. : 6.30
M:97	1st Qu.:2.30	1st Qu.: 8.95
	Median :2.70	Median :10.10
	Mean :2.72	Mean :10.63
	3rd Qu.:3.02	3rd Qu.:12.12
	Max. :3.90	Max. :20.50

### 6.2 Régression

#### 6.2.1 Modèle linéaire

```
> cats.lm <- lm(Hwt ~ Bwt*Sex, data=cats)
> summary(cats.lm)
```

Call:

```
lm(formula = Hwt ~ Bwt * Sex, data = cats)
```

Residuals:

Min	1Q	Median	3Q	Max
-3.773	-1.012	-0.120	0.927	4.865

Coefficients:

Estimate	Std. Error	t value	Pr(> t )
----------	------------	---------	----------

```

(Intercept)    2.981    1.843    1.62  0.10796
Bwt            2.636    0.776    3.40  0.00088 ***
SexM          -4.165    2.062   -2.02  0.04526 *
Bwt:SexM       1.676    0.837    2.00  0.04722 *

```

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.44 on 140 degrees of freedom

Multiple R-squared: 0.657, Adjusted R-squared: 0.649

F-statistic: 89.2 on 3 and 140 DF, p-value: <2e-16

cats.lm est un objet de type lm. On peut visualiser ces composantes par `names(cats.lm)` ou `attributes(cats.lm)`.

On peut en extraire notamment

- les coefficients du modèle : `cats.lm$coefficients` ou `coefficients(cats.lm)`

- la formule du modèle : `formula(cats.lm)`

## 6.2.2 Régression logistique

La base de données `birthwt` est issue de la library `MASS`. Les variables sont les suivantes :

- low : petit poids de naissance
- age : age de la mère
- smoke : tabagisme de la mère
- ptl : nombre d'accouchements prématurés
- ht : antécédents d'hypertension

```

> lbwt.glm <- glm(low ~ age + ptl + smoke + ht, data=birthwt, family=binomial)
> summary(lbwt.glm)

```

Call:

```
glm(formula = low ~ age + ptl + smoke + ht, family = binomial,
    data = birthwt)
```

Deviance Residuals:

```

      Min       1Q   Median       3Q      Max
-1.855  -0.818  -0.676   1.128   1.914

```

Coefficients:

```

              Estimate Std. Error z value Pr(>|z|)
(Intercept)   0.0760     0.7854    0.10   0.923
age          -0.0598     0.0336   -1.78   0.075 .
ptl           0.7928     0.3312    2.39   0.017 *
smoke         0.5640     0.3355    1.68   0.093 .
ht            1.2732     0.6199    2.05   0.040 *

```

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 234.67 on 188 degrees of freedom

Residual deviance: 217.18 on 184 degrees of freedom

AIC: 227.2

Number of Fisher Scoring iterations: 4

On peut essayer de retirer une variable.

```
> drop1(lbwt.glm, test="Chisq")
```

Single term deletions

Model:

```
low ~ age + ptl + smoke + ht
```

	Df	Deviance	AIC	LRT	Pr(>Chi)
<none>		217	227		
age	1	220	228	3.34	0.068 .
ptl	1	223	231	6.10	0.013 *
smoke	1	220	228	2.82	0.093 .
ht	1	221	229	4.27	0.039 *

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Résultat : On enlève la variable ayant la p-value la plus élevée, ici smoke.

On peut aussi effectuer une sélection automatique par sélection pas à pas, grâce à la fonction step, qui fait appel à drop1 et add1.

```
> scope.lm<-paste(names(birthwt)[2:10],collapse="+")
> scope.lm<-paste(names(birthwt)[1],scope.lm,sep="~")
> lbwt.full <- glm(as.formula(scope.lm),family=binomial,data=birthwt)
> step(lbwt.full)
```

Start: AIC=20

```
low ~ age + lwt + race + smoke + ptl + ht + ui + ftv + bwt
```

	Df	Deviance	AIC
- smoke	1	0	18
- age	1	0	18
- race	1	0	18
- ui	1	0	18
- ptl	1	0	18
- ftv	1	0	18
- lwt	1	0	18
- ht	1	0	18
<none>		0	20
- bwt	1	204	222

Step: AIC=18

```
low ~ age + lwt + race + ptl + ht + ui + ftv + bwt
```

	Df	Deviance	AIC
- age	1	0	16
- ptl	1	0	16
- lwt	1	0	16
- ftv	1	0	16
- ht	1	0	16
- race	1	0	16
- ui	1	0	16
<none>		0	18
- bwt	1	210	226

Step: AIC=16

```
low ~ lwt + race + ptl + ht + ui + ftv + bwt
```

	Df	Deviance	AIC
- ptl	1	0	14

```

- lwt 1 0 14
- ftv 1 0 14
- race 1 0 14
- ht 1 0 14
- ui 1 0 14
<none> 0 16
- bwt 1 211 225

```

Step: AIC=14

```
low ~ lwt + race + ht + ui + ftv + bwt
```

```

      Df Deviance AIC
- lwt 1 0 12
- ftv 1 0 12
- race 1 0 12
- ht 1 0 12
- ui 1 0 12
<none> 0 14
- bwt 1 215 227

```

Step: AIC=12

```
low ~ race + ht + ui + ftv + bwt
```

```

      Df Deviance AIC
- race 1 0 10
- ht 1 0 10
- ftv 1 0 10
- ui 1 0 10
<none> 0 12
- bwt 1 221 231

```

Step: AIC=10

```
low ~ ht + ui + ftv + bwt
```

```

      Df Deviance AIC
- ht 1 0 8
- ftv 1 0 8
- ui 1 0 8
<none> 0 10
- bwt 1 224 232

```

Step: AIC=8

```
low ~ ui + ftv + bwt
```

```

      Df Deviance AIC
- ftv 1 0 6
- ui 1 0 6
<none> 0 8
- bwt 1 229 235

```

Step: AIC=6

```
low ~ ui + bwt
```

```

      Df Deviance AIC
- ui 1 0 4
<none> 0 6
- bwt 1 230 234

```



```
Step: AIC=4
```

```
low ~ bwt
```

```

      Df Deviance AIC
<none>      0     4
- bwt    1    235 237

```

```
Call: glm(formula = low ~ bwt, family = binomial, data = birthwt)
```

```
Coefficients:
```

```

(Intercept)      bwt
  2975.98      -1.19

```

```
Degrees of Freedom: 188 Total (i.e. Null); 187 Residual
```

```
Null Deviance:      235
```

```
Residual Deviance: 4.93e-07      AIC: 4
```

## 6.3 ACP

Dans les fonctionnalités chargées par défaut 2 fonctions permettent de réaliser une ACP :

- princomp() calcule les valeurs propres sur la matrice de covariance ou de corrélation
- prcomp() utilise la décomposition en valeurs singulières ; prcomp est une généralisation de la méthode princomp.

Il existe également des packages spécifiques dont FactoMineR.

```

> library(FactoMineR)
> data(decathlon)
> res <- PCA(decathlon, quanti.sup=11:12, quali.sup=13)
> # ----- Impressions supplémentaires
> plot(res,invisible="quali")
> plot(res,choix="var",invisible="quanti.sup")
> plot(res,habillage=13)
> aa=cbind.data.frame(decathlon[,13],res$ind$coord)
> bb=coord.ellipse(aa,bary=TRUE)
> plot.PCA(res,habillage=13,ellipse=bb)
> res$eig
> barplot(res$eig[,1],main="Eigenvalues",names.arg=1:nrow(res$eig))
> res$ind$coord
> res$ind$cos2
> res$ind$contrib
> dimdesc(res)

```

## 6.4 Classifications

Le but d'une classification est de regrouper les objets similaires. La similarité dépend de la métrique retenue. Plusieurs distances sont disponibles dans R. Il existe plusieurs façons de regrouper les objets : soit par un processus hiérarchique, ascendant ou descendant, soit sur la base du "partitionnement" à partir d'un nombre prédéfini de classes.

La bibliothèque cluster propose plusieurs méthodes de classification.

### 6.4.1 La fonction agnes()

Elle permet de réaliser une classification hiérarchique ascendante. Elle retourne une liste composée de plusieurs champs :

- le coefficient d'agglomération ( $\alpha$ )

## Chapitre 7

# Modélisation

### 7.1 Analyse de la qualité de la régression

Il existe des graphiques préétablis.

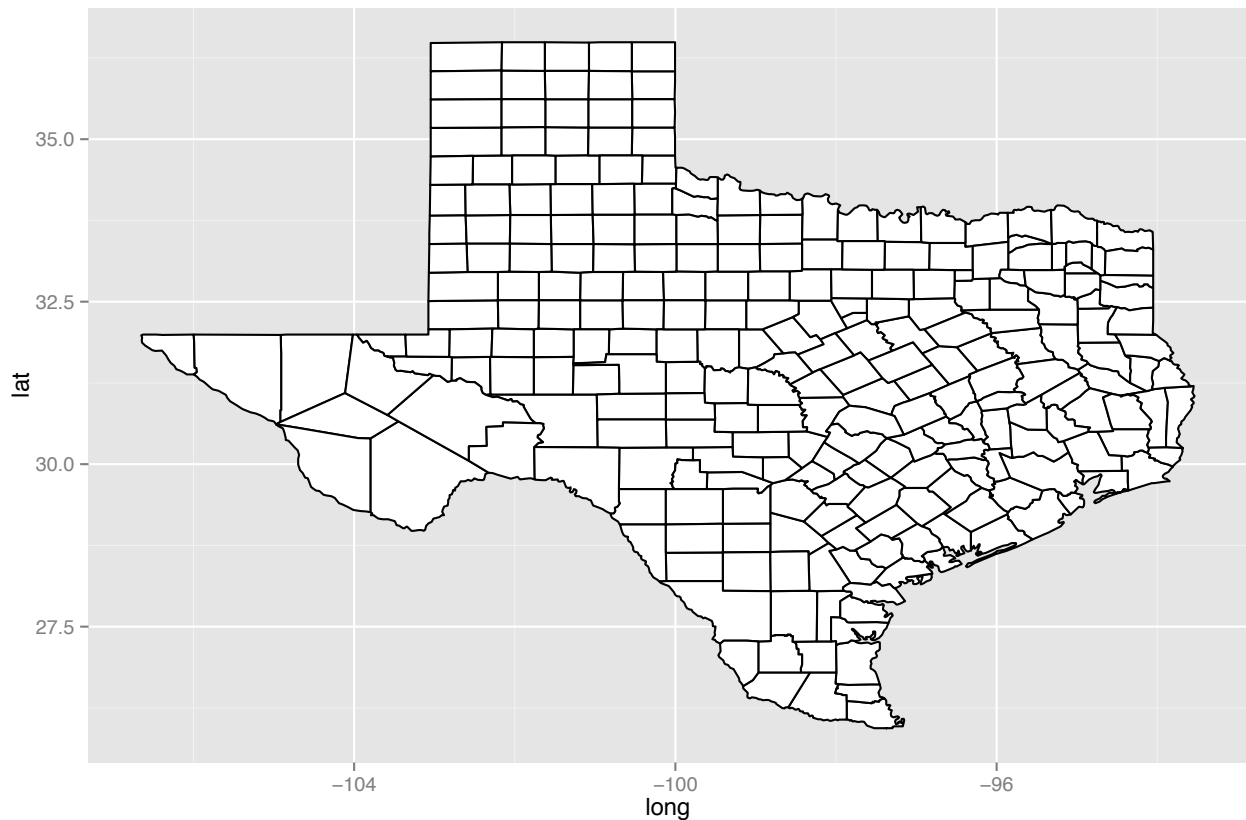
```
> mod <- lm(mpg ~ wt, data = mtcars)
> plot.lm(mod)
> plot(mod, which = 1)
> plot(mod, which = 2)
> plot(mod, which = 3)
> plot(mod, which = 4)
> plot(mod, which = 5)
> plot(mod, which = 6)
```

Il est également possible d'améliorer ces graphiques standards. La fonction `fortify` permet de transférer dans un `data.frame` les éléments de la régression.

```
> mod <- lm(mpg ~ wt, data = mtcars)
> head(fortify(mod))
> head(fortify(mod, mtcars))
> ggplot(mod, aes(.fitted, .stdresid)) + geom_point() + geom_hline(yintercept = 0) + geom_smooth(se = 1)
> ggplot(fortify(mod, mtcars), aes(mpg, .stdresid, colour = factor(cyl))) + geom_point()
```

Elle permet également de transférer la géométrie d'une carte dans un `data.frame`.

```
> library(maps)
> tx <- map("county", "texas", plot = FALSE, fill = T)
> ggplot(tx, aes(long, lat, group = group)) + geom_polygon(colour = "black", fill="white")
```



### 7.1.1 Modèle de croissance exponentiel

```
> malthus <- function(t,N0,r){N0*exp(r*t)} # modèle de Malthus
> t <- seq(0:100)
> N0 <- 2
> r <- 0.05
> plot(malthus(t,N0,r)~t,type='l',col='green',lwd=2)
> #on crée des données en ajoutant du bruit
> sim <- malthus(t,N0,r) + rnorm(t,sd=0.3*malthus(t,N0,r))
> plot(sim~t,pch=20)
> #on ajuste la fonction sur les données simulées en utilisant les moindres carrés
> fitmalthus <- nls(sim~malthus(t,a,b),start=list(a=1,b=0.01))
> fitmalthus
> summary(fitmalthus)
> #on vérifie
> plot(sim~t, pch=20)
> lines(malthus(t,N0,r)~t,type='l',col='green',lwd=2)
> lines(malthus(t,coef(fitmalthus)[1],coef(fitmalthus)[2])~t,type='l',col='red',lwd=2)
```

### 7.1.2 Fonction logistique

```
> logist<-function(t,v,w,z){v/(1+w*exp(-z*t))}
> v<-10;w<-8;z<-0.09;x<-seq(from=-100,to=100)
> plot(logist(x,v,w,z)~x,type='l',col='blue',lwd=2)
> #on crée des données avec du bruit
> noisy<-rnorm(x,mean=logist(x,v,w,z),sd=0.9)
> plot(noisy,pch=20) #notez qu'il y a des valeurs négatives pas réalistes...
> fitlog <- nls(noisy~logist(x,a,b,c),start=list(a=5,b=5,c=0.02))
```

```

> fitlog
> summary(fitlog)
> #on vérifie
> plot(noisy~x,pch=20,col="grey")
> lines(logist(x,v,w,z)~x,type='l',col='blue',lwd=2)
> lines(logist(x,coef(fitlog)[1],coef(fitlog)[2],coef(fitlog)[3])~x,type='l',col='red',lwd=2)

```

Résoudre équation différentielle

```

> library(deSolve)
> library(odesolve)
> # on définit le système dans une fonction six ici
> six <- function(t,x,parms){
+   with( as.list(c(parms,x)),{
+
+     rp<-ap*exp(-bp*t)
+     rs<-as*exp(-0.5*(log(t/gs)/bs)^2)
+
+     dI<- (rp*X*S+rs*I*S)
+     dS<- -(rp*X*S+rs*I*S)
+
+     res<-c(dI,dS)
+     list(res)}}
+ }
> #on définit les paramètres pour la simulation du système
>
> parms<-c(ap=0.002, bp=0.0084, as=5.9e-7, bs=0.25, gs=1396, X=1, N=1010)
> #on crée un vecteur pour le temps
> times<-seq(0:3000)
> #valeurs initiales des variables (ici tous les individus sont sains au début)
>
> y<- xstart <-(c(I = 0, S = 1010))
> #on résout le système avec la fonction lsoda
>
> out<-as.data.frame(lsoda(xstart, times, six, parms))
> # toujours regarder le résultat pour détecter les incohérences
>
> plot(out$S~out$time, type="l",col='blue')
> plot(out$I~out$time, type="l",col='green',lwd=3)
> #on crée un jeu de données en ajoutant un bruit blanc (gaussien)
>
> noisy<-out$I+rnorm(nrow(out),sd=0.15*out$I)
> plot(noisy)
> #on ajuste le modèle sur ces données par les moindres carrés
> # pour cela on utilise la fonction nls, il faut faire attention aux paramètres initiaux
>
> fitnoisy <-nls(noisy~lsoda(xstart,times,six,c(ap=ap, bp=bp, as=as, bs=bs, gs=gs, X=1, N=1010))[,2]
+   ,start=list(ap=0.002, bp=0.0084, as=5.9e-7, bs=0.25, gs=1000)
+   , control=list(minFactor=1e-20,maxiter=150))
> summary(fitnoisy)
> # un beau graphique pour visualiser tout ça!
> plot(noisy,pch=20,col='grey',main="SIX ode Model Fit",xlab="time",ylab="disease progress")
> lines(predict(fitnoisy,times),type='l',col='green',lwd=4.5)

```

### 7.1.3 Estimation des paramètres d'une loi de probabilités

```
> library(MASS)
> z <- c( rep(14,3), rep(17,19), rep(20,26), 23)
> hist(z)
> # on utilise la fonction fitdistr pour une loi Weibull, regarder ?fitdistr
> paraw <- fitdistr(z,densfun="weibull")
> logLik(paraw) # on peut avoir le loglikelihood
> # on visualise les résultats sur un graphique : histogramme+ loi
> hist(z,freq=FALSE) # penser à freq=FALSE!!
> lines(dweibull(0:max(z),shape=paraw$estimate[1],scale=paraw$estimate[2]),type='l',col='green',lwd=2)
```

# Chapitre 8

## Compléments

### 8.1 Commandes de base

Avec R, une fonction, pour être exécutée, s'écrit toujours avec des parenthèses. Si l'opérateur oublie les parenthèses, R affichera le contenu des instructions de cette fonction.

Quelques exemples

`ls()` Liste des objets disponibles

`ls.str()` affiche des détails sur les objets

`getwd()` permet de savoir quel est le répertoire de travail. Si ce n'est pas le bon, on peut le changer avec `menu` changer répertoire.

`help("read.table")` Consulter la documentation sur l'instruction `read.table`

`?read.table` Idem

`rm(x)` permet de supprimer l'objet `x`

`rm(list = ls())` permet de supprimer tous les objets

`rm(list=ls(pattern=".*a.*"))` permet de supprimer tous les objets dont le nom contient la lettre `a`.

`load('.Rdata')` permet de recharger les objets enregistrés

`list.files("../data")` Liste tous les fichiers contenus dans le répertoire `data` situé au niveau supérieur de l'arborescence.

`library()` Liste les bibliothèques disponibles

Deux commandes sur une même ligne sont séparées par le point-virgule.

Pour connaître les méthodes d'une fonction, on tape `methods`, par exemple

```
> methods("plot")
```

```
[1] plot.aareg*      plot.acf*
[3] plot.agnes*      plot.areg
[5] plot.areg.boot   plot.aregImpute
[7] plot.bagplot     plot.biVar
[9] plot.bvevd*      plot.bvpot*
[11] plot.clusGap*    plot.correspondence*
[13] plot.cox.zph*    plot.curveRep
[15] plot.data.frame* plot.decomposed.ts*
[17] plot.default     plot.dendrogram*
[19] plot.density     plot.diana*
[21] plot.drawPlot    plot.ecdf
[23] plot.faces       plot.factor*
[25] plot.formula*    plot.function
[27] plot.gbayes      plot.ggplot*
[29] plot.gtable*     plot.hclust*
[31] plot.histogram*  plot.HoltWinters*
[33] plot.isoreg*     plot.lda*
[35] plot.ldBands     plot.lm
```

[37] plot.mca*	plot.medpolish*
[39] plot.mlm	plot.mona*
[41] plot.partition*	plot.ppr*
[43] plot.prcomp*	plot.princomp*
[45] plot.profile*	plot.profile.evd*
[47] plot.profile.nls*	plot.profile2d.evd*
[49] plot.Quantile2	plot.ridgelm*
[51] plot.rm.boot	plot.shingle*
[53] plot.silhouette*	plot.spec
[55] plot.spline*	plot.stepfun
[57] plot.stl*	plot.summary.formula.response
[59] plot.summary.formula.reverse	plot.survfit*
[61] plot.table*	plot.transcan
[63] plot.trellis*	plot.ts
[65] plot.tskernel*	plot.TukeyHSD
[67] plot.uvevd*	plot.varclus
[69] plot.xyVector*	

Non-visible functions are asterisked

save(a,c,e,file="fichier") pour ne sauver qu'une selection d'objets

## 8.2 Touches

| alt + majuscule + touche L

alt + majuscule + touche ‘

alt + n

## 8.3 Aide

taper ? + fonction

taper help.search("permutation") Plus intéressant RSiteSearch("regression")