Langage R

MASTER I

9 mai 2005

Table des matières

1 Introduction

R est un logiciel de calcul statistique librement disponible sur le réseau CRAN (Comprehensive R Archive Network) à l'adresse `http ://cran.r-project.org`. Le logiciel a été développé aux Bell Labs (USA) et est amélioré continuellement par des volontaires.

2 Exemple d'EDA - éruptions d'un geyser

Les données dans le fichier `faithful` représentent les délais entre éruptions et les durées des éruptions d'un geyser dans le parc Yellowstone aux USA. Nous allons examiner la distribution des ces données à l'aide des graphiques et des statistiques de base. C'est une étape préalable obligatoire avant toute analyse et modélisation statistique.

```
> help(faithful)
> data(faithful)
> attach(faithful)
> summary(eruptions)
> fivenum(eruptions) # résumé à 5 nombres de Tukey
```

Nous pouvons calculer les statistiques de base selon nos besoins.

```
> mean(eruptions)
> median(eruptions)
> min(eruptions)
> range(eruptions)
> quantile(eruptions)
> var(eruptions)   # variance
> sqrt(var(eruptions)) # écart type
```

Les graphiques tige et feuilles donnent une bonne idée de la distribution. L'histogramme est meilleur pour des grandes ensembles de données.

```
> stem(eruptions)
> hist(eruptions)
> # classes plus petites, tracé de la densité
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE);
  lines(density(eruptions, bw=0.1))
> rug(eruptions)  # les données elles-même
```

Des tracés de densité plus sophistiqués sont obtenus avec `density`. La largeur de bande (`bw`) est choisie par tâtonnement parce que la valeur par défaut donne trop de lissage. Nous pouvons tracer la fonction de répartition empirique à l'aide de la fonction `ecdf` définie par

$$F_n(t) = \#\{x_i \leq t\} = \sum_{i=1}^{n} \mathbf{1}_{x_i \leq t}\,.$$

```
> library(stepfun)
> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)
```

La distribution est loin de quelque chose de standard. Mais nous pouvons isoler les éruptions autour de la mode de droite qui sont de durée plus grande que 3 minutes. Nous ajustons une distribution normale que nous superposons sur le cdf.

```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```

Un graphique de quantiles peut nous aider.

```
> par(pty="s")  # tracé carré
> qqnorm(long); qqline(long)
```

Il montre une ajustage raisonnable, mais la queue droite est plus courte qu'une distribution normale. Nous comparons avec des données simulées d'une distribution-t

```
> x <- rt(250, df=5)
> qqnorm(x); qqline(x)
```

qui normalement (étant un échantillon aléatoire) montre des queues plus longues qu'attendues pour une loi normale. Nous pouvons faire un graphique quantile contre la distribution t par

```
> qqplot(qt(ppoints(250), df=5), x, xlab="Q-Q graphique pour t")
> qqline(x)
```

Finalement on peut appliquer deux tests de normalité.

```
> library(ctest)
> shapiro.test(long)
> ks.test(long, "pnorm", mean=mean(long), sd=sqrt(var(long)))
```

3 Les essentiels

- lancement et arrêt : `$ R, > quit()`
- aide-en-ligne : `> help.start(), >?plot`
- ligne de commande :
  - sensible à la casse, séparation par « ; »,
  - commentaires précédés par « # »,
  - rappel de commandes et édition ;
- fichiers de commandes et de sortie :
  - `> source("commands.R")`,
  - `> sink("record.lst")`
- espace de travail :
  - `> objects/ls(), > rm(x,y,foo,bar)`,
  - sauvegarde facultative dans `.Rdata .`

4 Manipulation de données

R travaille sur des objets (structures de données) qui ont tous deux attributs intrinsèques : `mode` et `length`. Le `mode` est le type des éléments d'un objet ; il en existe quatre : `numeric, character, complex,` et `logical`. D'autres modes existent qui ne représentent pas des données, par exemple : `function, expression, formula`. La `length` est le nombre total d'éléments de l'objet. Voir le tableau dans « R pour les débutants », page 4, qui donne un résumé des différents objets manipulés par R : `vector, factor, array, matrix, data.frame, list`

- vecteur est l'objet de base ;
- matrices sont des généralisations multi-d des vecteurs (à 2 indices ou plus) ;
- facteurs permettent de gérer des données catégoriques efficacement ;
- dataframes sont des structures matricielles dont les colonnes peuvent être de types différents ; par exemple une ligne par observation ayant des variables numériques et catégoriques ;
- fonctions sont elles même des objets qui peuvent être stockées dans l'espace de travail et permettent d'étendre R.

4.1 Nombres et Vecteurs

L'objet de base est le vecteur.

4.1.1 Affectation :

L'affectation est faite par l'opérateur `c` (concaténation) ou par `assign ;` les arguments de `c` peuvent être des vecteurs et le résultat est un vecteur ;

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
> x <- assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
> y <- c(x, 0, x)
```

4.1.2 Arithmétique :

Les opérations sur les vecteurs sont exécutées élément par élément ; une valeur constante est répétée afin d'avoir la même longueur que les vecteurs dans une expression ;

- `> v <- 2*x + y + 1` génère un nouveau vecteur de longueur 11 ;
- opérations : `+, −, *, / , ^`
- fonctions :
  - `log, exp, sin, cos, tan, sqrt ;`

- `max, min, range=c(min(x),max(x));`
- `length, sum, prod;`
- `mean, var=sum((x-mean(x))^2)/(length(x)-1)`

4.1.3 Séries de valeurs

Plusieurs façons de générer des séquences régulières :
- opérateur « `:` »
  - `> 1 :30 # = c(1,2,...,30)`
  - priorité `> 2*1 :15 = c(2,4,...,30)`
    `> 1 :n-1 = c(0,1,...,n-1)`
    `> 1 :(n-1) = c(1,2,...,n-1)`
- fonction `seq()` génère des suites générales et prend 5 arguments
  - `from` = valeur initiale
  - `to` = valeur finale `> seq(-5,5)`
  - `by` = valeur de pas `> seq(-5,5,by=.2)`
  - `length` = longueur de suite `> seq(length=51,from=-5,by=.2)`
  - `along` = vecteur
- fonction `rep()` génère des réplications
  - `> rep(x,times=5)`

4.1.4 Valeurs manquantes

Lorsqu'une valeur est manquante ou inconnue, une place peut être réservée en affectant la valeur spéciale `NA`
- `> z <- c(1 :3, NA); ind <- is.na(z)`

4.1.5 Vecteurs de caractères

Ils sont souvent utilisés, par exemple pour les étiquettes sur une tracé ;
- ils peuvent être concaténés par la fonction `c("X", "Y");`
- ils peuvent être collés un par un par la fonction `paste()` qui donne aussi la possibilité de spécifier le séparateur - par exemple
  `> labs <- paste(c("X", "Y"), 1 :10, sep="")`
  produit le vecteur
  `c("X1","Y2","X3","Y4",...,"X9","Y10")`

4.1.6 Vecteurs logiques

Les éléments ont que deux valeurs possibles, `FALSE` et `TRUE` ;
- des vecteurs logiques sont générés par des *conditions* - par exemple
  `> temp <- x > 13`
  donne un vecteur de la même longueur de `x` ayant les valeurs `F` et `T` ;
- les opérateurs logiques sont :
  - `< , > , <= , >=, == , !=`
  - `c1 & c2 , c1 | c2 , !c1`
    où `c1` et `c2` sont des expressions logiques
- lorsqu'un vecteur logique est utilisé dans l'arithmétique ordinaire, `FALSE` devient `0` est `TRUE` devient `1` ;

4.1.7 Indices

Des sous ensembles peuvent être sélectionnés à l'aide d'un *vecteur d'indices* entre crochets - ce vecteur peut prendre 4 formes différentes

- vecteur logique : les valeurs correspondantes à `TRUE` dans le vecteur d'indices sont sélectionnées et celles correspondantes à `FALSE` sont omises - exemples
  - `> y <- x[!is.na(x)]`
    crée un objet `y` qui contient les valeurs non-manquantes de `x` (dans le même ordre)
  - `> (x+1)[(!is.na(x)) & (x>0)] -> z`
    crée un objet `z` qui contient les valeurs de `x+1` qui sont non-manquantes et positives.
- vecteur de valeurs entières positives : le vecteur d'indices peut être de toute longueur et le résultat est de la même longueur que le vecteur d'indices - exemples
  - `> x[6]; x[1 :10]`
    à condition que `length(x)` est inférieure à 10 ;
- vecteur de valeurs entières négatives : spécifie les valeurs à *exclure*
  - `> y <- x[-(1 :5)]`
    affecte à `y` tous les éléments de `x` à part les 5 premiers ;
- vecteur de chaînes de caractères : cette possibilité s'applique seulement lorsque l'objet a un attribut `names` ;

```
> fruit <- c(5, 10, 1, 20)
> names(fruit) <- c("orange", "banane", "pomme", "peche")
> lunch <- fruit[c("pomme","orange")]
```

4.2 Matrices et Tableaux

4.2.1 Tableaux (arrays)

Un *tableau* est une collection d'écritures à sous indices multiples. Un *vecteur de dimension* est un k-vecteur d'entiers positifs dont les valeurs donnent les limites supérieures pour chacune des k dimensions. Un vecteur peut être utilisé par R comme un tableau seulement si son attribut `dim` est défini par un vecteur de dimension. Si `z` est un vecteur de 1500 éléments, l'affectation

```
> dim(z) <- c(3, 5, 100)
```

lui donne l'attribut `dim` qui lui permet d'être traité comme un tableau 3 fois 5 fois 100. Les valeurs dans un tableau de données sont stockées colonne par colonne.

- **indices** : sous indices entre crochets, séparés par des virgules ; on peut utiliser des vecteurs d'indices ; une position d'indice vide implique tout l'étendu ; exemples : `z[2,,]` est un tableau 5 fois 100
- **tableaux d'indices** ... utiles pour des matrices de plans d'expérience
- **la fonction `array()`** permet la construction de tableaux
  - `> Z <- array(vec_données, vec_dim)`
  - exemple : si le vecteur `h` contient 24 éléments, la commande suivante construira un tableau `Z`, 3 par 4 par 2
    `> Z <- array(h, dim(3,4,2))`
  - un tableau de zéros est construit par
    `> Z <- array(0, c(3,4,2))`
- **opérations** sur tableaux :
  - arithmétique est élément par élément à condition que les attributs `dim` sont les mêmes ;
    `> D <- 2*A*B + C + 1`
  - produit externe : tous les produits possibles
    `> ab <- a %o% b`
  - transposé généralisée d'un tableau : la fonction `aperm(a, perm)` peut être utilisée afin de permuter un tableau `a` ; l'argument `perm` doit être une permutation des entiers $\{1, ..., k\}$ où k est le nombre de sous indices dans `a`.

```
> B <- aperm(A, c(2,1))
```
est la transposée de `A` ; on peut utiliser `t(A)` pour ce cas spécial ...

4.2.2 Matrices

Une matrice est simplement un tableau à deux indices.
- **multiplication** : si `A` et `B` sont des matrices de même dimension, `x` est un vecteur
  - `> A * B` est la matrice de produits élément par élément,
  - `> A %*% B` est le produit matriciel,
  - `> x %*% A %*% x` est une forme quadratique.
- **autres opérations** : `t(A), nrow(A), ncol(A), crossprod(X,y)=t(X) %*% y, diag(v), diag(M)`
- **systèmes d'équation**s linéaires et inversion : `solve(A,b)` résout l'équation $Ax = b$; `solve(A)` calcul l'inverse de `A`.
- **valeurs et vecteurs propres** : la fonction `eigen(Sm)` calcul les valeurs et vecteurs propres d'une matrice symétrique `Sm` ; le résultat est une liste de deux composantes nommées `values` et `vectors`
  `> ev <- eigen(Sm); ev$val; ev$vec`
- **SVD** et déterminantes : `> absdet <- function(M) prod(svd(M)$d)`
- **ajustage** par moindres carrés : `>? lsfit`
- **matrices partitionnées** : `rbind()` et `cbind()` forment des matrices ligne-par-ligne ou colonne-par-colonne.
- **concaténation** : transformer une matrice en vecteur par
  `> vec <- c(X)`
- **tables de fréquences** (contingence) : la fonction `table()` calcul des tables de fréquence à partir de facteurs de longueur égale, ordonné par les niveaux de chaque facteur ;
  `> statefr <- table(statefac)`

4.3 Listes et dataframes

Une *liste* est un objet formé d'une collection ordonnée d'objets *composants.* Les composantes peuvent être de types différents. Un *data frame* est une liste ayant la classe « `data.frame` »

4.3.1 Listes

- exemple simple :
  `> Lst <- list(nom="Fred", epouse="Marie", no.enfants=3, enfant.ages=c(4,7,9))`
- les composantes sont toujours *numérotées*
  - si `Lst` est une liste à quatre composantes, on peut les accéder par
    `> Lst[[1]], Lst[[2]], Lst[[3]], Lst[[4]]`
  - si, de plus, `Lst` est un tableau avec vecteur de sous indices, sa première entrée est
    `> Lst[[4]][1]`
- les composantes peuvent être *nommées,* ce qui facilite l'accès par des expressions de la forme
  `> nom$nom_composante`
  et pour l'exemple ci-dessus
  - `Lst$nom = Lst[[1]] = "Fred"`
  - `Lst$epouse = Lst[[2]] = "Marie"`
  - `Lst$enfant.ages[1] = Lst[[4]][1] = 4`
- les noms peuvent être utilisés directement entre les crochets :
  `> Lst[["nom"]] = Lst$nom`
  `> x <- "nom"; Lst[[x]]`

- il est très important de distinguer entre `Lst[[1]]` et `Lst[1]`
  - `[[....]]` est l'opérateur utilisé afin de sélectionner un seul élément d'une liste, **sans** son nom ;
  - `[....]` est l'opérateur général de sous indices et donne une *sous liste* avec son nom.
- le vecteur de noms est un attribut de la liste

4.3.2 Data Frames

Un data frame est une matrice ayant des colonnes de mode et d'attribut (possiblement) différents. La commande `attach()` permet l'utilisation directe des noms de variables sans le syntaxe `nom$nom_composante`.

4.4 Import et export de données

Des grandes bases de données seront lues des fichiers externes. On peut lire un fichier (champs de largeur fixé) avec la fonction `read.fwf()`, mais il est fortement conseillé d'utiliser la fonction `read.table()` qui donne un data frame directement. Une fonction moins sophistiquée est `scan()`.

4.4.1 La fonction `read.table()`

Afin de lire un data frame entier directement, le fichier externe doit avoir une forme spéciale :

- la première ligne comporte un *nom* pour chaque variable ;
- les lignes suivantes comportent chacune une *étiquette de ligne (row label)* comme premier article, puis les valeurs des variables.

Voici un exemple de fichier de données `maisons.data` avec noms et étiquettes.

	Prix	Superficie	Quartier	Pièces	Age	Chauff.centr
01	52.00	111.0	830	5	6.2	non
02	54.75	128.0	710	5	7.5	non
03	57.50	101.0	1000	5	4.2	non
04	57.50	131.0	690	6	8.8	non
05	59.75	93.0	900	5	1.9	oui
...						

La fonction `read.table()` est maintenant utilisée afin de le lire dans un data frame

```
> PrixMaison <- read.table("maisons.data", header=TRUE)
```

Souvent on n'inclut pas les étiquettes de ligne et on utilise les étiquettes par défaut. Dans ce cas, le fichier n'aura pas la première colonne ci-dessus et la lecture est faite par

```
> PrixMaison <- read.table("maisons.data")
```

Afin d'extraire une ligne de valeurs d'une structure (dataframe), on utilise la commande `as.vector`

```
> t <- as.vector(PrixMaison[,3], mode="numeric")
```

4.4.2 La fonction `scan()`

Cette fonction est déconseillée sauf pour la lecture de données directement du clavier. Un retour-chariot après une ligne vide met fin à la saisie.

```
> w0 <- scan()
```

4.4.3 Datasets internes

Plus de 50 datasets sont fournis avec R. Une liste est obtenue par la commande

```
> data()
```

et un dataset est chargé à l'aide de

```
> data(infert)
```

Des informations concernant les données sont dans l'aide

```
> help(infert)
```

Afin de charger des données qui se trouvent dans un autre package, on charge le package puis on utilise `data` :

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

4.4.4 Importation de données

Il existe un package pour l'importation de données en provenance des logiciels SAS, Minitab, SPSS ainsi que des interfaces aux bases de données SQL et ODBC. Voir aussi R-data.pdf.

4.4.5 Exportation de données

Le plus simple est de « copier - coller » directement de la fenêtre de R. Sinon, utiliser
- fonction `cat()` – arguments `file, append`
- fonction `write.table()`
- fonction `write()`

5 Fonctions statistiques

5.1 Densités, distributions, variables aléatoires

Un aspect commode de R est de fournir un ensemble très complet de tables statistiques. Pour chaque loi, on peut à l'aide d'un préfixe,
- évaluer la fonction de répartition $P(X \leq x)$ - `pxxx(q, ...)`
- évaluer la densité de probabilité $f(x)$ - `dxxx(x, ...)`
- évaluer la fonction quantile : pour une valeur q donnée, trouver x la plus petite tel que $P(X \leq x) > q$ - `qxxx(p,...)`
- simuler des variable aléatoires - `rxxx(n, ...)`.

5.1.1 Quelques exemples.

Loi binomiale pour $n = 10$ et $p = 1/3$:

```
> ?dbinom
> dbinom(0 :10,10,1/3)
> options(digits=4)
> dbinom(0 :10,10,1/3)
> dbinom(1,10,1/3) # prob. d'obtenir 1 = 0.0867
```

Quelle est la probabilité d'obtenir plus de 45 et moins de 55 avec une loi binomiale pour $n = 100$ et $p = 1/2$?

Loi	nom	arguments
binomiale	`binom`	`size, prob`
chi-deux	`chisq`	`df, ncp`
exponentielle	`exp`	`rate`
F	`f`	`df1, df2, ncp`
géométrique	`geom`	`prob`
hypergéométrique	`hyper`	`m, n, k`
log-normale	`lnorm`	`meanlog, sdlog`
binomiale négative	`nbinom`	`size, prob`
normale	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
t	`t`	`df, ncp`
uniforme	`unif`	`min, max`

TAB. 1 – Lois de probabilités dans R

```
> sum(dbinom(46 :54,100,1/2)) # 0.6318
```

Quelle est la probabilité d'obtenir plus que 4 pour une loi de Poisson de paramètre $\lambda = 2.7$?

```
> 1 - ppois(4, 2.7) # 0.1371
```

Quelle est la probabilité d'obtenir plus que 1.96 pour une loi normale réduite ?

```
> 1 - pnorm(1.96) # 0.025
```

Quelle est la valeur x telle que $P(X \leq x) = 0.975$ pour une loi normale réduite ?

```
> qnorm(0.975) # 1.96
```

Quel est le quantile 1% pour une loi t à 5 degrés de liberté ?

```
> qt(0.01, 5) # -3.365
```

Simuler un échantillon aléatoire simple de 10 valeurs

- d'une loi de Poisson de paramètre $\lambda = 2.7$
  `> rpois(10, 2.7)`
- d'une loi normale réduite
  `> rnorm(10)`
- d'une loi chi-deux à 2 degrés de liberté
  `> rchisq(10, 2)`
- d'une loi binomiale $n = 100$ et $p = 1/2$
  `> rbinom(10, 100, 0.5)`

5.2 Statistiques de base

- **moyenne, variance, etc**. :
  - `max, min, range`
  - `mean, median`
  - `var, cor`
  - `quantile, IQR`
- **summary** : un résumé très complet de toutes les statistiques de base d'un tableau
- **fivenum** : le résumé à cinq nombres de Tukey : min, quantile 25%, médiane, quantile 75%, max

```
> fivenum(c(rnorm(1000),-1 :1/0))
[1] -Inf   -0.66877   0.07399   0.6916   Inf
> qnorm(.25); qnorm(.75);
[1] -0.67449    0.67449
```

5.2.1 Exemples

```
> x <- c(1,2,3,3,3,4,7,8,9,NA)
```

– lorsque les données contiennent des valeurs manquantes, les fonctions `max(), min(), range(), mean()`, et `median()` rendent `NA`, et les fonctions `var(), cor()`, et `quantile()` renvoient un message d'erreur ;

```
> max(x, na.rm=T)
[1] 9
```

– spécifiant `na.rm=T` dans la fonction `max()` force R d'enlever toute valeur manquante du vecteur `x` et de renvoyer la valeur maximale dans `x` ;

```
> min(x, na.rm=T)
[1] 1

> range(x, na.rm=T)
[1] 1 9

> mean(x, na.rm=T)
[1]  4.444444

> mean(x, trim=0.2, na.rm=T)
[1]  4.285714
```

– l'argument `trim` peut prendre toute valeur entre 0 et 0.5 incluse, à être rogné de chaque extrémité des données ordonnées. Si `trim=0.5`, le résultat est la médiane ;

```
> median(x, na.rm=T)
[1] 3

> quantile(x, probs=c(0,0.1,0.9), na.rm=T)
  0% 10% 90%
   1 1.8 8.2
```

– la fonction `quantile()` renvoie les quantiles de `x` spécifiés dans l'argument `probs`.

Si il n'y a pas de valeurs manquantes dans le vecteur `x`, il n'est pas nécessaire de spécifier `na.rm=T` - utiliser simplement `min(x), max(x)`, etc.

Ces fonctions peuvent être utilisées sur des matrices. Elles ne seront pas appliquées aux lignes ou colonnes individuellement, mais trouveront plutôt le `min, max,` etc. de la matrice entière.

```
> var(x[!is.na(x)])
[1] 8.027778
```

– les valeurs manquantes sont enlevées du vecteur `x` utilisant le sous indice `!is.na(x)`
– spécifiant deux arguments à la function `var(x)`, `var(x,y)` renvoie la *covariance* entre les deux arguments
– les arguments peuvent être des vecteurs ou des matrices

```
> y <- c(1,2,3,4,5,6,7,8,9,10)
> cor(x[!is.na(x)],y[!is.na(x)]
[1] 0.9504597
```

– vu que la fonction `cor()` requiert que `x` et `y` soient de la même longueur, il est nécessaire d'enlever la valeur de `y` qui correspond à la valeur manquante dans `x` ; ceci est obtenu par `y[!is.na(x)]`

boy	A	B
1	13.2 (L)	14.0 (R)
2	8.2 (L)	8.8 (R)
3	10.9 (R)	11.2 (L)
4	14.3 (L)	14.2 (R)
5	10.7 (R)	11.8 (L)
6	6.6 (L)	6.4 (R)
7	9.5 (L)	9.8 (R)
8	10.8 (L)	11.3 (R)
9	8.8 (R)	9.3 (L)
10	13.3 (L)	13.6 (R)

TAB. 2 – Usure de chaussures avec 2 matériaux.

```
> summary(x)
 Min. 1st Qu. Median   Mean   3rd Qu. Max. NA's
  1       3      3     4.444     7     9    1
```

```
> z <- c(5,4,3,2,1,9,8,7,6,5)
> pmax(x,y,z)
 [1]  5  4  3  4  5  9  8  8  9 NA
> pmin(x,y,z)
 [1]  1  2  3  2  1  4  7  7  6 NA
```

– `pmax()` renvoie la valeur maximale pour chaque position dans un nombre de vecteurs
– de même, `pmin()` renvoie la valeur minimale
– `na.rm=T` peut aussi être spécifié afin d'enlever des valeurs manquantes.

5.3 Tests standards

La liste des tests classiques comprend :

binom.test	chisq.test	cor.test	fisher.test
friedman.test	kruskal.test	ks.test	mcnemar.test
prop.test	t.test	var.test	wilcox.test

Ces tests se trouvent dans la bibliothèque `ctest`. Le test le plus répandu pour la moyenne d'une population est le test-t .

5.3.1 Exemples de test-t

La table ci-dessus montre la quantité d'usure dans une expérience de chaussures avec 10 garçons. Il y avaient deux matériaux (A et B) affectés au hasard à la chaussure gauche (L) ou droite (R).

Nous pouvons utiliser ces données afin d'illustrer des tests simples appariés et non appariés.

```
> library(MASS)
> data(shoes)
> shoes   # afficher les données
```

```
> shoes$A
> shoes$B
> par(mfcol(1,2))  # tracer afin de vérifier normalité
> hist(shoes$A)
> hist(shoes$B)
> summary(shoes$A)  # stat's de base
> summary(shoes$B)
> help(t.test)      # test-t
> t.test(shoes$A, mu=10) # test-t, H_0 : mu=10
> t.test(shoes$B, mu=10) # test-t, H_0 : mu=10
> t.test(shoes$A)$conf.int  # intervalle de confiance
> t.test(shoes$B)$conf.int  # intervalle de confiance
> help(wilcox.test)  # test de rang signé
> wilcox.test(shoes$A, mu=10)
> t.test(shoes$A, shoes$B, var.equal=T)  # test-t bi-échantillon
> t.test(shoes$A, shoesB, var.equal=F) # modification de Welch
> t.test(shoes$A, shoes$B, paired=T) # test-t apparié
> wilcox.test(shoes$A, shoes$B, paired=T) # sans hyp˙de normalité
```

6 Graphiques

Les outils graphiques de R sont très importants et versatiles. Il est possible de les utiliser afin d'afficher une large éventail de graphiques statistiques et aussi de construire des types entièrement nouveaux. Les commandes pour tracer des graphiques sont divisées en trois groupes : haut niveau, bas niveau et interactif. De plus R maintien une liste de *paramètres graphiques* qui peuvent être manipulés afin de personnaliser vos graphiques. Essayer la commande

```
> demo(graphics)
```

6.1 Commandes de haut niveau

Ces fonctions génèrent un graphique complet des données passées en argument. Les axes, les étiquettes et les titres sont générés automatiquement.

6.1.1 La fonction `plot()`

Une des fonctions graphiques la plus utilisée est `plot()`. Ceci est une fonction générique : le type de graphique produit dépend du type ou classe de son premier argument.

plot(x,y), plot(xy) si `x` et `y` sont des vecteurs, `plot(x,y)` produit un « scatterplot » de `y` contre `x`. La deuxième forme (un seul argument) envoie une liste à deux éléments ou une matrice à deux colonnes.

plot(x) si `x` est une série temporelle, produit un graphique de `x` contre son indice.

plot(f), plot(f,y) `f` est un objet facteur, `y` est un vecteur numérique ; la première forme génère un graphique en barre (« bar plot ») de `f`, la seconde produit des graphiques en boite (« boxplots ») de `y` pour chaque niveau de `f`.

plot(df), plot(~expr), plot(y~expr) `df` est un data frame, `y` est tout objet, `expr` est une liste d'objets séparés par ' + ' (eg. `a+b+c`) ; les 2 premières formes produisent des tracés distribuées des variables dans le data frame, ou d'un nombre d'objets nommés ; la troisième forme trace `y` contre chaque objet nommé dans `expr`.

6.1.2 Données multi-variables

– Si `X` est une matrice ou data frame numérique, la commande

```
> pairs(X)
```

produit une matrice de scatterplot deux-par-deux des variables définies par des colonnes de `X, c` ce qui donne une matrice de $n(n-1)$ graphiques ;

– Lorsque 3 ou 4 variables sont concernées, un *coplot* est plus instructif. Si `a` et `b c` sont des vecteurs numériques, et `c` est un vecteur numérique ou vecteur de facteurs, alors la commande

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

produit un nombre de scatterplots de `a` contre `b` pour de valeurs données par `c`. Lorsque `c` est un facteur, ceci donne un graphique pour chaque niveau de `c`.

– Les deux fonctions prennent un argument `panel=` qui est utilisé afin de personnaliser le type de tracé dans chaque panneau.

6.1.3 Autres graphiques

D'autres fonctions fournissent des tracés de types différents. Quelques exemples sont :

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

Graphiques de distribution-comparaison. Le premier trace le vecteur numérique `x` contre les « Normal order scores » (graphique de probabilité normale) et le deuxième rajoute une ligne droite passant par la distribution et les quartiles de données. La troisième forme trace des quantiles de `x` contre ceux de `y` afin de comparer leurs distributions respectives.

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

Produit un histogramme du vecteur numérique `x`. Un nombre raisonnable de classes est normalement choisi, mais une recommandation peut être donnée par l'argument `nclass=`. Alternativement, les breakpoints peuvent être spécifiés exactement avec l'argument `breaks=`. Si l'argument `probability=TRUE` est donné, les barres représentent des fréquences relatives à la place de denombrements.

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

Graphiques de trois variables. Le graphique `image` trace une grille de rectangles utilisant des couleurs différentes de représenter la valeur de `z`, le graphique `contour` trace des lignes de niveau (« contour lines ») afin de représenter la valeur de `z`, et le graphique `persp` trace une surface en 3D.

6.1.4 Arguments

Il y a nombreux arguments qui peuvent être passé aux fonctions graphiques haut niveau. Voir la Table 3.

6.2 Commandes de bas niveau

Parfois la commande de haut niveau ne donne pas un résultat satisfaisant. Dans ce cas, des commandes de bas niveau sont utilisées afin de rajouter des informations supplémentaires, comme points, lignes, texte.

Argument	Détails
`add=TRUE`	Force la function d'agir comme une fonction graphique bas niveau, superimposant le graphique sur le graphique courant.
`axes=FALSE`	Supprime la génération des axes – utile pour rajouter des axes personalisés avec la fonction `axis()`. La valeur par défaut, `axes=TRUE`, inclut des axes.

`log="x"`
`log="y"`
`log="xy"`
	Produit des axes logarithmiques en `x, y` ou les deux.

`type=`	L'argument `type=` contrôle le type de graphique produit : `type="p"` trace des points individuels (par défaut) `type="l"` trace des lignes `type="b"` trace des points connectés par des lignes (both) `type="o"` trace des points recouvrés par des lignes (overlayed) `type="h"` trace des lignes verticales à partir des points vers l'axe zéro (high-density) `type="s"` `type="S"` Tracés de Step-function – point de marches défini par le haut et le bas respectivement. `type="n"` Pas de tracé (none). Néanmoins les axes sont toujours dessinés. Idéal pour créer des graphiques avec des fonctions bas-niveau.
`xlab=string` `ylab=string`	Etiquettes pour des axes `x` et `y`. Utiliser ces arguments afin de changer les etiquettes par défaut.
`xlim=c(xmin,xmax)` `ylim=...`	Limites pour les axes.
`main=string`	Titre de figure, placé en haut de graphique, grande police de caractères.
`sub=string`	Sous-titre, placé juste en dessous d'axe `x` , plus petite police.

TAB. 3 – Arguments graphiques

Argument	Détails
`points(x, y)` `lines(x, y)`	Rajoute points ou lignes au graphique courant. L'argument `type=` de `plot()` peut aussi être envoyé à ces fonctions (par défault "`p`" pour `points()` et "`l`" pour `lines()`.)
`text(x, y, labels, ...)`	Rajoute texte aux points donnés par `x, y`. Normalement `labels` est un vecteur entier ou caractère en quel cas `labels[i]` est tracé au point `(x[i], y[i])`. **Note :** cette fonction est souvent utilisée dans le séquence `> plot(x, y, type="n"); text(x, y, names)` Le paramètre graphique `type="n"` supprime les points, et la fonction `text()` fournit des caractères speciales, comme specifié par le vecteur de caractères `names` pour les points.
`abline(a, b)` `abline(h=y)` `abline(v=x)` `abline(lm.obj)`	Rajoute une ligne de pente `b` et intercepte `a` au graphique courant. `h=y` peut specifier des coordonées y pour les hauteurs des lignes horizontales traversant le graphique, et `v=x` pour les coordonées x des lignes verticales. Aussi `lm.obj` peut être une liste avec des coefficients qui resultent des fonctions d'ajustage de modèles.functions, et qui sont pris comme intercepte et pente, dans cet ordre.
`polygon(x, y, ...)`	Dessine un polygone défini par des sommets ordonnés dans `(x, y)` et (facultativement) remplit avec lignes ou couleurs.
`legend(x, y, legend, ...)`	Rajoute une légende dans la position spécifiée. Caractères de trace, styles de ligne, couleurs etc., sont identifiés avec les étiquettes dans le vecteur de carctères `legend`. Au moins un argument supplementaire `v` (un vecteur de la même taille que `legend`) avec les valeurs correspondantes valeurs de l'unité graphique, doit aussi être fournie comme suit : `legend( , fill=v)` Couleurs pour boites remplies `legend( , col=v)` Couleurs des points ou lignes `legend( , lty=v)` Line styles `legend( , lwd=v)` Line widths `legend( , pch=v)` Caractères (vecteur) de trace (plotting characters)
`title(main, sub)`	Titre de figure, placé en haut de graphique avec une grande police de caractères et (facultativement) un sous-titre, placé juste en dessous d'axe `x` dans une plus petite police.
`axis(side, ...)`	Rajoute un axe au graphique courant sur le coté donné par le premiert argument (1 à 4, contre le sens de montre à parit du bas.) Auitres arguments contrôlent le positionnement de l'axe dans ou à coté du graphique, et positions de tick et étiquettes. Utile après un appel `plot()` avec argument `axes=FALSE` .

TAB. 4 – Commandes graphiques

6.2.1 Les commandes

Les fonctions de bas-niveau les plus utiles sont détaillées dans la Table 4. Ces fonctions ont normalement besoin d'information de positionnement (e.g., coordonnés x et y) afin de déterminer où placer les nouveaux éléments. Les coordonnés sont données en termes de coordonnées utilisateur qui sont définies par la précédente commande graphique de haut-niveau. Des fonctions tel que `locator()` (voir ci-dessous) peuvent être utilisées afin de spécifier des positions interactivement.

6.2.2 Notation mathématique

Dans certains cas, il est utile de rajouter des symboles mathématiques et des formules aux graphiques. Dans R on peut spécifier une expression plutôt qu'une chaîne de caractères dans `text, mtext, axis,` ou `title.`

Par exemple, le code suivant dessine la formule pour la fonction de probabilité binomiale :

```
> text(x,y,expression(paste(bgroup("(", atop(n, x), ")"),
        p^x, q^{n-x})))
```

Pour plus d'information, y compris une liste complète de possibilités, utiliser l'aide

```
> help(plotmath)
> example(plotmath)
```

6.3 Interaction avec des graphiques

R fournit aussi des fonctions permettant aux utilisateurs d'extraire ou de rajouter de l'information à un graphique à l'aide de la souris.

`locator(n, type)`

Attend que l'utilisateur sélectionne des locations sur le graphique courant à l'aide du bouton *gauche* de la souris. Ceci continue jusqu'à ce que `n` points sont choisis, ou un autre bouton est enfoncé. L'argument `type` permet de tracer au points sélectionnés. Par défaut, il n'y a pas de trace. `locator()` renvoie les locations dans une liste avec deux composantes x et y . Normalement `locator()` est appelée sans arguments. Elle est particulièrement utile pour sélectionner interactivement des positions pour des légendes ou étiquettes. Par exemple, afin de placer un texte d'information prés d'une valeur exceptionnelle, la commande

```
> text(locator(1), "Outlier", adj=0)
```

pourrait être utile.

`identify(x, y, labels)`

Permet de souligner des points définis par `x` et `y` (avec le bouton gauche) en traçant la composante correspondant de `labels` à proximité. Renvoie l'indice en appuyant sur le bouton de milieu.

Afin d'identifier des points particuliers, plutôt que leurs positions, on utilisera :

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

6.4 Paramètres graphiques

R maintien un très grand nombre de paramètres graphiques qui contrôlent le style de ligne, couleurs, disposition de figures, justification de texte, etc. Chaque paramètre a un *nom* et une *valeur.* Les paramètres graphiques peuvent être réglés de manière

- permanente, ainsi modifiant toutes les fonctions graphiques
- temporaire, ainsi modifiant un seul appel à une fonction graphique.

6.4.1 Changements permanents : la fonction `par()`

La fonction `par()` est utilisée afin d'accéder et de modifier la liste de paramètres graphiques.

`par()`	sans arguments, renvoie une liste de tous les paramètres graphiques et leurs valeurs
`par(c("col", "lty"))`	avec un argument de vecteur de caractères, renvoie seulement les paramètres nommés
`par(col=4, lty=2)`	avec arguments nommés, pose les valeurs des paramètres nommés

Fixer les paramètres graphiques avec la fonction `par()` change leurs valeurs à titre définitif, dans le sens que tous les appels futurs aux fonctions graphiques seront affectés par la nouvelle valeur. Afin de restaurer les valeurs initiales, il faut sauver le résultat de `par()` d'abord.

```
> oldpar <- par(col=4, lty=2) # sauver
  ...  commands graphiques ...
> par(oldpar) # restaurer
```

6.4.2 Changements temporaires

Des paramètres graphiques peuvent être passés aux fonctions comme arguments nommés. Dans ce cas, les changements durent que pendant l'appel de fonction. Par exemple,

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

produit un scatterplot utilisant une signe plus comme caractère graphique, sans changer les graphiques futurs.

6.5 Liste de paramètres graphiques

Ici nous allons détailler des paramètres graphiques les plus utilisés. La documentation d'aide pour `par()` est exhaustif mais concis.

6.5.1 Éléments graphiques

Ce sont des paramètres qui contrôlent comment les éléments graphiques (points, lignes, texte, polygones) sont dessinés - voir la Table 5.

6.5.2 Axes et tick marks

Les axes ont trois composantes principales : la ligne (style contrôlé par le paramètre graphique `lty`), les « tick marks » (qui marquent les divisions unitaires le long de la ligne d'axe) et les « tick labels » (qui marquent des unités.) Ces composantes peuvent être personnalisées avec les paramètres graphiques suivants :

- `lab = c(nx, ny, longueur)` : nombre d'intervalles tick en x et y, longueur en caractères
- `las = valeur` : orientation d'étiquettes d'axe - 0=parallèle, 1=horizontale, 2=perpendiculaire à l'axe

Nom et valeur	Déscription
`pch="+"`	Caractère utilisé pour tracer des points. Par défat, un cercle. Le caractère "." produit des points centrés.
`pch=4`	Chaque valeur donne un symbole graphique spécial. Afin de les voir, utiliser `> legend(locator(1), as.character(0 :18), marks=0 :18)`
`lty=2`	Types de ligne. Type 1 est solide, à partir de 2 ils sont pointillés ou hachés.
`lwd=2`	Largeurs de ligne (width).
`col=2`	Couleurs à utiliser pour points, lignes, régions remplies et images.
`font=2`	Un entier qui spécifie la police (font). 1 = ordinaire, 2 = **gras**, 3 = *italique* et 4 = ***italique gras.***

`font.axis`
`font.lab`
`font.main`
`font.sub`
	Les polices à utiliser pour les axes, étiquettes, titres et sous-titres.

`adj=-0.1`	Justification de texte relative au position de plotting. 0 = gauche, 1 = droite, 0.5 = centrer. La valeur actuelle est la proportion de texte à gauche de la position, donc -0.1 laisse un espace de 10% de la largeur du texte entre le texte et la position
`cex=1.5`	Expansion de caractères. Valeur désirée de la taille de texte relative à la taille par défaut.

TAB. 5 – Paramètres graphiques

– `xaxs="s", yaxs="d" :` styles pour axes - s=standard, e=étendu, r=range (par défaut), d=directe (vérouille axe courant pour futurs axes)

6.5.3 Figures multiples

Dans R vous pouvez créer un tableau *n* par *m* de figures sur une seule page. Chaque figure a ses propres marges, et le tableau de figures est entouré par une marge extérieure.

Les paramètres graphiques pour des figures multiples sont détaillés dans la Table 6

Des marges extérieures sont utiles pour des titres pleine page, etc. Des dispositions plus complexes peuvent être produites par des fonctions `split.screen(), layout()`.

Nom et valeur	Déscription
`mfcol=c(3,2)` `mfrow=c(2,4)`	Dimensions du tableau de figures multiples : nombre de lignes, nombre de colonnes. `mfcol` remplit colonne par colonne, `mfrow` remplit par lignes.
`mfg=c(2,2,3,2)`	Position de figure en cours : ligne, colonne de figure en cours, nombre de liognes, colonnes dans le tableau.
`fig=c(4,9,1,4)/10`	Position de figure en cours sur la page : bord gauche, droit, bas, haut comme pourcentage de la page mesuré du coin inferieur, gauche.
`oma=c(2,0,3,0)` `omi=c(0,0,0.8,0)`	Taille des marges extérieures.

TAB. 6 – Paramètres graphiques pour figures multiples

6.6 Sortie graphique

R peut générer des graphiques sur des périphériques différents, mais il faut préciser lequel. La liste de possibilités est obtenue par

```
> help(device)
```

Ensuite, la commande

```
> postscript()
```

cause tous les graphiques futurs d'être écrits en format de Postscript. Autres périphériques utiles sont `bmp(), png(), jpeg(), pdf()`.

Un exemple de postscript et de png pour l'inclusion dans un document est :

```
> postscript("plot1.eps", horizontal=FALSE, onefile=FALSE,
             height=8, width=6, pointsize=10)
> png(width=480, height=480, pointsize=12)
```

7 Programmation et Fonctions

R est un langage d'expressions - le seule type de commande est une fonction ou une expression qui renvoie un résultat. Même une affectation est une expression dont le résultat est la valeur affectée. Les commandes peuvent être regroupées dans des accolades, `{expr_1; ...; expr_m}` et la valeur du groupe est le résultat de la dernière expression évaluée.

7.1 Boucles et conditions

7.1.1 Exécution conditionnelle : `if`

– la construction de base est

```
> if (expr_1) expr_2 else expr_3
```

où *expr_1* doit donner une valeur logique ;

– les opérateurs de raccourci `&&` et `||` sont souvent utilisés comme la condition ; ils s'appliquent aux vecteurs de longueur 1 et ils évaluent leur deuxième argument si nécessaire ;

– version vectorisée : `ifelse(condition, a, b)` renvoie un vecteur de longueur de son argument le plus long avec éléments `a[i]` si `condition[i]` est vraie, sinon `b[i]`.

– exemples :

```
> if ( any(x) <= 0 ) y <- log(1+x) else y <- log(x)
> y <- if ( any(x) <= 0 ) log(1+x) else log(x)
> ####
> x <- c(6 :-4)
> sqrt(x) # donne des avertissements
> sqrt(ifelse(x >= 0, x, NA)) # propre !
```

7.1.2 Boucles

R possède 3 commandes pour la construction de boucles répétitives. Ce sont `for`, `while` et `repeat`. Les commandes `next` et `break` donnent contrôle supplémentaire sur l'évaluation. Il existe aussi d'autres fonctions pour boucles implicites comme `tapply`, `apply` et `lapply`. De plus, nombreuses opérations arithmétique sont vectorisées déjà.

La commande `break` cause la sortie immédiate de la boucle intérieure qui est en train de s'exécuter. La commande `next` renvoie l'exécution au début de la boucle et l'itération suivante est poursuivie.

for boucle classique de la forme

```
> for (nom in expr_1) expr_2
```

où *nom* est la variable de boucle, *expr_1* est une expression vectorielle (souvent une série comme `1 :20`), et *expr_2* est une expression groupée avec ses sous-expressions formulées en termes de la variable interne *nom*. L'*expr_2* est évaluée de façon répétée pendant que *nom* prend toutes les valeurs de l'*expr_1*. Exemple simple :

```
> for (i in 1 :5) print(1 :i)
```

Exemple plus intéressante : afin de produire des graphiques séparés de `y` contre `x` pour chaque indicateur de classe dans un vecteur `ind` :

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1 :length(yc)) {
     plot(xc[[i]], yc[[i]]);
     abline(lsfit(xc[[i]], yc[[i]]))
    }
```

repeat évaluation répétée jusqu'à ce qu'un `break` est rencontré - syntaxe

```
> repeat bloc_expr
```

while tant qu'une condition est remplie, l'exécution continue - syntaxe

```
> while (expr_1) expr_2
```

switch renvoie l'exécution à une parmi plusieurs options - syntaxe

```
> switch (expr_1, liste)
```

D'abord, *expr_1* est évaluée et le résultat *valeur* est obtenu. Si *valeur* est un nombre entre 1 et la longueur de *liste*, alors l'élément correspondant de *liste* est évalué et le résultat renvoyé. Si *valeur* est trop grande ou trop petite, `NULL` est renvoyé.

Voici quelques exemples de cette fonction très utile.

```
> x <- 3
> switch(x, 2+2, mean(1 :10), rnorm(5))
[1] 2.2903605 2.3271663 -0.7060073 1.3622045 -0.2892720
> switch(2, 2+2, mean(1 :10), rnorm(5))
[1] 5.5
> switch(6, 2+2, mean(1 :10), rnorm(5))
NULL
```

Si `valeur` est un vecteur caractère alors l'élément ayant un nom qui accord avec *valeur* est évalué. Si il n'y a pas d'accord, `NULL` est renvoyé.

```
> y <- "fruit"
> switch(y, fruit = "banane", legume = "broccoli", viande = "boeuf")
[1] banane
```

Une utilisation répandue de `switch` est de brancher selon la valeur caractère d'un des arguments vers une fonction.

```
> centre <- function(x, type) {
+ switch(type,
+ mean = mean(x),
+ median = median(x),
+ trimmed = mean(x, trim = .1))
+ }
> x <- rcauchy(10)
> centre(x, "mean")
[1] 0.8760325
> centre(x, "median")
[1] 0.5360891
> centre(x, "trimmed")
[1] 0.6086504
```

`switch` renvoie ou la valeur de l'expression évaluée ou `NULL` si aucune expression a été évaluée.

Afin de choisir d'une liste d'alternatives qui existe déjà, `switch` est peut être pas la meilleure façon de faire. Il est souvent mieux d'utiliser `eval` et l'opérateur de sous ensemble, `[[`, directement par `eval(x[[condition]])`.

7.2 Fonctions simples

Comme nous avons vu, R permet l'utilisateur de créer des objets de mode *function*. Ce sont des vraies fonctions de R et peuvent être réutilisées. La plupart de fonctions qui font partie de R, comme `mean(), var(), postscript(),` etc., sont elles même écrites en R.

Une fonction est définie par une affectation de la forme

```
> nom <- function(arg_1, arg_2, ...) expression
```

- *expression* est une expression R, d'habitude groupée, qui utilise
- *arg_1, arg_2, ...* comme arguments afin de calculer une valeur
- l'appel est de la forme

```
> toto <- nom(expr_1, expr_2, ...)
```

- la fonction/programme est sauvé dans un fichier texte avec l'extension « `.R` »

7.2.1 Exemples simples

Un premier exemple trivial calcul la puissance deux d'une valeur :

```
> fdeux <- function(x){x^2}
> fdeux(3)
[1] 9
> is.function(fdeux)
[1] TRUE
```

Un deuxième exemple calcul la statistique-t bi-échantillon.

```
> twosam <- function(y1, y2) {
    n1  <- length(y1); n2 <- length(y2)
    yb1 <- mean(y1);  yb2 <- mean(y2)
    s1  <- var(y1);    s2 <- var(y2)
    s   <- ((n1-1)*s1 + (n2-1)*s2) / (n1+n2-2)
    tst <- (yb1 - yb2) / sqrt(s*(1/n1 + 1/n2))
    tst
  }
> # appel :
> tstat <- twosam(data$male, data$femelle); tstat
```

7.2.2 Arguments nommés

Si les arguments d'une fonction sont données dans la forme « *nom=objet* », alors ils peuvent être données dans n'importe quel ordre. De plus la séquence d'arguments peut commencer sans noms, puis continuer avec.

Si une fonction `fct1` est définie par

```
> fct1 <- function(data, data.frame, graphe, limite) {
  ...
  }
```

alors la fonction peut être invoquée dans plusieurs manières :

```
> ans <- fct1(d, df, TRUE, 20) # sans noms
> ans <- fct1(d, df, graphe=TRUE, limite=20) # avec noms et ordre
> ans <- fct1(data=d, limite=20, graphe=TRUE, data.frame=df) # sans ordre
```

Nous donnons d'habitude des valeurs par défaut à certains arguments qui peuvent alors être omis de l'appel. Si `fct1` est définie comme suit

```
> fct1 <- function(data, data.frame, graphe=TRUE, limite=20) {
  ...
  }
```

elle peut être appelée par

```
> ans <- fct1(d, df)
> ans <- fct1(d, df, limit=10)
```

Les valeurs par défaut peuvent être des expressions arbitraires et pas forcement des constantes.

7.2.3 Scope

Toutes les affectations et les variables dans le corps d'une fonction sont *locales.*

8 Exemple - Galapagos

Considérons un exemple concernant le nombre d'espèces de tortue sur des îles Galapagos. Il y a 30 cas (îles), et 7 variables dans les données. Nous commençons par charger les données et les examiner,

```
> library(faraway)
> data(gala) / gala <- read.table("gala.data")
> gala
```

Les variables sont

Species Le nombre d'espèces de tortue trouvé sur l'île.

Endemic Le nombre d'espèces endémiques.

Elevation L'hauteur de l'île (m).

Nearest La distance de l'île la plus proche (km).

Scruz La distance de l'île de Santa Cruz (km).

Adjacent La superficie de l'île la plus proche.

```
> # statistiques de base
> dim(gala)      # les dimensions
> summary(gala)  # resumé statistique
> gala$Species   # calculs séparés
> mean(gala$Sp)
> median(gala$Sp)
> min(gala$Sp)
> range(gala$Sp)
> quantile(gala$Sp)
> var(gala$Sp)   # variance
> sqrt(var(gala$Sp)) # écart type
> cor(gala)      # toutes les corrélations
> round(cor(gala),3) # plus propre ...
> gala$En        # espèces endémiques
> stem(gala$En)  # graphique tige et feuilles
> # histogrammes et graphiques en boîte
> hist(gala$Sp)
> hist(gala$Sp, main="Histogrammes des espèces", xlab="Nombre d'espèces")
> # nuages des points (scatterplots)
> plot(gala$Area, gala$Sp)
> plot(log(gala$Area),gala$Sp,xlab="log(Area)",ylab="Espéces")
> # matrice de nuages des points
> plot(gala)
> # plusieurs graphiques
> par(mfrow=c(2,2))
> boxplot(gala$Ar)
> boxplot(gala$Adj)
> boxplot(gala$Elev)
> boxplot(gala$SC)
> par(mfrow=c(1,1))
> # selection de sous-ensembles
> gala[2,]     # deuxième ligne
> gala[,3]     # troisième colonne
> gala[2,3]    # élément 2,3
> gala[c(1,4,8),] # lignes 1, 4 et 8
> gala[3 :11,]  # lignes 3 à 11
> gala[,-c(1,2)] # tout sauf colonnes 1 et 2
> gala[gala$Area > 500,] # critère de selection
```

Nous essayons une régression linéaire simple du nombre d'espèces en fonction des 4 dernières variables.

```
> help(lm)
> gfit <- lm(Species ~ Area + Elevation + Nearest + Scruz + Adjacent, data=ga
> summary(gfit)
```