

Une introduction à R

Notes sur R: Un Environnement pour l'Analyse des Données
Version 0.4

R Development Core Team

Copyright © 1990, 1992 W. Venables
Copyright © 1997, R. Gentleman & R. Ihaka
Copyright © 1997, 1998 M. Mächler
Copyright © 1999, 2000 R Development Core Team
Copyright © 2000, M. Ros

Copier et distribuer des copies de ce manuel est autorisé à condition que cette permission et le copyright soient distribués sur toutes les copies.

Copier et distribuer des versions modifiées de ce manuel est autorisé à condition que la totalité du travail résultant soit distribuée avec une permission identique à celle-ci.

Copier et distribuer des traductions de ce manuel est autorisé aux conditions données ci-dessus pour les versions modifiées avec, de plus, l'approbation du R Development Core Team pour la traduction de la présente permission.

Table des Matieres

Préface	1
1 Introduction et préliminaires	2
1.1 L'environnement R	2
1.2 Logiciels apparentés à R et documentation	2
1.3 R et les statistiques	2
1.4 R et le gestionnaire de fenêtres	3
1.5 Utiliser R de façon interactive	3
1.6 Une session d'introduction	4
1.7 Accéder à l'aide	4
1.8 Les commandes de R; la casse etc.	4
1.9 Rappel et correction de commandes	5
1.10 Excuter des commandes depuis un fichier; Rediriger des sorties vers un fichier	5
1.11 Durée de vie des données; Effacer de objets	5
2 Manipulations de base; nombres et vecteurs	7
2.1 Vecteurs et affectation	7
2.2 Opérations sur les vecteurs	8
2.3 Générer des séquences de nombres	9
2.4 Vecteurs logiques	9
2.5 Valeurs manquantes	10
2.6 Vecteurs de caractères	10
2.7 Vecteurs indice; sélectionner et modifier des parties d'un jeu de données	11
2.8 Autres types d'objets	12
3 Objets; leurs modes et attributs	13
3.1 Attributs intrinsèques : mode et longueur	13
3.2 Changer la longueur d'un objet	14
3.3 Obtenir et modifier des attributs	14
3.4 La classe d'un objet	15
4 Listes et data frames	16
4.1 Listes	16
4.2 Construire et modifier des listes	17
4.2.1 Concatèner des listes	17
4.3 Data frames	17
4.3.1 Construire des data frames	17
4.3.2 attach() et detach()	18

4.3.3	Travailler avec les data frames	19
4.3.4	Attacher des listes	19
4.3.5	Gérer le search path	19

Appendice A	Une session d'introduction à R	20
Appendice B	Index des variables et fonctions	24
Appendice C	Index des concepts	26
Appendice D	Références	27

Préface

Ce document est la traduction française de **An introduction to R**.

Cette traduction est actuellement en cours néanmoins, si vous prenez le temps de lire ces pages et de m'envoyer un commentaire (mathieuros@bigfoot.com), je vous en serais très reconnaissant.

Les tournures et les choix pour la traduction du vocabulaire spécifique à R sont sujets à discussion : toutes les propositions ou remarques sont les bienvenues.

J'essaie d'ores et déjà de structurer le document, ne vous étonnez donc pas de trouver vides certains chapitres et sous-chapitres.

Mathieu Ros
Juin, 2000.

1 Introduction et préliminaires

1.1 L'environnement R

R est un système logiciel pour la manipulation de données, les calculs et la représentation graphique. Entre autres choses, il offre

- la possibilité de stocker et manipuler des données de façon fiable et efficace.
- une grande variété de fonctions pour l'analyse des données.
- une suite d'opérateurs pour les calculs sur les tableaux et matrices.
- la possibilité de faire des représentations graphiques pour l'analyse statistique.
- un langage de programmation (S) développé, simple et efficace qui inclut conditions, boucles, fonctions récursives, entrées sorties. La plupart des fonctions sont d'ailleurs elles-même écrites en S.

On définit R comme un environnement car, tout en restant cohérent, il est avant tout extensible et modifiable. R n'est pas une collection figée d'outils, comme c'est le cas de beaucoup de logiciels statistiques.

R doit plutôt être vu comme un outil pour développer de nouvelles méthodes d'analyse de données. De ce fait, il est dynamique et les nouvelles versions ne sont pas toujours complètement compatibles avec les anciennes. Certains utilisateurs accueillent favorablement ces changements à cause du gain et nouvelles méthodes et techniques apportées par les nouvelles versions. D'autres sont plus inquiets du fait que certains codes sont à revoir. R étant surtout un langage de programmation, on doit considérer la plupart des programmes que l'on écrit comme éphémères.

1.2 Logiciels apparentés à R et documentation

On peut voir en R une re-implémentation du langage S développé à AT&T par Rick Becker, John Chambers et Allan Wilks. De nombreux livres et manuels traitant de S sont également appropriés pour R.

La référence est *The New S Language: A Programming Environment for Data Analysis and Graphics* par Richard A. Becker, John M. Chambers et Allan R. Wilks. Les nouveautés apparues avec la version de S de 1991 (Sversion 3) sont traitées dans *Statistical Models in S* édité par John M. Chambers et Trevor J. Hastie. Voir Appendice D [References], page 27, pour des références plus précises.

1.3 R et les statistiques

On ne parle pas des *statistiques* dans l'introduction à R alors que de nombreuses personnes l'utilisent comme un logiciel statistique. Nous préférons voir en R un environnement dans lequel de nombreuses techniques statistiques, classiques et modernes, ont été implémentées. Certaines sont intégrées à l'environnement R de base mais beaucoup sont disponibles sous forme de *packages* (la distinction est plus historique qu'autre chose). Il y a 8 packages (ou bibliothèques) fournis avec R (et appelés packages standards) et beaucoup d'autres sont disponibles sur l'un des sites du CRAN (via <http://www.r-project.org>). la

plupart des statistiques classiques et des dernières techniques sont disponibles dans R mais les utilisateurs seront parfois amenés à faire quelques efforts pour les trouver.

Il y a une importante différence de philosophie entre S (et donc R) et les principaux autres systèmes statistiques. En S une analyse statistique se fait en une série d'étapes avec stockage des résultats intermédiaires dans des objets. SAS et SPSS donneront de copieuses sorties pour une régression ou une analyse discriminante alors que R donnera un minimum de sorties et stockera les résultats dans un objet pour consultation ultérieure par d'autres fonctions.

1.4 R et le gestionnaire de fenêtres

La façon la plus agréable d'utiliser R est sur une machine avec gestionnaire de fenêtres. On se référera en particulier à l'usage de R sur un système X window, même si la plupart de ce qui est dit s'applique à toutes les implémentations de l'environnement R. Les utilisateurs peuvent, de temps en temps, vouloir interagir directement depuis R avec le système d'exploitation. Dans ce manuel, on se regardera principalement les interactions avec le système UNIX. Si vous faites tourner R sous MS-Windows, vous devrez faire quelques ajustements.

1.5 Utiliser R de façon interactive

R affiche un prompt lorsqu'il attend des commandes. Le prompt par défaut est '>', ce qui peut-être la même chose que votre shell prompt sous UNIX. Cependant, comme nous le verrons plus loin, il est facile de changer de prompt si cela vous gêne. On considèrera par la suite que le shell prompt est '\$'. Quand on est sous UNIX, la procédure à suivre lors de la première utilisation est la suivante :

1. Créez un sous répertoire de travail pour stocker les fichiers relatifs à un problème que vous souhaitez traiter avec R :

```
$ mkdir travail
$ cd travail
```

2. Démarrez R avec

```
$ R
```

3. Vous pouvez maintenant taper des commandes R.

4. Pour quitter, tapez

```
> q()
```

Il vous est alors demandé si vous souhaitez ou non sauver les données de la session. Vous pouvez alors choisir de répondre *yes*, *no* ou *cancel* (la première lettre suffit) pour sauver les données avant de quitter, quitter sans sauver ou revenir à la session R.

Les sessions qui suivent sont sur le même modèle :

1. Se placer dans le répertoire de travail et démarrer le programme comme précédemment

```
$ cd travail
$ R
```

2. Utiliser R en terminant par `q()` à la fin de la session.

Sous MS-Windows, la procédure est grosso modo la même. Vous devez créer un répertoire de travail et mettre son chemin dans le champ ‘start in’ (des propriétés) du raccourci. Lancez alors R en double cliquant sur l’icone.

1.6 Une session d’introduction

Si vous souhaitez tester un peu les possibilités de R avant de continuer, vous pouvez vous reporter à la session d’introduction, voir Appendice A [Une session introductive], page 20.

1.7 Accéder à l’aide

R possède un système d’aide interne similaire au `man` d’UNIX. pour obtenir des informations sur une fonction, par exemple `solve`, la commande est :

```
> help(solve)
```

ou

```
> ?solve
```

Pour une fonctionnalité définie par des caractères spéciaux, l’argument doit être entouré de guillemets, afin d’en faire une chaîne de caractères.

```
> help("[[")
```

Les deux formes de guillemets (simples et doubles) peuvent être utilisés ici; on préférera les doubles, mais ce n’est qu’une convention.

Sur la plupart des versions, l’aide est disponible au format HTML et

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

lancera un butineur (netscape sous UNIX) qui permet de naviguer à travers les pages d’aide grâce à des liens.

1.8 Les commandes de R; la casse etc.

Techniquement, R est un *langage de commandes* (interprété) avec une syntaxe assez simple. Il est *sensible à la casse* comme de nombreux programmes basés sur UNIX. Ainsi `A` et `a` sont des symboles différents et ils identifient des variables différentes.

Les commandes de base consistent soit en *expressions* soit en *allocations*. Si une expression est tapée comme commande, elle est évaluée, affichée, et la valeur retournée est perdue. Une allocation évaluée également une expression mais passe sa valeur à une variable sans automatiquement afficher le résultat.

Les commandes sont séparées soit par des points virgules (;), soit par des sauts de ligne. Les *commentaires* peuvent être mis a peu près n’importe où¹, en les précédant d’une dièse (#) : tout ce qui suivra jusqu’à la fin de la ligne sera un commentaire. Si une commande n’est pas complète à la fin de la ligne, R affichera un prompt différent, par défaut

```
+
```

sur la seconde ligne et les suivantes. Il continuera à lire les entrées jusqu’à ce que la commande soit syntaxiquement complète. Ce prompt peut être modifié par l’utilisateur. On l’omettra généralement par la suite, signalant la continuation d’une commande par l’indentation de ses lignes.

¹ pas dans des chaînes ni dans la définition de la liste des arguments d’une fonction

1.9 Rappel et correction de commandes

Sous la plupart des versions UNIX et MS-Windows, R fournit un mécanisme de rappel et de re-exécution des commandes. Les flèches verticales du clavier permettent de parcourir un *historique des commandes*. Quand une commande est ainsi rappelée, on peut y faire circuler le curseur avec les flèches horizontales afin d'effacer (avec la touche `DEL`) ou d'ajouter des caractères. Plus de détails sont donnés par la suite, voir `<undefined>` [L'éditeur en ligne de commande], page `<undefined>`.

Les possibilités de rappel et d'édition sont hautement personnalisables sous UNIX. Pour apprendre comment procéder, consultez la page man de la bibliothèque **readline**.

D'autre part, l'éditeur de texte **emacs** propose des possibilités très intéressantes pour travailler interactivement avec R (via "**ESS**", *Emacs Speaks Statistics*). Voir section "R and Emacs" dans *The R statistical system FAQ*.

1.10 Excuter des commandes depuis un fichier; Rediriger des sorties vers un fichier

On peut exécuter des commandes stockées dans un fichier 'commands.R' placé dans le répertoire de travail 'work' à tout moment pendant la session R :

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

Pour MS-Windows, **source** est disponible dans le menu **File**. La fonction **sink**,

```
> sink("record.lis")
```

enverra les sorties qui suivent non plus sur la console mais dans un fichier, 'record.lis'. La commande

```
> sink()
```

termine ce mode et permet que les sorties soient de nouveau envoyées vers la console.

1.11 Durée de vie des données; Effacer de objets

On appelle *objets* les entités créées et manipulées avec R. Ceux-ci peuvent être des variables, des tableaux de nombres, des chaînes de caractères, des fonctions ou, plus généralement des structures construites avec à partir de ces composants.

Pendant une session R, les objets sont créés et rangés par nom (on reparlera de ce mécanisme dans le prochain chapitre). La commande

```
> objects()
```

(ou **ls**) sert à afficher les noms des objets stockés dans R sur le moment. La collection d'objets stockés couramment est appelée *workspace*.

Pour effacer des objets on utilise la fonction **rm** :

```
> rm(x, y, z, ink, junk, temp, foo, bar)
```

Tous les objets créés pendant une session R peuvent être sauvés dans un fichier pour une réutilisation dans une session ultérieure. A la fin de chaque session, R vous demande si vous désirez sauver tous les objets courants. Si vous choisissez de le faire, les objets seront écrits dans un fichier appelé '.RData'² du répertoire courant.

² le "point" rends le fichier *invisible* lors d'un listing normal sous UNIX

Quand on démarrera R plus tard, il rechargera le workspace contenu dans ce fichier ainsi que l'historique des commandes correspondant.

Il est recommandé d'utiliser des répertoires différents pour les différentes analyses faites avec R. Il est en effet assez fréquent que des objets nommés `x` ou `y` soient créés durant une analyse. De tels noms peuvent avoir un sens pour un problème particulier, mais il peut être difficile de s'y retrouver quand plusieurs problèmes sont menés à bien dans le même répertoire.

2 Manipulations de base; nombres et vecteurs

2.1 Vecteurs et affectation

R opère sur des *structures de données*. La structure de données la plus simple est le *vecteur* numérique, qui est une entité constituée d'une collection ordonnée de chiffres. Pour construire un vecteur **x** constitué des 5 chiffres 10.4, 5.6, 3.1, 6.4 et 21.7 on utilise la commande :

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

c'est une *affectation* utilisant ici la *fonction* `c()`. Dans ce contexte, la fonction `c()` prends un nombre arbitraire d'*arguments*. Elle retourne un vecteur en concaténant ses arguments¹ (en les mettant bout à bout). Voir `<undefined> [...]`, page `<undefined>`

Un chiffre apparaissant seul dans une expression est interprété comme un vecteur de longueur 1.

On note que l'opérateur d'affectation (`<-`) n'est **pas** l'usuel opérateur `=` qui est réservé pour d'autres usages. Il est constitué de deux caractères, `<` et `-` placés strictement cote à cote et qui 'pointent' vers l'objet recevant la valeur de l'expression².

On peut également utiliser la fonction `assign()`. Avec celle-ci, une façon équivalente de faire l'affectation précédente serait :

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

L'opérateur usuel, `<-`, peut être vu comme un raccourci de cette opération.

Les affectations peuvent également être faites dans l'autre sens, en utilisant le changement trivial de l'opérateur. Cette même affectation peut donc s'écrire comme suit

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

Si une expression est utilisée comme une commande complète, la valeur est imprimée à l'écran *et perdue*³.

Si on utilisait la commande

```
> 1/x
```

les inverses des 5 valeurs seraient imprimées (et la valeur de **x**, bien sûr, resterait inchangée).

L'affectation suivante

```
> y <- c(x, 0, x)
```

crée un vecteur **y** avec 11 éléments : 2 copies de **x** avec un zéro au milieu.

¹ Avec d'autres types d'arguments que des vecteurs, comme des arguments de mode liste, l'action de `c()` est sensiblement différente...

² Le caractère underscore `'_'` est un synonyme de `<-` mais il rend le code moins lisible. On décourage donc son usage.

³ elle reste cependant disponible dans `.Last.value` avant qu'une autre action soit exécutée

2.2 Opérations sur les vecteurs

Les vecteurs peuvent être utilisés comme des expressions arithmétiques. Dans ce cas, les opérations sont effectuées élément par élément.

Les vecteurs utilisés dans une même expression doivent tous être de même longueur. Si ce n'est pas le cas, la valeur de l'expression est un vecteur de même longueur que le plus grand vecteur utilisé dans l'expression.

Les vecteurs plus courts sont alors *recyclés* autant de fois que c'est nécessaire (avec une éventuelle troncature) pour qu'ils aient la même taille que le plus grand. En particulier, une constante est simplement répétée.

Avec la commande d'affectation suivante

```
> v <- 2*x + y + 1
```

On génère un nouveau vecteur `v` de longueur 11 construit en additionnant `2*x` répété 2.2 fois avec `y` répété 1 fois et `1` répété 11 fois.

Les opérateurs arithmétiques élémentaires sont `+`, `-`, `*`, `/` et `^` pour élever à une puissance. De plus toutes les fonctions arithmétiques usuelles sont disponibles. `log`, `exp`, `sin`, `cos`, `tan`, `sqrt` (racine carrée) etc. ont leur sens usuel. `max` et `min` renvoient respectivement le plus grand et le plus petit élément d'un vecteur. `range` est une fonction retournant un vecteur de longueur équivalent à `c(min(x),max(x))`. `length(x)` renvoie le nombre d'éléments de `x`, `sum(x)` donne la somme des éléments de `x` et `prod(x)` leur produit.

Deux fonction statistiques de base : `mean(x)` calcule la moyenne d'un échantillon, c'est à dire la même chose que `sum(x)/length(x)`, et `var(x)` calcule

$$\text{sum}((x-\text{mean}(x))^2)/(\text{length}(x)-1)$$

soit la variance de l'échantillon.

Si l'argument de `var()` est une matrice n -par- p , la valeur retournée est une matrice p -par- p de covariance obtenue en considérant les lignes comme des p -variables indépendantes (individus de l'échantillon).

`sort(x)` est une fonction de tri qui renvoie un vecteur de la même taille que `x` avec ses éléments rangés par ordre croissant. Il existe d'autres fonctions adaptées à différentes sortes de tris (voir `order()` et `sort.list()`).

`max()` (resp. `min()`) sélectionne le plus grand (resp. petit) élément de son ou ses arguments (même si ce sont plusieurs vecteurs).

`pmax()` (resp. `pmin()`) est la fonction *parallèle* de maximum (resp. minimum). Elle renvoie un vecteur de longueur égale à celle de son plus grand argument, qui contient le plus grand (resp. petit) élément de chaque vecteur donné en argument.

La plupart du temps l'utilisateur n'aura pas à se soucier du fait que les nombres qui composent un vecteur numérique sont des entiers, des réels ou même des complexes : les calculs internes sont de toute façon faits en double précision.

Pour travailler avec des nombres complexes il suffit d'ajouter explicitement la partie imaginaire. Ainsi

```
sqrt(-17)
```

donnera une erreur alors que

```
sqrt(-17+0i)
```

fera les calculs.

2.3 Générer des séquences de nombres

R permet de générer facilement les séquences de nombres les plus utiles. `1:30`, par exemple, correspond au vecteur `c(1,2,3,...,29,30)`. L'opérateur `:` a la plus haute priorité dans une expression. Ainsi `2*1:15` est le vecteur `c(2,4,6,...,28,30)`. On peut s'amuser pour s'en convaincre à comparer, avec `n<-10`, les suites `1:n-1` et `1:(n-1)`.

La construction `30:1` peut être utilisée pour générer une séquence en sens inverse.

On dispose également de la fonction `seq()` qui est une façon plus générale de produire des séquences. Elle admet 5 arguments qui ne peuvent être utilisés tous en même temps. Les 2 premiers arguments, s'ils sont spécifiés, donnent le début et la fin de la séquence. Si ces 2 arguments sont les seuls à être donnés en entrée, le résultat est le même que pour l'opérateur `:`. Ainsi `seq(2,10)` est équivalent à `2:10`.

Les paramètres de `seq`, de même que ceux de nombreuses autres fonctions R, peuvent être passés en argument avec leur nom. Dans ce cas, l'ordre dans lequel ils sont donnés n'a pas d'importance. Les deux premiers arguments de `seq()` peuvent être passés sous la forme `from=valeur` et `to=valeur`; Ainsi `seq(1,30)`, `seq(from=1,to=30)` et `seq(to=30,from=1)` sont tous équivalents à `1:30`. Deux des autres arguments de `seq` sont `by=valeur` et `length=valeur`. ils spécifient respectivement la taille du pas et la longueur de la séquence. Si aucun des deux n'est donné, la valeur par défaut `by=1` est utilisée.

Par exemple

```
> seq(-5, 5, by=.2) -> s3
```

affecte à `s3` le vecteur `c(-5.0,-4.8,-4.6,...,4.6,4.8,5.0)`.

De la même façon,

```
> s4 <- seq(length=51, from=-5, by=.2)
```

affecte le même vecteur à `s4`.

Le 5ème argument est `along=vecteur`, qui doit être utilisé seul. Il crée une séquence `1,2,...,length(vecteur)` ou une suite vide si le vecteur est vide (il peut donc l'être...).

`rep()` est une fonction assez voisine de la précédente qui est utilisée pour répéter un objet. La forme la plus simple de cette fonction (qui peut faire des choses plutôt complexes) est

```
> s5 <- rep(x, times=5)
```

cela mettra 5 copies de `x` bout à bout dans `s5`.

2.4 Vecteurs logiques

Comme pour les vecteurs numériques, R permet de faire des calculs avec des quantités logiques. Les éléments des vecteurs logiques peuvent prendre uniquement 2 valeurs, représentées par `FALSE` et `TRUE` (abréviées respectivement `F` et `T`).

Les vecteurs logiques sont générés par des *conditions*. Par exemple

```
> temp <- x > 13
```

initialise `temp` à un vecteur de même longueur que `x` et dont les éléments ont la valeur `FALSE` pour les éléments de `x` ne satisfaisant *pas* la condition, et `TRUE` pour les autres.

Les opérateurs logiques sont `<`, `<=`, `>`, `>=`, `==` pour l'égalité logique et `!=` pour l'inégalité. De plus, si `c1` et `c2` sont des expressions logiques, alors `c1 & c2` est leur intersection ("*et*"), `c1 | c2` est leur union ("*ou*") et `!c1` est la négation de `c1`.

Les vecteurs logiques peuvent être utilisés comme des vecteurs arithmétiques pour des calculs. Dans ce cas ils sont *forcés* en vecteurs numériques pour lesquels FALSE devient 0 et TRUE 1.

Il existe cependant des situations dans lesquelles les vecteurs logiques et leur pendant numérique ne sont pas équivalents (voir le sous-chapitre suivant pour un exemple).

2.5 Valeurs manquantes

Dans certains cas, les composants d'un vecteur peuvent ne pas tous être connus. Quand un élément ou une valeur n'est "pas disponible" ou "valeur manquante" au sens statistique du terme, une place dans le vecteur peut lui être réservée en employant la valeur particulière NA.

En général, toute opération sur un NA donne un NA. La raison pour laquelle cette règle a été mise en place est que si l'on ne connaît pas tous les éléments d'une opération, le résultat ne peut pas être connu et donc n'est pas disponible. La fonction `is.na(x)` donne un vecteur logique de la même taille que `x` avec la valeur TRUE si et seulement si l'élément correspondant de `x` est un NA.

```
> z <- c(1:3,NA); ind <- is.na(z)
```

A noter que l'expression logique `x==NA` donne un résultat complètement différent de `is.na(x)` puisque NA n'est pas vraiment une valeur mais un identificateur pour une quantité qui n'est pas disponible. `x==NA` est donc un vecteur de même longueur que `x` dont *tous* les éléments sont des NA puisque l'expression logique est incomplète.

Il y a une deuxième sorte de valeur "manquante" qui est produite par les calculs numériques : les valeurs appelées *Not A Number*, NaN. Par exemple

```
> 0/0
```

ou

```
> Inf - Inf
```

donnent toutes deux NaN car le résultat n'est pas défini.

Enfin, `is.na(xx)` renvoie TRUE pour les NA *et* les NaN tandis que `is.na(xx)` donne TRUE seulement pour les NaN.

2.6 Vecteurs de caractères

Les quantités caractères et les vecteurs de caractères sont fréquemment utilisés dans R, par exemple pour les labels des graphes. Ils sont représentés par une séquence de caractères délimitée par un double guillemet : "`x-max`", "`résultats de l'itération`".

Les vecteurs de caractères peuvent être constitués avec la fonction `c()`.

La fonction `paste()` prend un nombre quelconque d'arguments et les concatène en une seule chaîne de caractères. Tous les nombres donnés parmi les arguments sont transformés en chaînes de caractères. Par défaut, les arguments sont séparés par un caractère espace. Ceci peut être changé en utilisant le paramètre `sep=chaine`, qui échange le caractère espace pour *chaine* (qui peut être vide).

Par exemple

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

met dans `labs` le vecteur de caractères

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

On remarque que le recyclage a aussi cours ici; ainsi `c("X", "Y")` est répété 5 fois pour correspondre à la séquence `1:10`.⁴

2.7 Vecteurs indice; sélectionner et modifier des parties d'un jeu de données

Une partie des éléments d'un vecteur peut être sélectionnée en ajoutant après le nom du vecteur un *vecteur indice* entre crochets. Plus généralement, quand une expression retourne un vecteur, on peut en sélectionner une partie en ajoutant un vecteur indice immédiatement après l'expression.

Les vecteurs indice peuvent être de 4 types distinct :

1. **Un vecteur logique.** Dans ce cas le vecteur indice doit être de même longueur que le vecteur sélectionné. Les valeurs correspondant à `TRUE` dans le vecteur logique sont sélectionnées et les valeurs correspondant à `FALSE` sont omises.

Par exemple

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

crée (ou recrée) un objet `y` qui contiendra les valeurs autres que manquantes, dans le même ordre qu'elles apparaissent dans `x`. Si `x` a des valeurs manquantes, `y` sera plus court que `x`. Ainsi

```
> (x+1)[(!is.na(x)) & x>0] -> z
```

crée un objet `z` et place dedans les valeurs de `x` qui sont non manquantes et positives.

2. **Un vecteur d'entiers positifs.** Dans ce cas les éléments de l'indice doivent appartenir à l'ensemble $\{1, 2, \dots, \text{length}(x)\}$. Les éléments correspondant du vecteur sont placés *suivant l'ordre* donné par l'indice dans le vecteur résultat. Ce vecteur indice n'est pas forcément de même longueur que le vecteur. Par exemple `x[6]` renvoie le sixième composant de `x` et

```
> x[1:10]
```

sélectionne les 10 premiers éléments de `x` (si, bien sûr, `length(x)` est supérieur à 10). De même

```
> c("x","y")[rep(c(1,2,2,1), times=4)]
```

produit un vecteur de caractères de longueur 16 constitué de "x", "y", "y", "x" répété 4 fois.

3. **Un vecteur d'entiers négatifs.** Un vecteur d'indice de cette forme sert à spécifier les valeurs à *écarter* plutôt que celles à sélectionner. Ainsi

```
> y <- x[-(1:5)]
```

met dans `y` tous les éléments de `x` sauf les 5 premiers.

⁴ `paste(..., collapse=ss)` permet d'envoyer les arguments dans une chaîne de caractères en mettant `ss` entre eux. Il y a beaucoup d'autres outils pour la manipulation des caractères. Voir l'aide pour `sub` et `substring`.

4. **Un vecteur de caractères.** Cette possibilité s'applique seulement dans le cas où l'objet a un attribut `names` pour identifier ses éléments. Dans ce cas un sous-vecteur du vecteur des noms permet de sélectionner des éléments de `x` de la même façon que les vecteurs d'entiers du cas 2.

```
> fruit <- c(5, 10, 1, 20)
> names(fruit) <- c("orange", "banana", "apple", "peach")
> lunch <- fruit[c("apple", "orange")]
```

L'avantage des caractères (et de l'attribut `names`) sur les chiffres c'est qu'on s'en souvient plus facilement. Cette dernière solution est particulièrement utile quand on manipule des *data frames* comme nous le verrons plus tard.

Une expression indicée peut aussi apparaître du côté réception de la flèche. Dans ce cas l'affectation se fait *seulement pour ces éléments (sélectionnés) du vecteur*. L'expression doit être de la forme `vecteur[vecteur_indice]` puisqu'avoir une expression à la place du nom du vecteur n'aurait pas ici beaucoup de sens. Le vecteur affecté doit évidemment être de même longueur que le vecteur indice.

```
> x[is.na(x)] <- 0
```

remplace toute valeur manquante de `x` par zero et

```
> y[y < 0] <- -y[y < 0]
```

a le même effet que

```
> y <- abs(y)
```

2.8 Autres types d'objets

Les vecteurs représentent le type d'objet le plus important de R mais il en est d'autres que nous rencontrerons plus précisément dans les chapitres suivants

- *Les matrices* ou plus généralement les *tableaux* sont des généralisations multidimensionnelles des vecteurs. En fait ce *sont* des vecteurs qui peuvent être indicés par 2 (ou plus) indices et qui seront affichés à l'écran de façon spéciale. Voir (undefined) [Tableaux et matrices], page (undefined).
- *Les facteurs* permettent de manipuler les données qualitatives. Voir (undefined) [Facteurs], page (undefined).
- *Les listes* sont une forme plus générale de vecteurs dans laquelle les éléments n'ont pas besoin d'être tous du même type. Ces éléments sont souvent eux-même des vecteurs ou des listes. Les listes fournissent un bon moyen pour renvoyer les résultats de fonctions statistiques. Voir (undefined) [Listes et data frames], page (undefined).
- *Les data frames* sont des structures semblables aux matrices, dans lesquelles les colonnes peuvent être de types différents. Pensez aux data frames comme à des 'matrices de données' avec une ligne par observation et avec (si nécessaire) des variables numériques et catégorielles. Les data frames sont adaptés pour traiter un grand nombre de données expérimentales : les traitements sont catégoriels mais la réponse est numérique. Voir (undefined) [Listes et data frames], page (undefined).
- *les fonctions* sont aussi des objets en R et elles peuvent être stockées dans le workspace d'un projet. Ceci fournit un moyen pratique et simple d'étendre R. Voir (undefined) [Ecrire vos propres fonctions], page (undefined).

3 Objets; leurs modes et attributs

3.1 Attributs intrinsèques : mode et longueur

Les éléments sur lesquels agit R sont techniquement nommés *objets*. Les vecteurs de réels ou de nombres complexes, les vecteurs de valeurs logiques ou de chaînes de caractères en sont des exemples. Tous ces éléments sont des structures dites "atomiques" car leur composants sont tous de même type ou mode, respectivement *numeric*¹, *complex*, *logical* et *character*.

Les éléments d'un vecteur doivent *tous être du même mode*. Ainsi tout vecteur doit sans ambiguïté être *numeric*, *complex*, *logical* ou *character*. La seule exception à cette règle est pour les valeurs spéciales **NA** représentant des valeurs manquantes.

On notera qu'un vecteur vide peut aussi avoir un mode,. Par exemple, le vecteur de chaîne de caractère vide est listé `character(0)` tandis que le vecteur numérique vide donne `numeric(0)`.

R agit aussi sur des objets appelés *listes*, qui sont de mode *list*. Ce sont des séquences ordonnées d'objets qui peuvent être de n'importe quel mode. Les listes sont dites "récursives", plutôt qu'atomiques, car leurs composants peuvent eux-même être des listes.

Les structures de mode *function* et *expression* sont également récursives. Les fonctions sont des objets qui forment la base de R, avec les fonctions utilisateur sur lesquelles nous reviendrons plus en détail par la suite. Les expressions, en tant qu'objets, ont un rôle assez complexe en R. Elles ne seront pas documentées dans ce guide, si ce n'est indirectement dans la description des *formules* utilisées pour la modélisation.

On nomme *mode* d'un objet le type de base de ses éléments fondamentaux. C'est l'une des "propriétés" d'un objet. La *longueur* est une autre des propriétés que possède tout objet. Les fonctions `mode(objet)` et `length(objet)` permettent d'accéder au mode et à la longueur de toute structure.

D'autres propriétés d'un objet peuvent être extraites par `attributes(object)`, voir Section 3.3 [Obtenir et modifier des attributs], page 14. Les *mode* et *longueur* sont appelés "attributs intrinsèques" d'un objets car ils sont communs à tous les objets.

Par exemple, si `z` est un vecteur de complexes de longueur 100, alors, dans une expression, `mode(z)` est la chaîne de caractères "`complex`" et `length(z)` est l'entier 100.

R s'occupe des changements de mode partout où il est sensé le faire. Par exemple avec

```
> z <- 0:9
```

On peut faire

```
> nombres <- as.character(z)
```

après quoi `nombres` sera le vecteur de caractères `c("0", "1", "2", ..., "9")`. Un *forçage* de plus, ou changement de mode, reconstruit le vecteur numérique :

```
> d <- as.integer(digits)
```

¹ Le mode *numeric* est un amalgame de deux modes distincts, *integer* et *double* précision.

maintenant `d` et `z` sont les mêmes.² Il y a une grande diversité de fonctions de la forme `as.qqchose()` soit pour le forçage d'un mode à l'autre, soit pour rajouter un attribut à un objet. Le lecteur consultera les différents fichiers d'aide pour se familiariser avec ces fonctions.

3.2 Changer la longueur d'un objet

Un objet peut être vide et avoir un mode. Par exemple

```
> e <- numeric()
```

fait de `e` une structure vectorielle de mode numérique. De la même façon, `character()` est un vecteur -vide- de caractères et ainsi de suite. Une fois qu'un objet, quelle que soit sa taille, a été créé, de nouveaux composants peuvent lui être ajoutés en l'indiquant avec une valeur plus grande que son rang. Ainsi

```
> e[3] <- 17
```

fait de `e` un vecteur de longueur 3 (dont les deux premiers sont pour l'instant des `NA`). Ceci s'applique à toute structure, en admettant que le mode des nouveaux arguments est compatible avec celui des premiers objets de la structure.

Cet ajustement automatique de la longueur des objets est souvent utilisé, par exemple pour les entrées de la fonction `scan()`. (Voir `<undefined>` [La fonction `scan()`], page `<undefined>`.)

De même, tronquer la taille d'un objet ne nécessite qu'une affectation. Ainsi si `alpha` est un objet de longueur 10, alors

```
> alpha <- alpha[2 * 1:5]
```

en fait un objet de longueur 5 avec les composants sélectionnés, indices compris. Les anciens indices ne sont bien sûr pas retenus.

3.3 Obtenir et modifier des attributs

La fonction `attributes(objet)` donne une liste de tous les attributs non-intrinsèques définis pour cet objet. La fonction `attr(objet, nom)` peut être utilisée pour sélectionner un attribut spécifique. Ces fonctions sont rarement utilisées, sauf dans des cas assez particuliers, quand on crée un nouvel attribut dans un but précis (par exemple pour associer une date de création ou un opérateur à un objet R). Le concept, cependant, est très important.

On devra donc faire attention, quand on ajoute ou supprime des attributs, car ils sont partie intégrante du système d'objets utilisé en R.

utiliser la fonction `attr` du côté gauche d'une affectation sert soit à associer un nouvel attribut à un objet, soit à en modifier un déjà existant. Par exemple

```
> attr(z,"dim") <- c(10,10)
```

permet de traiter `z` comme une matrice 10x10.

² En général, le forçage de `numeric` à `character` ne sera pas tout à fait équivalent à cause des erreurs d'arrondis dans la représentation des caractères

3.4 La classe d'un objet

Un attribut spécial, appelé *classe* d'un objet, est utilisé pour un style de programmation R orientée objet.

Par exemple si un objet est de classe `"data.frame"`, il sera affiché d'une certaine façon, la fonction `plot` fera un graphique adaptée, et d'autres fonctions génériques, comme `summary()`, le traiteront de la façon appropriée à sa classe.

Pour supprimer temporairement les effets dus à une classe, on utilise la fonction `unclass()`. Par exemple si `winter` est de classe `"data.frame"` alors

```
> winter
```

l'affichera comme un data frame, ce qui ressemble à la façon dont sont affichées les matrices, tandis que

```
> unclass(winter)
```

l'affichera comme une liste ordinaire. Ce n'est que dans des situations particulières que l'on a besoin de cette fonctionnalité, et seulement quand on en vient à utiliser les fonctions génériques

Ces dernières seront brièvement développées dans [\(undefined\) \[Oriente objet\]](#), page [\(undefined\)](#).

4 Listes et data frames

4.1 Listes

Une *liste* R est un objet qui contient une collection ordonnée d'objets appelés ses *composants*.

Il n'est pas nécessaire que les composants soient du même type ou du même mode, et, par exemple, une liste peut être composée d'un vecteur numérique, d'une valeur logique, d'une matrice, d'un vecteur de complexes, d'un tableau de caractères, d'une fonction etc. Voici un exemple de la manière de créer une liste :

```
> Lst <- list(nom="Fred", femme="Stella", nb.enfants=3,
             ages.enfants=c(4,7,9))
```

Les composants sont toujours numérotés et peuvent donc être identifiés par leur numéro. Ainsi, si `Lst` est le nom d'une liste de 4 composants, on peut accéder à ceux-ci par `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` et `Lst[[4]]`. Si, de plus, `Lst[[4]]` est un vecteur alors `Lst[[4]][1]` est sa première valeur.

Si `Lst` est une liste, alors la fonction `length(Lst)` donne le nombre de ses composants (de premier niveau).

Les composants d'une liste peuvent aussi être *nommés*, et dans ce cas on peut y accéder soit en remplaçant le nombre entre crochets par le nom (sous forme de chaîne de caractères), soit, de façon plus conventionnelle, en donnant une expression de la forme

```
> name$nom_composant
```

Cette convention est très utile car on peut accéder au composant même si on a oublié son numéro.

Dans l'exemple basique ci-dessus :

`Lst$nom` identifie la même chose que `Lst[[1]]` c'est à dire la chaîne "Fred",

`Lst$femme` identifie la même chose que `Lst[[2]]` c'est à dire la chaîne "Stella",

`Lst$child.ages[1]` identifie la même chose que `Lst[[4]][1]` c'est à dire le nombre 4.

De plus on peut utiliser les noms des composants entre double crochets, i.e., `Lst[["nom"]]` ce qui est la même chose que `Lst$nom`. Ceci est particulièrement utile quand le nom du composant à extraire est stocké dans une variable; comme

```
> x <- "name"; Lst[[x]]
```

Il est très important de distinguer `Lst[[1]]` de `Lst[1]`. '`[[...]]`' est un opérateur utilisé pour sélectionner un seul élément, tandis que '`[...]`' est un opérateur plus général d'indilage. Ainsi la première forme donne le *premier objet de la liste Lst*, et, si c'est une liste nommée, le nom de l'objet n'est **pas** inclus. La deuxième forme extrait une *sous-liste de la liste Lst* constituée de la première entrée seulement. Si c'est une liste nommée, les noms sont inclus dans la sous-liste.

Les noms des composants peuvent être raccourcis au plus petit nombre de lettres nécessaire pour les distinguer de manière unique. Ainsi `Lst$coefficients` peut être spécifié au minimum par `Lst$coe` et `Lst$covariance` par `Lst$cov`. Le vecteur des noms est en fait un simple attribut de la liste et peut donc être utilisé comme tel. D'autres structures que les listes peuvent aussi avoir un attribut *names*.

4.2 Construire et modifier des listes

Les listes peuvent être construites à partir d'objets existants avec la fonction `list()`. Une affectation de la forme

```
> Lst <- list(nom_1=objet_1, ..., nom_m=objet_m)
```

initialise une liste `Lst` de m composants avec `objet_1, \dots, objet_m` pour composants dont les noms sont respectivement `nom_1, \dots, nom_m`. Si ces nom étaient omis, les composants seraient repérés seulement par leurs numéros correspondant. Les composants utilisés sont *copiés* pour former la liste, et les originaux restent inchangés.

Les listes, comme tous les objets indiquables, peuvent être étendues en leur ajoutant des composants. Par exemple

```
> Lst[5] <- list(matrix=Mat)
```

4.2.1 Concatèner des listes

Quand les arguments de la fonction de concaténation `c()` sont des listes, le résultat est aussi un objet de mode liste, dont les composants sont ceux des listes ajoutés bout à bout

```
> list.ABC <- c(list.A, list.B, list.C)
```

Rappelez-vous qu'avec des vecteurs en argument, la fonction de concaténation met de façon similaire tous les arguments dans une seule structure vectorielle. Dans ce cas, les autres attributs, comme les attributs `dim`, sont perdus.

4.3 Data frames

Un "*data.frame*" est une liste de classe "`data.frame`", mais il y a des restrictions sur les listes pour devenir des data frames :

- Les composants doivent être des vecteurs (numériques, de caractères ou logiques), des facteurs, des matrices numériques, des listes ou d'autres data frames.
- Les matrices, listes ou data frames fournissent autant de variables au nouveau data frame qu'elles ont de colonnes, d'éléments ou de variables respectivement.
- Les vecteurs numériques et les facteurs sont inclus tels quels mais les vecteurs non-numériques (caractères et logiques) sont forcés en facteurs dont les niveaux sont les valeurs (uniques) apparaissant dans le vecteur.
- Les structures vectorielles variables du data frame doivent avoir la *même longueur*, et les structures matricielles doivent avoir le même *nombre de lignes*.

Un data frame peut, dans de nombreux cas, être vu comme une matrice avec des colonnes ayant des modes et attributs différents. Il peut être affiché comme une matrice et ses lignes et colonnes extraites en utilisant les conventions d'indication des matrices.

4.3.1 Construire des data frames

Les objets satisfaisant les conditions sur les colonnes (composants) d'un data frame peuvent être utilisés pour en construire un à l'aide de la fonction `data.frame`:

```
> accountants <- data.frame(home=statef, loot=income, shot=incomef)
```

Une liste dont les composants se conforment aux restrictions imposées aux data frame peut être *forcée* en data frame en utilisant la fonction `as.data.frame()`

La façon la plus simple de construire un data frame à partir de rien est d'utiliser la fonction `read.table` pour lire un data frame stocké dans un fichier. Ceci est expliqué plus en détails dans la section `<undefined>` [Lire des données dans des fichiers], page `<undefined>`.

4.3.2 `attach()` et `detach()`

La notation `$`, comme dans `accountants$statef`, n'est pas toujours très pratique pour accéder aux composants des listes. Il serait utile de les rendre temporairement visibles comme si c'étaient des variables (sans avoir besoin d'y adjoindre constamment le nom de la liste).

La fonction `attach()`, de même qu'elle peut avoir un nom de répertoire en argument, peut aussi avoir un data frame. En supposant que `Lentilles` est un data frame avec 3 variables `Lentilles$u`, `Lentilles$v`, `Lentilles$w` l'attacher

```
> attach(Lentilles)
```

place le data frame en position 2 du path. S'il n'y a pas de variables `u`, `v` ou `w` en position 1, `u`, `v` et `w` sont maintenant manipulables en tant que variables du data frame mais sous leurs propres noms.

Ici une affectation comme

```
> u <- v+w
```

ne modifiera pas le composant `u` du data frame mais le masquera par une nouvelle valeur `u` en position 1 du search path du répertoire de travail. Pour faire un changement dans le data frame, la façon la plus simple est de se resservir de la notation `$` :

```
> Lentilles$u <- v+w
```

Cependant la nouvelle valeur du composant `u` ne sera pas visible avant que le data frame ne soit détaché.

Pour détacher un data frame on utilise la fonction

```
> detach()
```

Ceci, plus précisément, détache du search path ce qui se trouve en position 2. Ainsi, dans ce contexte, les variables `u`, `v` et `w` ne seront plus visibles, excepté avec la notation de liste `$`. Les entités aux positions supérieures à 2 dans le search path peuvent être détachées en donnant leur numéro en argument de `detach`, mais il est toujours plus prudent d'utiliser leur nom, par exemple par `detach(Lentilles)` ou `detach("Lentilles")`

NOTE: Dans la version actuelle de R, le search path ne peut contenir plus de 20 éléments. Évitez d'attacher le même data frame plus d'une fois. Détachez toujours un data frame aussitôt que vous en avez fini avec ses variables.

NOTE: Dans la version actuelle, les data frames ne peuvent être attachés qu'en position 2 ou après.

4.3.3 Travailler avec les data frames

Voici quelques règles qui vous permettront de travailler de front sur de nombreux problèmes dans le même répertoire :

- rassemblez, pour chaque analyse, toutes les variables dans un data frame au nom explicite;
- quand vous travaillez sur une analyse, attachez le data frame correspondant en position 2 et utilisez le répertoire de travail au niveau 1 pour stocker les variables intermédiaires et temporaires;
- avant d'en terminer avec votre analyse, rajoutez toutes les variables de quelque intérêt à votre data frame en utilisant le `$` et ensuite faites `detach()`;
- finalement, effacez du répertoire de travail les variables indésirables et essayer de le garder vierge de toute variable temporaire dans la mesure du possible.

De cette façon il est facile de travailler sur de nombreux problèmes dans le même répertoire, même si on a dans plusieurs de ceux-ci des variables `x`, `y` et `z` par exemple.

4.3.4 Attacher des listes

`attach()` est une fonction générique qui permet d'attacher non seulement des répertoires et des data frames au search path, mais aussi d'autres classes d'objets. En particulier, tout objet de mode `list` peut être attaché :

```
> attach(une.vieille.liste)
```

Tout ce qui a été attaché peut ensuite être détaché par `detach`, avec le numéro de position, ou, de préférence, avec le nom en argument.

4.3.5 Gérer le search path

La fonction `search` affiche le search path courant et est donc très utile pour garder la trace des data frames et des listes (et aussi des packages) qui ont été attachés et détachés. Au départ cela donne :

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

où `.GlobalEnv` est le workspace.¹

Après que `Lentilles` ait été attaché on a

```
> search()
[1] ".GlobalEnv" "Lentilles" "Autoloads" "package:base"
> ls(2)
[1] "u" "v" "w"
```

comme nous avons vu `ls` (ou `objects`) peut être utilisé pour examiner le contenu de chaque position du search path.

Finalement, on détache le data frame et on vérifie qu'il a bien été détaché.

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

¹ Voir l'aide en ligne pour `autoload` pour la signification du deuxième terme.

Appendice A Une session d'introduction à R

La session qui va suivre se propose de vous présenter certaines fonctionnalités de l'environnement R par la pratique. De nombreuses fonctionnalités sembleront déroutantes au premier abord, mais cela disparaîtra avec l'usage. Ceci est écrit pour l'utilisateur UNIX, ceux qui utilisent Windows devront adapter un peu.

Loggez-vous et démarrez votre gestionnaire de fenêtres. Vous devez avoir le fichier `'morley.data'` dans votre répertoire de travail. Si ce n'est pas le cas, adressez-vous à votre responsable système (si vous avez la chance d'en avoir un;) ou prenez-la directement dans le répertoire `'base/data'` de la bibliothèque R (dans `'/usr/local/R'` par exemple).

`$ R` Démarre R.

Le programme R commence, avec un bandeau de commentaires..

(Par la suite le prompt de R ne sera pas représenté pour éviter les confusions.)

`help.start()`

Démarre l'aide HTML (en utilisant un butineur disponible sur votre machine).

Vous pouvez regarder un peu les différentes possibilités pour obtenir de l'aide.

Iconifiez ensuite la fenêtre.

`x <- rnorm(50)`

`y <- rnorm(x)`

Génère 2 vecteurs x et y pseudo-aléatoires de loi normales et de longueur 50.

`plot(x, y)`

dessine les points dans le plan. une fenetre graphique apparaitra automatiquement.

`ls()`

Visualise les objets qui sont dans l'espace de travail de R.

`rm(x, y)` supprime les objets dont on n'a plus besoin.

`x <- 1:20` Crée $x = (1, 2, \dots, 20)$.

`w <- 1 + sqrt(x)/2`

Cre un vecteur de 'poids' (écarts types) w .

`dummy <- data.frame(x=x, y= x + rnorm(x)*w)`

`dummy` Construit un *data frame* de 2 colonnes, x et y , et regarde ce qu'il y a dedans..

`fm <- lm(y ~ x, data=dummy)`

`summary(fm)`

Regression linéaire de y sur x et affichage de l'analyse.

`fm1 <- lm(y ~ x, data=dummy, weight=1/w^2)`

`summary(fm1)`

Vu qu'on connait les écarts types, on peut faire une regression pondérée.

`attach(dummy)`

Rends les colonnes du data frame visibles comme des variables.

`lrf <- lowess(x, y)`

Régression non-paramétrique.

```

plot(x, y)
    Graphique standard.

lines(x, lrf$y)
    Ajoute la courbe de régression au graphe.

abline(0, 1, lty=3)
    La vraie droite de régression : (ordonnée à l'origine 0, pente 1).

abline(coef(fm))
    La droite de la régression non-pondérée.

abline(coef(fm1), col = "red")
    La droite de la régression pondérée.

detach()
    Enlève le data frame de la search list.

plot(fitted(fm), resid(fm),
     xlab="Fitted values",
     ylab="Residuals",
     main="Residuals vs Fitted")
    Un diagnostic graphique standard de régression pour confirmer ou infirmer
    l'hypothèse d'hétéroscédasticité. Pouvez-vous le voir ?

qqnorm(resid(fm), main="Residuals Rankit Plot")
    Un graphe de normalité des scores pour repérer les individus aberrants (pas
    très utile dans ce cas).

rm(fm, fm1, lrf, x, dummy)
    On refait le ménage...

    On étudiera dans la section qui vient des données provenant des expériences de Michael-
    son et Morley ayant pour but de mesurer la vitesse de la lumière.

file.show("morley.tab")
    Optionnel. Interrompt temporairement R et consulte le fichier.

mm <- read.table("morley.tab")
mm
    enregistre les données de Michaelson et Morley comme data frame, et le con-
    sulte. Il y a cinq expériences (colonne Expt) répétés 20 fois (colonne Run) et sl
    est la vitesse de la lumière enregistrée.

mm$Expt <- factor(mm$Expt)
mm$Run <- factor(mm$Run)
    Change Expt et Run en facteurs.

attach(mm)
    rends le data frame visible en position 2 (le défaut).

plot(Expt, Speed, main="Speed of Light Data", xlab="Experiment No.")
    Compare les 5 expériences avec des botes à moustaches.

fm <- aov(Speed ~ Run + Expt, data=mm)
summary(fm)
    Analyse de la variance, avec 'runs' et 'experiments' comme facteurs.

```

```
fm0 <- update(fm, . ~ . - Run)
anova(fm0, fm)
```

Ajuste un sous modèle en omettant 'runs', puis le compare à l'autre.

```
detach()
rm(fm, fm0)
```

Fait le ménage avant la suite.

We now look at some more graphical features: contour and image plots.

```
x <- seq(-pi, pi, len=50)
```

$y <- x$ x is a vector of 50 equally spaced values in $-\pi \leq x \leq \pi$. y is the same.

```
f <- outer(x, y, function(x, y) cos(y)/(1 + x^2))
```

f is a square matrix, with rows and columns indexed by x and y respectively, of values of the function $\cos(y)/(1 + x^2)$.

```
oldpar <- par(no.readonly = TRUE)
```

```
par(pty="s")
```

Save the plotting parameters and set the plotting region to "square".

```
contour(x, y, f)
```

```
contour(x, y, f, nlevels=15, add=TRUE)
```

Make a contour map of f ; add in more lines for more detail.

```
fa <- (f-t(f))/2
```

fa is the "asymmetric part" of f . ($t()$ is transpose).

```
contour(x, y, fa, nint=15)
```

Make a contour plot, ...

```
par(oldpar)
```

... and restore the old graphics parameters.

```
image(x, y, f)
```

```
image(x, y, fa)
```

Make some high density image plots, (of which you can get hardcopies if you wish), ...

```
objects(); rm(x, y, f, fa)
```

... and clean up before moving on.

R can do complex arithmetic, also.

```
th <- seq(-pi, pi, len=100)
```

```
z <- exp(1i*th)
```

$1i$ is used for the complex number i .

```
par(pty="s")
```

```
plot(z, type="l")
```

Plotting complex arguments means plot imaginary versus real parts. This should be a circle.

```
w <- rnorm(100) + rnorm(100)*1i
```

Suppose we want to sample points within the unit circle. One method would be to take complex numbers with standard normal real and imaginary parts ...

```
w <- ifelse(Mod(w) > 1, 1/w, w)
```

... and to map any outside the circle onto their reciprocal.

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")  
lines(z)
```

All points are inside the unit circle, but the distribution is not uniform.

```
w <- sqrt(runif(100))*exp(2*pi*runif(100)*1i)
```

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")  
lines(z)
```

The second method uses the uniform distribution. The points should now look more evenly spaced over the disc.

```
rm(th, w, z)
```

Clean up again.

```
q()
```

Quit the R program. You will be asked if you want to save the R workspace, and for an exploratory session like this, you probably do not want to save it.

Appendice B Index des variables et fonctions

!		<	
!	9	<	9
!=	9	<=	9
&		A	
&	9	<code>as.data.frame</code>	18
*		<code>attr</code>	14
*	8	<code>attributes</code>	14
-		C	
-	8	<code>c</code>	7, 17
/		<code>cos</code>	8
/	8	D	
:		<code>data.frame</code>	17
:	9	E	
=		<code>exp</code>	8
=	9	H	
?		<code>help</code>	4
?	4	I	
 		<code>is.na</code>	10
.....	9	<code>is.nan</code>	10
+		L	
+	8	<code>list</code>	16
>		<code>log</code>	8
>	9	M	
>=	9	<code>max</code>	8
^		<code>mean</code>	8
^	8	<code>min</code>	8
		N	
		<code>NaN</code>	10
		O	
		<code>order</code>	8

P

<code>pmax</code>	8
<code>pmin</code>	8
<code>prod</code>	8

R

<code>range</code>	8
<code>rep</code>	9
<code>rm</code>	5

S

<code>seq</code>	9
<code>sin</code>	8
<code>sink</code>	5

<code>sort</code>	8
<code>source</code>	5
<code>sqrt</code>	8
<code>sum</code>	8

T

<code>tan</code>	8
------------------------	---

U

<code>unclass</code>	15
----------------------------	----

V

<code>var</code>	8
<code>vector</code>	7

Appendice C Index des concepts

A

Affectation 7

D

Data frames 17

F

Fonctions arithmetiques et operateurs 8

L

Lists 16

P

Packages 2

R

Rediriger les entrées et les sorties 5

regle de recyclage 8

S

Supprimer des objets 5

V

Valeurs manquantes 10

Vecteurs de caractères 10

W

Workspace 5

Appendice D Références

D. M. Bates and D. G. Watts (1988), *Nonlinear Regression Analysis and Its Applications*. John Wiley & Sons, New York.

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), *The New S Language*. Chapman & Hall, New York. Ce livre est souvent appelé le “*Blue Book*”.

John M. Chambers and Trevor J. Hastie eds. (1992), *Statistical Models in S*. Chapman & Hall, New York. Aussi appelé le “*White Book*”.

Annette J. Dobson (1990), *An Introduction to Generalized Linear Models*, Chapman and Hall, London.

Peter McCullagh and John A. Nelder (1989), *Generalized Linear Models*. Second edition, Chapman and Hall, London.

John A. Rice (1995), *Mathematical Statistics and Data Analysis*. Second edition. Duxbury Press, Belmont, CA.

S. D. Silvey (1970), *Statistical Inference*. Penguin, London.