Introduction à 

Julien Barnier
Centre Max Weber
CNRS – UMR 5283
julien.barnier@ens-lyon.fr

8 février 2012

Table des matières

Partie 1

Introduction

1.1 À propos de ce document

Ce document a pour objet de fournir une introduction à l'utilisation du logiciel libre de traitement de données et d'analyse statistiques R. Il se veut le plus accessible possible, y compris pour ceux qui ne sont pas particulièrement familiers avec l'informatique.

Ce document est basé sur `R version 2.14.1 (2011-12-22)`.

La page Web « officielle » sur laquelle on pourra trouver la dernière version de ce document se trouve à l'adresse :

http://alea.fr.eu.org/pages/intro-R

L'auteur tient à remercier Mayeul Kauffmann et Julien Biaudet pour leurs corrections et suggestions.

1.2 Conventions typographiques

Ce document suit un certain nombre de conventions typographiques visant à en faciliter la lecture. Ainsi les noms de logiciel et d'extensions sont indiqués en caractères sans empattement (R, SAS, Linux, rgrs, ade4...). Les noms de fichiers sont imprimés avec une police à chasse fixe (`test.R`, `data.txt`...), tout comme les fonctions R (`summary`, `mean`, `<-`...).

Lorsqu'on présente des commandes saisies sous R et leur résultat, la commande saisie est indiquée avec une police à chasse fixe et précédée de l'invite de commande `R>` :

```
R> summary(rnorm(100))
```

Le résultat de la commande tel qu'affiché par R est également indiqué dans une police à chasse fixe :

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -2.570  -0.708   0.096   0.034   0.555   3.730
```

Lorsque la commande R est trop longue et répartie sur plusieurs lignes, les lignes suivantes sont précédées du symbole `+` :

```
R> coo <- scatterutil.base(dfxy = dfxy, xax = xax, yax = yax,
+     xlim = xlim, ylim = ylim, grid = grid, addaxes = addaxes, cgrid = cgrid,
+     include.origin = include.origin)
```

1.3 Présentation de R

R est un langage orienté vers le traitement de données et l'analyse statistique dérivé du langage S. Il est développé depuis une vingtaine d'années par un groupe de volontaires de différents pays. C'est un logiciel libre [1], publié sous licence GNU GPL.

L'utilisation de R présente plusieurs avantages :

- c'est un logiciel *multiplateforme*, qui fonctionne aussi bien sur des sytèmes Linux, Mac OS X ou Windows ;
- c'est un logiciel *libre*, développé par ses utilisateurs et modifiable par tout un chacun ;
- c'est un logiciel *gratuit* ;
- c'est un logiciel très puissant, dont les fonctionnalités de base peuvent être étendues à l'aide d'extensions [2] ;
- c'est un logiciel dont le développement est très actif et dont la communauté d'utilisateurs ne cesse de s'élargir ;
- c'est un logiciel avec d'excellentes capacités graphiques.

Comme rien n'est parfait, on peut également trouver quelques inconvénients :

- le logiciel, la documentation de référence et les principales ressources sont en anglais. Il est toutefois parfaitement possible d'utiliser R sans spécialement maîtriser cette langue ;
- par son mode de fonctionnement, R charge normalement l'intégralité des données traitées en mémoire. Il nécessite donc une machine relativement puissante pour travailler sur des grosses enquêtes de plusieurs milliers d'individus ;
- il n'existe pas encore d'interface graphique pour R équivalente à celle d'autres logiciels comme SPSS ou Modalisa. R fonctionne à l'aide de scripts (des petits programmes) édités et exécutés au fur et à mesure de l'analyse, et se rapprocherait davantage de SAS dans son utilisation (mais avec une syntaxe et une philosophie très différentes).

À noter que ce dernier point, qui peut apparaître comme un gros handicap, s'avère après un temps d'apprentissage être un mode d'utilisation d'une grande souplesse.

1.4 Philosophie de R

Deux points particuliers dans le fonctionnement de R peuvent parfois dérouter les utilisateurs habitués à d'autres logiciels :

- sous R, en général, on ne voit pas les données sur lesquelles on travaille ; on ne dispose pas en permanence d'une vue des données sous forme de tableau, comme sous Modalisa ou SPSS. Ceci peut être déroutant au début, mais on se rend vite compte qu'on n'a pas besoin de voir en permanence les données pour les analyser ;
- avec les autres logiciels, en général la production d'une analyse génère un grand nombre de résultats de toutes sortes dans lesquels l'utilisateur est censé retrouver et isoler ceux qui l'intéressent. Avec R, c'est l'inverse : par défaut l'affichage est réduit au minimum, et c'est l'utilisateur qui demande à voir des résultats supplémentaires ou plus détaillés.

1. Pour plus d'informations sur ce qu'est un logiciel libre, voir : http://www.gnu.org/philosophy/free-sw.fr.html
2. Il en existe actuellement plus de 3500, disponibles sur le *Comprehensive R Archive Network* (CRAN) : http://cran.r-project.org/

Inhabituel au début, ce fonctionnement permet en fait assez rapidement de gagner du temps dans la conduite des analyses.

Partie 2

Prise en main

L'installation du logiciel proprement dite n'est pas décrite ici mais indiquée dans l'annexe A, page 130. On part donc du principe que vous avez sous la main un ordinateur avec une installation récente de R, quel que soit le système d'exploitation que vous utilisez (Linux, Mac OS X ou Windows).

2.1 L'invite de commandes

Une fois R lancé, vous obtenez une fenêtre appelée *console*. Celle-ci contient un petit texte de bienvenue ressemblant à peu près à ce qui suit [1] :

1. La figure 2.1 de la présente page montre l'interface par défaut sous Windows.

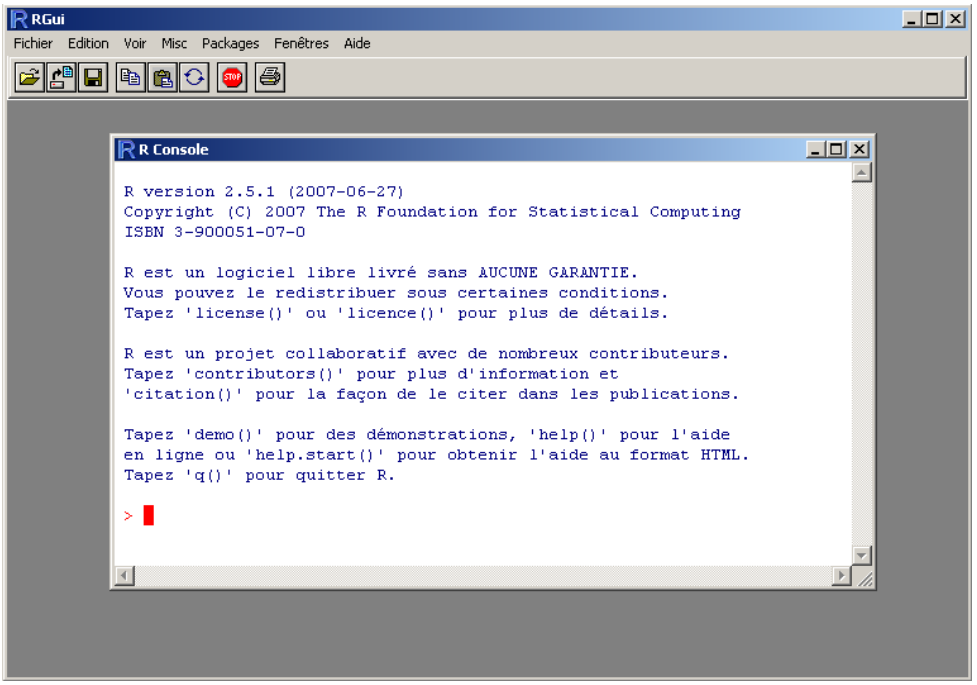


FIGURE 2.1 – L'interface de R sous Windows au démarrage

```
R version 2.14.1 (2011-12-22)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-linux-gnu (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

(...)
```

suivi d'une ligne commençant par le caractère `>` et sur laquelle devrait se trouver votre curseur. Cette ligne est appelée *l'invite de commande* (ou *prompt* en anglais). Elle signifie que R est disponible et en attente de votre prochaine commande.

Nous allons tout de suite lui fournir une première commande :

```
R> 2 + 3

[1] 5
```

Bien, nous savons désormais que R sait faire les additions à un chiffre[2]. Nous pouvons désormais continuer avec d'autres opérations arithmétiques de base :

```
R> 8 - 12

[1] -4

R> 14 * 25

[1] 350

R> -3/10

[1] -0.3
```

Une petite astuce très utile lorsque vous tapez des commandes directement dans la console : en utilisant les flèches *Haut* et *Bas* du clavier, vous pouvez naviguer dans l'historique des commandes tapées précédemment, que vous pouvez alors facilement réexécuter ou modifier.

Lorsqu'on fournit à R une commande incomplète, celui-ci nous propose de la compléter en nous présentant une invite de commande spéciale utilisant les signe `+`. Imaginons par exemple que nous avons malencontreusement tapé sur `Entrée` alors que nous souhaitions calculer `4*3` :

```
4 *
```

On peut alors compléter la commande en saisissant simplement `3` :

2. La présence du `[1]` en début de ligne sera expliquée par la suite, page 12.

```
R> 4 *
+ 3

[1] 12
```

Pour des commandes plus complexes, il arrive parfois qu'on se retrouve coincé avec un invite `+` sans plus savoir comment compléter la saisie correctement. On peut alors annuler la commande en utilisant la touche `Echap` ou `Esc` sous Windows. Sous Linunx on utilise le traditionnel `Control + C`.

À noter que les espaces autour des opérateurs n'ont pas d'importance lorsque l'on saisit les commandes dans R. Les trois commandes suivantes sont donc équivalentes, mais on privilégie en général la deuxième pour des raisons de lisibilité du code.

```
R> 10+2
R> 10 + 2
R> 10       +       2
```

2.2 Des objets

2.2.1 Objets simples

Faire des opérations arithmétiques, c'est bien, mais sans doute pas totalement suffisant. Notamment, on aimerait pouvoir réutiliser le résultat d'une opération sans avoir à le resaisir ou à le copier/coller.

Comme tout langage de programmation, R permet de faire cela en utilisant des *objets*. Prenons tout de suite un exemple :

```
R> x <- 2
```

Que signifie cette commande ? L'opérateur `<-` est appelé *opérateur d'assignation*. Il prend une valeur quelconque à droite et la place dans l'objet indiqué à gauche. La commande pourrait donc se lire *mettre la valeur 2 dans l'objet nommé* `x`.

On va ensuite pouvoir réutiliser cet objet dans d'autres calculs ou simplement afficher son contenu :

```
R> x + 3

[1] 5

R> x

[1] 2
```

Par défaut, si on donne à R seulement le nom d'un objet, il va se débrouiller pour nous présenter son contenu d'une manière plus ou moins lisible.

On peut utiliser autant d'objets qu'on veut. Ceux-ci peuvent contenir des nombres, des chaînes de caractères (indiquées par des guillemets droits ") et bien d'autres choses encore :

```
R> x <- 27
R> y <- 10
R> foo <- x + y
R> foo

[1] 37

R> x <- "Hello"
R> foo <- x
R> foo

[1] "Hello"
```

Les noms d'objets peuvent contenir des lettres, des chiffres (mais ils ne peuvent pas commencer par un chiffre), les symboles `.` et `_`, et doivent commencer par une lettre. R fait la différence entre les majuscules et les minuscules, ce qui signifie que `x` et `X` sont deux objets différents. On évitera également d'utiliser des caractères accentués dans les noms d'objets, et comme les espaces ne sont pas autorisés on pourra les remplacer par un point ou un tiret bas.

Enfin, signalons que certains noms courts sont réservés par R pour son usage interne et doivent être évités. On citera notamment `c, q, t, C, D, F, I, T, max, min`...

2.2.2 Vecteurs

Imaginons maintenant que nous avons interrogé dix personnes au hasard dans la rue et que nous avons relevé pour chacune d'elle sa taille en centimètres. Nous avons donc une série de dix nombres que nous souhaiterions pouvoir réunir de manière à pouvoir travailler sur l'ensemble de nos mesures.

Un ensemble de données de même nature constituent pour R un *vecteur* (en anglais *vector*) et se construit à l'aide d'un opérateur nommé `c`[3]. On l'utilise en lui donnant la liste de nos données, entre parenthèses, séparées par des virgules :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
+     163, 170)
```

Ce faisant, nous avons créé un objet nommé `tailles` et comprenant l'ensemble de nos données, que nous pouvons afficher :

```
R> tailles

 [1] 167 192 173 174 172 167 171 185 163 170
```

Dans le cas où notre vecteur serait beaucoup plus grand, et comporterait par exemple 40 tailles, on aurait le résultat suivant :

3. `c` est l'abbréviation de *combine*. Le nom de cette fonction est très court car on l'utilise très souvent.

```
R> tailles

 [1] 144 168 179 175 182 188 167 152 163 145 176 155 156 164 167 155 157 185 155 169
[21] 124 178 182 195 151 185 159 156 184 172 156 160 183 148 182 126 177 159 143 161
[41] 180 169 159 185 160
```

On a bien notre suite de quarante tailles, mais on peut remarquer la présence de nombres entre crochets au début de chaque ligne (`[1]`, `[18]` et `[35]`). En fait ces nombres entre crochets indiquent la position du premier élément de la ligne dans notre vecteur. Ainsi, le 185 en début de deuxième ligne est le 18ème élément du vecteur, tandis que le 182 de la troisième ligne est à la 35ème position.

On en déduira d'ailleurs que lorsque l'on fait :

```
R> 2

[1] 2
```

R considère en fait le nombre 2 comme un vecteur à un seul élément.

On peut appliquer des opérations arithmétiques simples directement sur des vecteurs :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
+     163, 170)
R> tailles + 20

 [1] 187 212 193 194 192 187 191 205 183 190

R> tailles/100

 [1] 1.67 1.92 1.73 1.74 1.72 1.67 1.71 1.85 1.63 1.70

R> tailles^2

 [1] 27889 36864 29929 30276 29584 27889 29241 34225 26569 28900
```

On peut aussi combiner des vecteurs entre eux. L'exemple suivant calcule l'indice de masse corporelle à partir de la taille et du poids :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
+     163, 170)
R> poids <- c(86, 74, 83, 50, 78, 66, 66, 51, 50, 55)
R> tailles.m <- tailles/100
R> imc <- poids/(tailles.m^2)
R> imc

 [1] 30.84 20.07 27.73 16.51 26.37 23.67 22.57 14.90 18.82 19.03
```

Quand on fait des opérations sur les vecteurs, il faut veiller à soit utiliser un vecteur et un chiffre (dans des opérations du type `v * 2` ou `v + 10`), soit à utiliser des vecteurs de même longueur (dans des opérations du type `u + v`).
Si on utilise des vecteurs de longueur différentes, on peut avoir quelques surprises [4].

On a vu jusque-là des vecteurs composés de nombres, mais on peut tout à fait créer des vecteurs composés de chaînes de caractères, représentant par exemple les réponses à une question ouverte ou fermée :

```
R> reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP",
+     "BEP")
```

Enfin, notons que l'on peut accéder à un élément particulier du vecteur en faisant suivre le nom du vecteur de crochets contenant le numéro de l'élément désiré. Par exemple :

```
R> reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP",
+     "BEP")
R> reponse[2]

[1] "Bac"
```

Cette opération s'appelle *l'indexation* d'un vecteur. Il s'agit ici de sa forme la plus simple, mais il en existe d'autres beaucoup plus complexes. L'indexation des vecteurs et des tableaux dans R est l'un des éléments particulièrement souples et puissants du langage (mais aussi l'un des plus délicats à comprendre et à maîtriser). Nous en reparlerons section 5.2 page 48.

2.3 Des fonctions

Nous savons désormais faire des opérations simples sur des nombres et des vecteurs, stocker ces données et résultats dans des objets pour les réutiliser par la suite.

Pour aller un peu plus loin nous allons aborder, après les *objets*, l'autre concept de base de R, à savoir les *fonctions*. Une fonction se caractérise de la manière suivante :

- elle a un nom ;
- elle accepte des arguments (qui peuvent avoir un nom ou pas) ;
- elle retourne un résultat et peut effectuer une action comme dessiner un graphique, lire un fichier, etc. ;

En fait rien de bien nouveau puisque nous avons déjà utilisé plusieurs fonctions jusqu'ici, dont la plus visible est la fonction `c`. Dans la ligne suivante :

```
R> reponse <- c("Bac+2", "Bac", "CAP", "Bac", "Bac", "CAP",
+     "BEP")
```

on fait appel à la fonction nommée `c`, on lui passe en arguments (entre parenthèses et séparées par des virgules) une série de chaînes de caractères, et elle retourne comme résultat un vecteur de chaînes de caractères, que nous stockons dans l'objet `tailles`.

Prenons tout de suite d'autres exemples de fonctions courantes :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
+     163, 170)
R> length(tailles)

[1] 10
```

4. Quand R effectue une opération avec deux vecteurs de longueurs différentes, il recopie le vecteur le plus court de manière à lui donner la même taille que le plus long, ce qui s'appelle la *règle de recyclage* (*recycling rule*). Ainsi, `c(1,2) + c(4,5,6,7,8)` vaudra l'équivalent de `c(1,2,1,2,1) + c(4,5,6,7,8)`.

```
R> mean(tailles)

[1] 173.4

R> var(tailles)

[1] 76.71
```

Ici, la fonction `length` nous renvoit le nombre d'éléments du vecteur, la fonction `mean` nous donne la moyenne des éléments du vecteur, et la fonction `var` sa variance.

2.3.1 Arguments

Les arguments de la fonction lui sont indiqués entre parenthèses, juste après son nom. En général les premiers arguments passés à la fonction sont des données servant au calcul, et les suivants des paramètres influant sur ce calcul. Ceux-ci sont en général transmis sous la forme d'argument nommés.

Reprenons l'exemple des tailles précédent :

```
R> tailles <- c(167, 192, 173, 174, 172, 167, 171, 185,
+     163, 170)
```

Imaginons que le deuxième enquêté n'ait pas voulu nous répondre. Nous avons alors dans notre vecteur une valeur manquante. Celle-ci est symbolisée dans R par le code `NA` :

```
R> tailles <- c(167, NA, 173, 174, 172, 167, 171, 185, 163,
+     170)
```

Recalculons notre taille moyenne :

```
R> mean(tailles)

[1] NA
```

Et oui, par défaut, R renvoit `NA` pour un grand nombre de calculs (dont la moyenne) lorsque les données comportent une valeur manquante. On peut cependant modifier ce comportement en fournissant un paramètre supplémentaire à la fonction `mean`, nommé `na.rm` :

```
R> mean(tailles, na.rm = TRUE)

[1] 171.3
```

Positionner le paramètre `na.rm` à `TRUE` (vrai) indique à la fonction `mean` de ne pas tenir compte des valeurs manquantes dans le calcul.

Lorsqu'on passe un argument à une fonction de cette manière, c'est-à-dire sous la forme `nom=valeur`, on parle d'*argument nommé*.

2.3.2 Quelques fonctions utiles

Récapitulons la liste des fonctions que nous avons déjà rencontrées :

Fonction	Description
`c`	construit un vecteur à partir d'une série de valeurs
`length`	nombre d'éléments d'un vecteur
`mean`	moyenne d'un vecteur de type numérique
`var`	variance d'un vecteur de type numérique
`+`, `-`, `*`, `/`	opérateurs mathématiques de base
`^`	passage à la puissance

On peut rajouter les fonctions de base suivantes :

Fonction	Description
`min`	valeur minimale d'un vecteur numérique
`max`	valeur maximale d'un vecteur numérique
`sd`	écart-type d'un vecteur numérique
`:`	génère une séquence de nombres. `1:4` équivaut à `c(1,2,3,4)`

2.3.3 Aide sur une fonction

Il est très fréquent de ne plus se rappeler quels sont les paramètres d'une fonction ou le type de résultat qu'elle retourne. Dans ce cas on peut très facilement accéder à l'aide décrivant une fonction particulière en tapant (remplacer `fonction` par le nom de la fonction) :

```
R> help("fonction")
```

Ou, de manière équivalente, `?fonction`[5].

Ces deux commandes affichent une page (en anglais) décrivant la fonction, ses paramètres, son résultat, le tout accompagné de diverses notes, références et exemples. Ces pages d'aide contiennent à peu près tout ce que vous pourrez chercher à savoir, mais elles ne sont pas toujours d'une lecture aisée.

Un autre cas très courant dans R est de ne pas se souvenir ou de ne pas connaître le nom de la fonction effectuant une tâche donnée. Dans ce cas on se reportera aux différentes manières de trouver de l'aide décrites dans l'annexe 10, page 124.

2.4 Exercices

Exercice 2.1

▷ *Solution page 136*

Construire le vecteur suivant :

```
[1] 120 134 256  12
```

Exercice 2.2

▷ *Solution page 136*

Générez les vecteurs suivants chacun de deux manières différentes :

5. L'utilisation du raccourci `?fonction` ne fonctionne pas pour certains opérateurs comme `*`. Dans ce cas on pourra utiliser `?'*'` ou bien simplement `help("*")`.

```
[1] 1 2 3 4

[1]  1  2  3  4  8  9 10 11

[1] 2 4 6 8
```

Exercice 2.3

▷ *Solution page 136*

On a demandé à 4 ménages le revenu du chef de ménage, celui de son conjoint, et le nombre de personnes du ménage :

```
R> chef <- c(1200, 1180, 1750, 2100)
R> conjoint <- c(1450, 1870, 1690, 0)
R> nb.personnes <- c(4, 2, 3, 2)
```

Calculez le revenu total par personne du ménage.

Exercice 2.4

▷ *Solution page 137*

Dans l'exercice précédent, calculez le revenu minimum et le revenu maximum parmi ceux du chef de ménage :

```
R> chef <- c(1200, 1180, 1750, 2100)
```

Recommencer avec les revenus suivants, parmi lesquels l'un des enquêtés n'a pas voulu répondre :

```
R> chef.na <- c(1200, 1180, 1750, NA)
```

Partie 3

Premier travail avec des données

3.1 Regrouper les commandes dans des scripts

Jusqu'à maintenant nous avons utilisé uniquement la console pour communiquer avec R *via* l'invite de commandes. Le principal problème de ce mode d'interaction est qu'une fois qu'une commande est tapée, elle est pour ainsi dire « perdue », c'est-à-dire qu'on doit la saisir à nouveau si on veut l'exécuter une seconde fois. L'utilisation de la console est donc restreinte aux petites commandes « jetables », le plus souvent utilisées comme test.

La plupart du temps, les commandes seront stockées dans un fichier à part, que l'on pourra facilement ouvrir, éditer et exécuter en tout ou partie si besoin. On appelle en général ce type de fichier un *script*.

Pour comprendre comment cela fonctionne, dans le menu *Fichier*, sélectionnez l'entrée *Nouveau script*[1]. Une nouvelle fenêtre (vide) apparaît. Nous pouvons désormais y saisir des commandes. Par exemple, tapez sur la première ligne la commande suivante :

```
2+2
```

Ensuite, allez dans le menu *Éditon*, et choisissez *Exécuter la ligne ou sélection*. Apparement rien ne se passe, mais si vous jetez un œil à la fenêtre de la console, les lignes suivantes ont dû faire leur apparition :

```
R> 2+2

[1] 4
```

Voici donc comment soumettre rapidement à R les commandes saisies dans votre fichier. Vous pouvez désormais l'enregistrer, l'ouvrir plus tard, et en exécuter tout ou partie. À noter que vous avez plusieurs possibilités pour soumettre des commandes à R :

- vous pouvez exécuter la ligne sur laquelle se trouve votre curseur en sélectionnant *Éditon* puis *Exécuter la ligne ou sélection*, ou plus simplement en appuyant simultanément sur les touches `Ctrl` et `R`[2] ;
- vous pouvez sélectionner plusieurs lignes contenant des commandes et les exécuter toutes en une seule fois exactement de la même manière ;
- vous pouvez exécuter d'un coup l'intégralité de votre fichier en choisissant *Édition* puis *Exécuter tout*.

1. Les indications données ici concernent l'interface par défaut de R sous Windows. Elles sont très semblables sous Mac OS X.
2. Sous Mac OS X, on utilise les touches `Pomme` et `Entrée`.

La plupart du travail sous R consistera donc à éditer un ou plusieurs fichiers de commandes et à envoyer régulièrement les commandes saisies à R en utilisant les raccourcis clavier *ad hoc*.

3.2 Ajouter des commentaires

Un commentaire est une ligne ou une portion de ligne qui sera ignorée par R. Ceci signifie qu'on peut y écrire ce qu'on veut, et qu'on va les utiliser pour ajouter tout un tas de commentaires à notre code permettant de décrire les différentes étapes du travail, les choses à se rappeler, les questions en suspens, etc.

Un commentaire sous R commence par un ou plusieurs symboles `#` (qui s'obtient avec les touches `<Alt Gr>` et `<3>` sur les claviers de type PC). Tout ce qui suit ce symbole jusqu'à la fin de la ligne est considéré comme un commentaire. On peut créer une ligne entière de commentaire, par exemple en la faisant débuter par `##` :

```
## Tableau croisé de la CSP par le nombre de livres lus
## Attention au nombre de non réponses !
```

On peut aussi créer des commentaires pour une ligne en cours :

```
x <- 2  # On met 2 dans x, parce qu'il le vaut bien
```

Dans tous les cas, il est très important de documenter ses fichiers R au fur et à mesure, faute de quoi on risque de ne plus y comprendre grand chose si on les reprend ne serait-ce que quelques semaines plus tard.

3.3 Tableaux de données

rgrs

Dans cette partie nous allons utiliser un jeu de données inclus dans l'extension rgrs. Cette extension et son installation sont décrites dans la partie B.3, page 133 [3].

Le jeu de données en question est un extrait de l'enquête *Histoire de vie* réalisée par l'INSEE en 2003. Il contient 2000 individus et 20 variables. Le descriptif des variables est indiqué dans l'annexe B.3.3, page 134.

Pour pouvoir utiliser ces données, il faut d'abord charger l'extension rgrs (après l'avoir installée, bien entendu) :

```
R> library(rgrs)
```

Puis indiquer à R que nous souhaitons accéder au jeu de données à l'aide de la commande `data` :

```
R> data(hdv2003)
```

Bien. Et maintenant, elles sont où mes données ? Et bien elles se trouvent dans un objet nommé `hdv2003` désormais accessible directement. Essayons de taper son nom à l'invite de commande :

3. À noter que les exemples de ce document ne pourront être reproduits qu'avec une version de rgrs supérieure à 0.2-11.

```
R> hdv2003
```

Le résultat (non reproduit ici) ne ressemble pas forcément à grand-chose... Il faut se rappeler que par défaut, lorsqu'on lui fournit seulement un nom d'objet, R essaye de l'afficher de la manière la meilleure (ou la moins pire) possible. La réponse à la commande `hdv2003` n'est donc rien moins que l'affichage des données brutes contenues dans cet objet.

Ce qui signifie donc que l'intégralité de notre jeu de données est inclus dans l'objet nommé `hdv2003` ! En effet, dans R, un objet peut très bien contenir un simple nombre, un vecteur ou bien le résultat d'une enquête tout entier. Dans ce cas, les objets sont appelés des *data frames*, ou tableaux de données. Ils peuvent être manipulés comme tout autre objet. Par exemple :

```
R> d <- hdv2003
```

va entraîner la copie de l'ensemble de nos données dans un nouvel objet nommé `d`, ce qui peut paraître parfaitement inutile mais a en fait l'avantage de fournir un objet avec un nom beaucoup plus court, ce qui diminuera la quantité de texte à saisir par la suite.

Résumons Comme nous avons désormais décidé de saisir nos commandes dans un script et non plus directement dans la console, les premières lignes de notre fichier de travail sur les données de l'enquête *Histoire de vie* pourraient donc ressembler à ceci :

```
## Chargement des extensions nécessaires
library(rgrs)

## Jeu de données hdv2003
data(hdv2003)
d <- hdv2003
```

3.4 Inspecter les données

3.4.1 Structure du tableau

Avant de travailler sur les données, nous allons essayer de voir à quoi elles ressemblent. Dans notre cas il s'agit de se familiariser avec la stucture du fichier. Lors de l'import de données depuis un autre logiciel, il s'agira souvent de vérifier que l'importation s'est bien déroulée.

Les fonctions `nrow`, `ncol` et `dim` donnent respectivement le nombre de lignes, le nombre de colonnes et les dimensions de notre tableau. Nous pouvons donc d'ores et déjà vérifier que nous avons bien 2000 lignes et 20 colonnes :

```
R> nrow(d)

[1] 2000

R> ncol(d)

[1] 20

R> dim(d)

[1] 2000   20
```

La fonction `names` donne les noms des colonnes de notre tableau, c'est-à-dire les noms des variables :

```
R> names(d)

 [1] "id"           "age"          "sexe"         "nivetud"      "poids"
 [6] "occup"        "qualif"       "freres.soeurs" "clso"        "relig"
[11] "trav.imp"     "trav.satisf"  "hard.rock"    "lecture.bd"   "peche.chasse"
[16] "cuisine"      "bricol"       "cinema"       "sport"        "heures.tv"
```

La fonction `str` est plus complète. Elle liste les différentes variables, indique leur type et donne le cas échéant des informations supplémentaires ainsi qu'un échantillon des premières valeurs prises par cette variable :

```
R> str(d)

'data.frame': 2000 obs. of  20 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ age          : int  28 23 59 34 71 35 60 47 20 28 ...
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 2 2 1 1 2 2 2 1 2 1 ...
 $ nivetud      : Factor w/ 8 levels "N'a jamais fait d'etudes",..: 8 NA 3 8 3 6 3 6 NA 7 ...
 $ poids        : num  2634 9738 3994 5732 4329 ...
 $ occup        : Factor w/ 7 levels "Exerce une profession",..: 1 3 1 1 4 1 6 1 3 1 ...
 $ qualif       : Factor w/ 7 levels "Ouvrier specialise",..: 6 NA 3 3 6 6 2 2 NA 7 ...
 $ freres.soeurs: int  8 2 2 1 0 5 1 5 4 2 ...
 $ clso         : Factor w/ 3 levels "Oui","Non","Ne sait pas": 1 1 2 2 1 2 1 2 1 2 ...
 $ relig        : Factor w/ 6 levels "Pratiquant regulier",..: 4 4 4 3 1 4 3 4 3 2 ...
 $ trav.imp     : Factor w/ 4 levels "Le plus important",..: 4 NA 2 3 NA 1 NA 4 NA 3 ...
 $ trav.satisf  : Factor w/ 3 levels "Satisfaction",..: 2 NA 3 1 NA 3 NA 2 NA 1 ...
 $ hard.rock    : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...
 $ lecture.bd   : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...
 $ peche.chasse : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 2 2 1 1 ...
 $ cuisine      : Factor w/ 2 levels "Non","Oui": 2 1 1 2 1 1 2 2 1 1 ...
 $ bricol       : Factor w/ 2 levels "Non","Oui": 1 1 1 2 1 1 1 2 1 1 ...
 $ cinema       : Factor w/ 2 levels "Non","Oui": 1 2 1 2 1 2 1 1 2 2 ...
 $ sport        : Factor w/ 2 levels "Non","Oui": 1 2 2 2 1 2 1 1 1 2 ...
 $ heures.tv    : num  0 1 0 2 3 2 2.9 1 2 2 ...
```

La première ligne nous informe qu'il s'agit bien d'un tableau de données avec 2000 observations et 20 variables. Vient ensuite la liste des variables. La première se nomme `id` et est de type *nombre entier* (`int`). La seconde se nomme `age` et est de type *numérique*. La troisième se nomme `sexe`, il s'agit d'un *facteur* (*factor*).

Un *facteur* et une variable pouvant prendre un nombre limité de modalités (*levels*). Ici notre variable a deux modalités possibles : `Homme` et `Femme`. Ce type de variable est décrit plus en détail section 5.1.3 page 46.

3.4.2 Inspection visuelle

La particularité de R par rapport à d'autres logiciels comme Modalisa ou SPSS est de ne pas proposer, par défaut, de vue des données sous forme de tableau. Ceci peut parfois être un peu déstabilisant dans les premiers temps d'utilisation, même si on perd vite l'habitude et qu'on finit par se rendre compte que « voir » les données n'est pas forcément un gage de productivité ou de rigueur dans le traitement.

Néanmoins, R propose une visualisation assez rudimentaire des données sous la forme d'une fenêtre de type tableur, *via* la fonction `edit` :

```
R> edit(d)
```

La fenêtre qui s'affiche permet de naviguer dans le tableau, et même d'éditer le contenu des cases et donc de modifier les données. Lorsque vous fermez la fenêtre, le contenu du tableau s'affiche dans la console : il s'agit en fait du tableau comportant les éventuelles modifications effectuées, `d` restant inchangé. Si vous souhaitez appliquer ces modifications, vous pouvez le faire en créant un nouveau tableau :

```
R> d.modif <- edit(d)
```

ou en remplaçant directement le contenu de `d`[4] :

```
R> d <- edit(d)
```

La fonction `edit` peut être utile pour un avoir un aperçu visuel des données, par contre il est **très fortement** déconseillé de l'utiliser pour modifier les données. Si on souhaite effectuer des modifications, on remonte en général aux données originales (retouches ponctuelles dans un tableur par exemple) ou on les effectue à l'aide de commandes (qui seront du coup reproductibles).

3.4.3 Accéder aux variables

`d` représente donc l'ensemble de notre tableau de données. Nous avons vu que si l'on saisit simplement `d` à l'invite de commandes, on obtient un affichage du tableau en question. Mais comment accéder aux variables, c'est à dire aux colonnes de notre tableau ?

La réponse est simple : on utilise le nom de l'objet, suivi de l'opérateur `$`, suivi du nom de la variable, comme ceci :

```
R> d$sexe

 [1] Femme Femme Homme Homme Femme Femme Femme Homme Femme Homme Femme Homme Femme
[14] Femme Femme Femme Homme Femme Homme Femme Femme Homme Femme Femme Femme Homme
[27] Femme Homme Homme Homme Homme Homme Homme Homme Homme Femme Femme Homme Femme Femme
[40] Homme Femme Homme Homme Femme Femme Homme Femme Femme Femme Femme Homme Femme
[53] Homme Femme Homme Femme Femme Femme Homme Femme Femme Homme Homme Homme Homme
[66] Femme Homme Homme Femme Femme
 [ reached getOption("max.print") -- omitted 1930 entries ]]
Levels: Homme Femme
```

On constate alors que R a bien accédé au contenu de notre variable `sexe` du tableau `d` et a affiché son contenu, c'est-à-dire l'ensemble des valeurs prises par la variable.

Les fonctions `head` et `tail` permettent d'afficher seulement les premières (respectivement les dernières) valeurs prises par la variable. On peut leur passer en argument le nombre d'éléments à afficher :

```
R> head(d$sport)

[1] Non Oui Oui Oui Non Oui
Levels: Non Oui
```

4. Dans ce cas on peut utiliser la fonction `fix` sous la forme `fix(d)`, qui est équivalente à `d <- edit(d)`.

```
R> tail(d$age, 10)

 [1] 52 42 50 41 46 45 46 24 24 66
```

A noter que ces fonctions marchent aussi pour afficher les lignes du tableau `d` :

```
R> head(d, 2)

  id age  sexe                                                nivetud poids
1  1  28 Femme Enseignement superieur y compris technique superieur  2634
2  2  23 Femme                                                   <NA>  9738
                  occup  qualif freres.soeurs clso                    relig
1 Exerce une profession Employe             8  Oui Ni croyance ni appartenance
2       Etudiant, eleve    <NA>             2  Oui Ni croyance ni appartenance
       trav.imp    trav.satisf hard.rock lecture.bd peche.chasse cuisine bricol
1 Peu important Insatisfaction       Non        Non          Non     Oui    Non
2          <NA>           <NA>       Non        Non          Non     Non    Non
  cinema sport heures.tv
1    Non   Non         0
2    Oui   Oui         1
```

3.5 Analyser une variable

3.5.1 Variable quantitative

Principaux indicateurs

Comme la fonction `str` nous l'a indiqué, notre tableau `d` contient plusieurs valeurs numériques, dont la variable `heures.tv` qui représente le nombre moyen passé par les enquêtés à regarder la télévision quotidiennement. On peut essayer de déterminer quelques caractéristiques de cette variable, en utilisant des fonctions déjà vues précédemment :

```
R> mean(d$heures.tv)

[1] NA

R> mean(d$heures.tv, na.rm = TRUE)

[1] 2.247

R> sd(d$heures.tv, na.rm = TRUE)

[1] 1.776

R> min(d$heures.tv, na.rm = TRUE)

[1] 0

R> max(d$heures.tv, na.rm = TRUE)
```

```
[1] 12

R> range(d$heures.tv, na.rm = TRUE)

[1]  0 12
```

On peut lui ajouter la fonction `median`, qui donne la valeur médiane, et le très utile `summary` qui donne toutes ces informations ou presque en une seule fois, avec en plus les valeurs des premier et troisième quartiles et le nombre de valeurs manquantes (`NA`) :

```
R> median(d$heures.tv, na.rm = TRUE)

[1] 2

R> summary(d$heures.tv)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00    1.00    2.00    2.25    3.00   12.00    5.00
```

Histogramme

Tout cela est bien pratique, mais pour pouvoir observer la distribution des valeurs d'une variable quantitative, il n'y a quand même rien de mieux qu'un bon graphique.

On peut commencer par un histogramme de la répartition des valeurs. Celui-ci peut être généré très facilement avec la fonction `hist`, comme indiqué figure 3.1 page suivante.

Ici, les options `main`, `xlab` et `ylab` permettent de personnaliser le titre du graphique, ainsi que les étiquettes des axes. De nombreuses autres options existent pour personnaliser l'histogramme, parmi celles-ci on notera :

probability si elle vaut `TRUE`, l'histogramme indique la proportion des classes de valeurs au lieu des effectifs.

breaks permet de contrôler les classes de valeurs. On peut lui passer un chiffre, qui indiquera alors le nombre de classes, un vecteur, qui indique alors les limites des différentes classes, ou encore une chaîne de caractère ou une fonction indiquant comment les classes doivent être calculées.

col la couleur de l'histogramme [5].

Deux exemples sont donnés figure 3.2 page 25 et figure 3.3 page 26.

Voir la page d'aide de la fonction `hist` pour plus de détails sur les différentes options.

Boîtes à moustaches

Les boîtes à moustaches, ou `boxplot` en anglais, sont une autre représentation graphique de la répartition des valeurs d'une variable quantitative. Elles sont particulièrement utiles pour comparer les distributions de plusieurs variables ou d'une même variable entre différents groupes, mais peuvent aussi être utilisées pour représenter la dispersion d'une unique variable. La fonction qui produit ces graphiques est la fonction `boxplot`. On trouvera un exemple figure 3.4 page 27.

Comment interpréter ce graphique ? On le comprendra mieux à partir de la figure 3.5 page 28 [6].

5. Il existe un grand nombre de couleurs prédéfinies dans R. On peut récupérer leur liste en utilisant la fonction `colors`

```
R> hist(d$heures.tv, main = "Nombre d'heures passées devant la télé par jour",
+     xlab = "Heures", ylab = "Effectif")
```

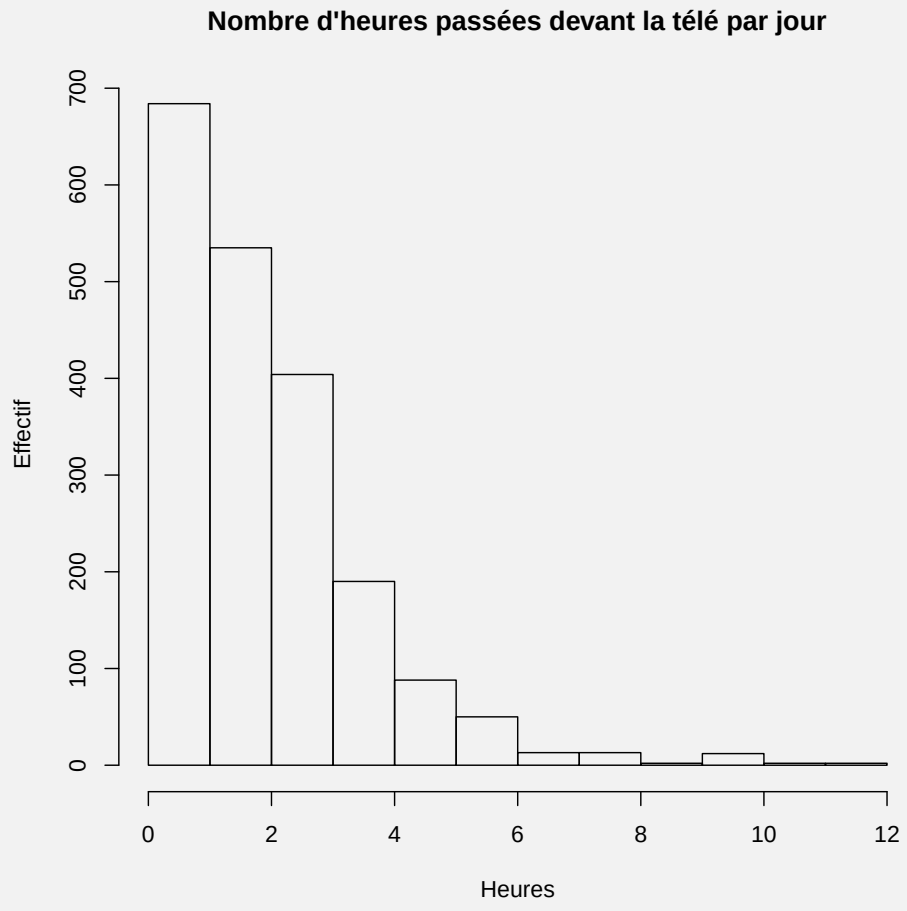


FIGURE 3.1 – Exemple d'histogramme

```
R> hist(d$heures.tv, main = "Heures de télé en 7 classes",
+     breaks = 7, xlab = "Heures", ylab = "Proportion", probability = TRUE,
+     col = "orange")
```

FIGURE 3.2 – Un autre exemple d'histogramme

```
R> hist(d$heures.tv, main = "Heures de télé avec classes spécifiées",
+     breaks = c(0, 1, 4, 9, 12), xlab = "Heures", ylab = "Proportion",
+     col = "red")
```

FIGURE 3.3 – Encore un autre exemple d'histogramme

```
R> boxplot(d$heures.tv, main = "Nombre d'heures passées devant la télé par jour",
+     ylab = "Heures")
```

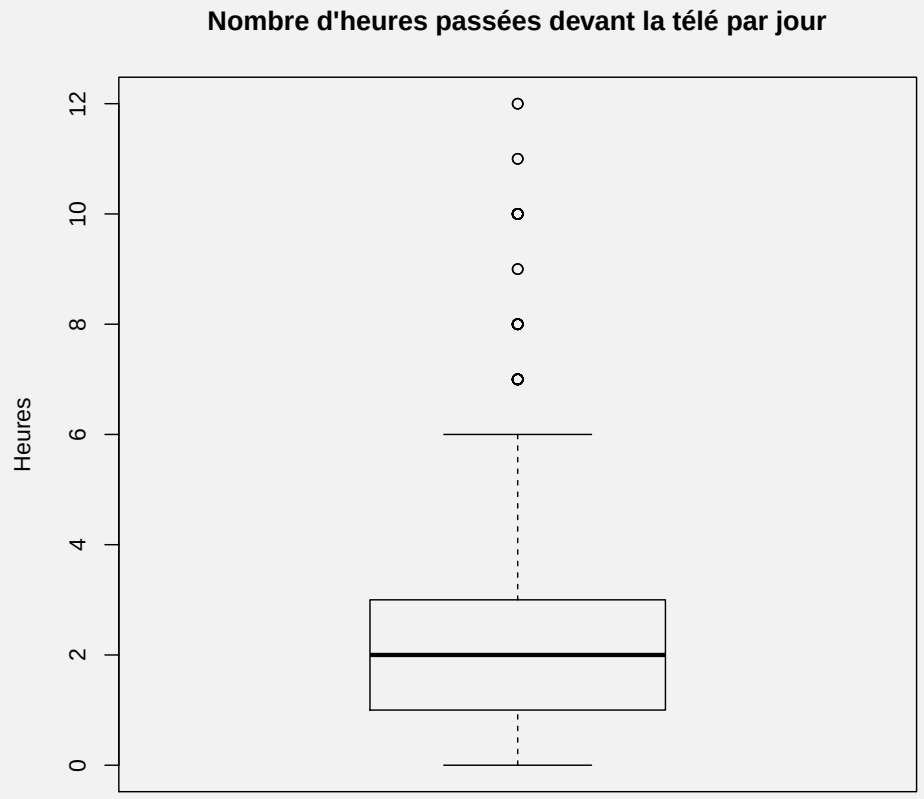


FIGURE 3.4 – Exemple de boîte à moustaches

```
R> boxplot(d$heures.tv, col = grey(0.8), main = "Nombre d'heures passées devant la
télé par jour",
+     ylab = "Heures")
R> abline(h = median(d$heures.tv, na.rm = TRUE), col = "navy",
+     lty = 2)
R> text(1.35, median(d$heures.tv, na.rm = TRUE) + 0.15,
+     "Médiane", col = "navy")
R> Q1 <- quantile(d$heures.tv, probs = 0.25, na.rm = TRUE)
R> abline(h = Q1, col = "darkred")
R> text(1.35, Q1 + 0.15, "Q1 : premier quartile", col = "darkred",
+     lty = 2)
R> Q3 <- quantile(d$heures.tv, probs = 0.75, na.rm = TRUE)
R> abline(h = Q3, col = "darkred")
R> text(1.35, Q3 + 0.15, "Q3 : troisième quartile", col = "darkred",
+     lty = 2)
R> arrows(x0 = 0.7, y0 = quantile(d$heures.tv, probs = 0.75,
+     na.rm = TRUE), x1 = 0.7, y1 = quantile(d$heures.tv, probs = 0.25,
+     na.rm = TRUE), length = 0.1, code = 3)
R> text(0.7, Q1 + (Q3 - Q1)/2 + 0.15, "h", pos = 2)
R> mtext("L'écart inter-quartile h contient 50 % des individus",
+     side = 1)
R> abline(h = Q1 - 1.5 * (Q3 - Q1), col = "darkgreen")
R> text(1.35, Q1 - 1.5 * (Q3 - Q1) + 0.15, "Q1 -1.5 h",
+     col = "darkgreen", lty = 2)
R> abline(h = Q3 + 1.5 * (Q3 - Q1), col = "darkgreen")
R> text(1.35, Q3 + 1.5 * (Q3 - Q1) + 0.15, "Q3 +1.5 h",
+     col = "darkgreen", lty = 2)
```

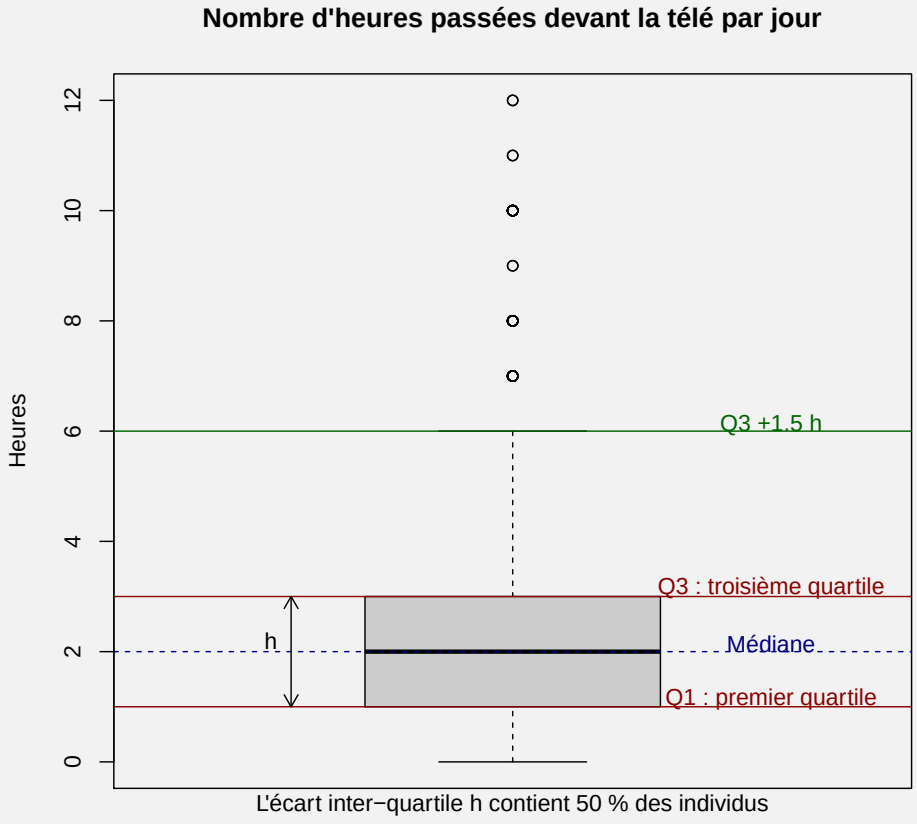


FIGURE 3.5 – Interprétation d'une boîte à moustaches

```
R> boxplot(d$heures.tv, main = "Nombre d'heures passées devant la télé par\njour",
+     ylab = "Heures")
R> rug(d$heures.tv, side = 2)
```

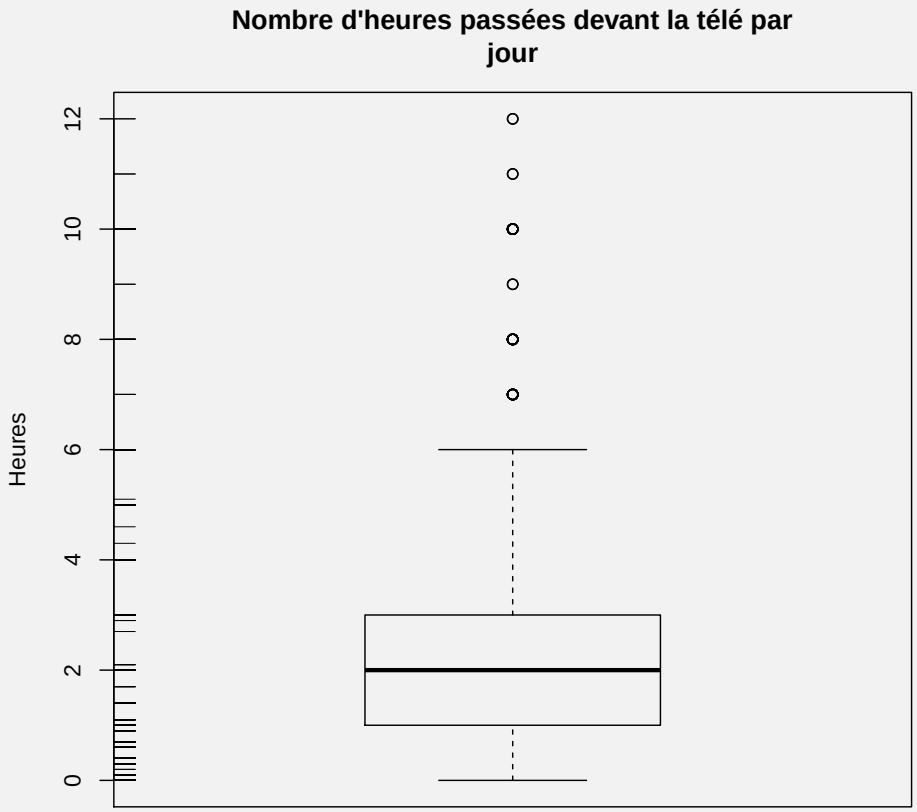


FIGURE 3.6 – Boîte à moustaches avec représentation des valeurs

Le carré au centre du graphique est délimité par les premiers et troisième quartiles, avec la médiane représentée par une ligne plus sombre au milieu. Les « fourchettes » s'étendant de part et d'autres vont soit jusqu'à la valeur minimale ou maximale, soit jusqu'à une valeur approximativement égale au quartile le plus proche plus 1,5 fois l'écart inter-quartile. Les points se situant en-dehors de cette fourchette sont représentés par des petits ronds et sont généralement considérés comme des valeurs extrêmes, potentiellement aberrantes.

On peut ajouter la représentation des valeurs sur le graphique pour en faciliter la lecture avec des petits traits dessinés sur l'axe vertical (fonction `rug`), comme sur la figure 3.6 de la présente page.

---

en tapant simplement `colors()` dans la console, ou en consultant le document suivant : http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

6. Le code ayant servi à générer cette figure est une copie quasi conforme de celui présenté dans l'excellent document de Jean Lobry sur les graphiques de base avec R, téléchargeable sur le site du Pôle bioinformatique lyonnais : http://pbil.univ-lyon1.fr/R/pdf/lang04.pdf.

3.5.2 Variable qualitative

Tris à plat

La fonction la plus utilisée pour le traitement et l'analyse des variables qualitatives (variable prenant ses valeurs dans un ensemble de modalités) est sans aucun doute la fonction `table`, qui donne les effectifs de chaque modalité de la variable.

```
R> table(d$sexe)


Homme Femme
  899  1101
```

La tableau précédent nous indique que parmi nos enquêtés on trouve 894 hommes et 1106 femmes.

Quand le nombre de modalités est élevé, on peut ordonner le tri à plat selon les effectifs à l'aide de la fonction `sort`.

```
R> table(d$occup)


Exerce une profession               Chomeur       Etudiant, eleve
                 1049                   134                    94
             Retraite   Retire des affaires              Au foyer
                  392                    77                   171
        Autre inactif
                   83

R> sort(table(d$occup))


  Retire des affaires         Autre inactif       Etudiant, eleve
                   77                    83                    94
              Chomeur              Au foyer              Retraite
                  134                   171                   392
Exerce une profession
                 1049

R> sort(table(d$occup), decreasing = TRUE)


Exerce une profession              Retraite              Au foyer
                 1049                   392                   171
              Chomeur       Etudiant, eleve         Autre inactif
                  134                    94                    83
  Retire des affaires
                   77
```

À noter que la fonction `table` exclut par défaut les non-réponses du tableau résultat. L'argument `useNA` de cette fonction permet de modifier ce comportement :

- avec `useNA="no"` (valeur par défaut), les valeurs manquantes ne sont jamais incluses dans le tri à plat ;

– avec `useNA="ifany"`, une colonne `NA` est ajoutée si des valeurs manquantes sont présentes dans les données ;
– avec `useNA="always"`, une colonne `NA` est toujours ajoutée, même s'il n'y a pas de valeurs manquantes dans les données.

On peut donc utiliser :

```
R> table(d$trav.satisf, useNA = "ifany")


  Satisfaction Insatisfaction      Equilibre           <NA>
           480            117            451            952
```

L'utilisation de `summary` permet également l'affichage du tri à plat et du nombre de non-réponses :

```
R> summary(d$trav.satisf)

  Satisfaction Insatisfaction      Equilibre           NA's
           480            117            451            952
```

Pour obtenir un tableau avec la répartition en pourcentages, on peut utiliser la fonction `freq` de l'extension `rgrs`. *rgrs*

```
R> freq(d$qualif)

                           n    %
Ouvrier specialise       203 10.2
Ouvrier qualifie         292 14.6
Technicien                86  4.3
Profession intermediaire 160  8.0
Cadre                    260 13.0
Employe                  594 29.7
Autre                     58  2.9
NA                       347 17.3
```

La colonne `n` donne les effectifs bruts, et la colonne `%` la répartition en pourcentages. La fonction accepte plusieurs paramètres permettant d'afficher les totaux, les pourcentages cumulés, de trierselon les effectifs ou de contrôler l'affichage. Par exemple :

```
R> freq(d$qualif, cum = TRUE, total = TRUE, sort = "inc",
+     digits = 2, exclude = NA)

                            n      %   %cum
Autre                      58   3.51   3.51
Technicien                 86   5.20   8.71
Profession intermediaire  160   9.68  18.39
Ouvrier specialise        203  12.28  30.67
Cadre                     260  15.73  46.40
Ouvrier qualifie          292  17.66  64.07
Employe                   594  35.93 100.00
Total                    1653 100.00 100.00
```

La colonne `%cum` indique ici le pourcentage cumulé, ce qui est ici une très mauvaise idée puisque pour ce type de variable cela n'a aucun sens. Les lignes du tableau résultat ont été triés par effectifs croissants, les totaux ont été ajoutés, les non-réponses exclues, et les pourcentages arrondis à deux décimales.

Pour plus d'informations sur la commande `freq`, consultez sa page d'aide en ligne avec `?freq` ou `help("freq")`.

Représentation graphique

Pour représenter la répartition des effectifs parmi les modalités d'une variable qualitative, on a souvent tendance à utiliser des diagrammes en secteurs (camemberts). Ceci est possible sous R avec la fonction `pie`, mais la page d'aide de ladite fonction nous le déconseille assez vivement : les diagrammes en secteur sont en effet une mauvaise manière de présenter ce type d'information, car l'œil humain préfère comparer des longueurs plutôt que des surfaces [7].

On privilégiera donc d'autres formes de représentations, à savoir les diagrammes en bâtons et les diagrammes de Cleveland.

Les diagrammes en bâtons sont utilisés automatiquement par R lorsqu'on applique la fonction générique `plot` à un tri à plat obtenu avec `table`. On privilégiera cependant ce type de représentations pour les variables de type numérique comportant un nombre fini de valeurs. Le nombre de frères, sœurs, demi-frères et demi-sœurs est un bon exemple, indiqué figure 3.7 page ci-contre.

Pour les autres types de variables qualitatives, on privilégiera les diagrammes de Cleveland, obtenus avec la fonction `dotchart`. On doit appliquer cette fonction au tri à plat de la variable, obtenu avec la fonction `table` [8]. Le résultat se trouve figure 3.8 page 34.

Quand la variable comprend un grand nombre de modalités, il est préférable d'ordonner le tri à plat obtenu à l'aide de la fonction `sort` (voir figure 3.9 page 35).

3.6 Exercices

Exercice 3.5

▷ *Solution page 137*

Créer un script qui effectue les actions suvantes et exécutez-le :
- charger l'extension rgrs
- charger le jeu de données `hdv2003`
- placer le jeu de données dans un objet nommé `df`
- afficher la liste des variables de `df` et leur type

Exercice 3.6

▷ *Solution page 138*

Des erreurs se sont produites lors de la saisie des données de l'enquête. En fait le premier individu du jeu de données n'a pas 42 ans mais seulement 24, et le second individu n'est pas un homme mais une femme. Corrigez les erreurs et stockez les données corrigées dans un objet nommé `df.ok`.

Affichez ensuite les 4 premières lignes de `df.ok` pour vérifier que les modifications ont bien été prises en compte.

Exercice 3.7

▷ *Solution page 138*

Nous souhaitons étudier la répartition des âges des enquêtés (variable `age`). Pour cela, affichez les principaux indicateurs de cette variable. Représentez ensuite sa distribution par un histogramme en 10 classes, puis sous forme de boîte à moustache, et enfin sous la forme d'un diagramme en bâtons représentant les effectifs de chaque âge.

7. On trouvera des exemples illustrant cette idée dans le document de Jean Lobry cité précédemment.
8. Pour des raisons liées au fonctionnement interne de la fonction `dotchart`, on doit l'appliquer à la transposition du tri à plat obtenu, d'où l'appel à la fonction `t`.

```
R> plot(table(d$freres.soeurs), main = "Nombre de frères, soeurs, demi-frères et
demi-soeurs",
+     ylab = "Effectif")
```

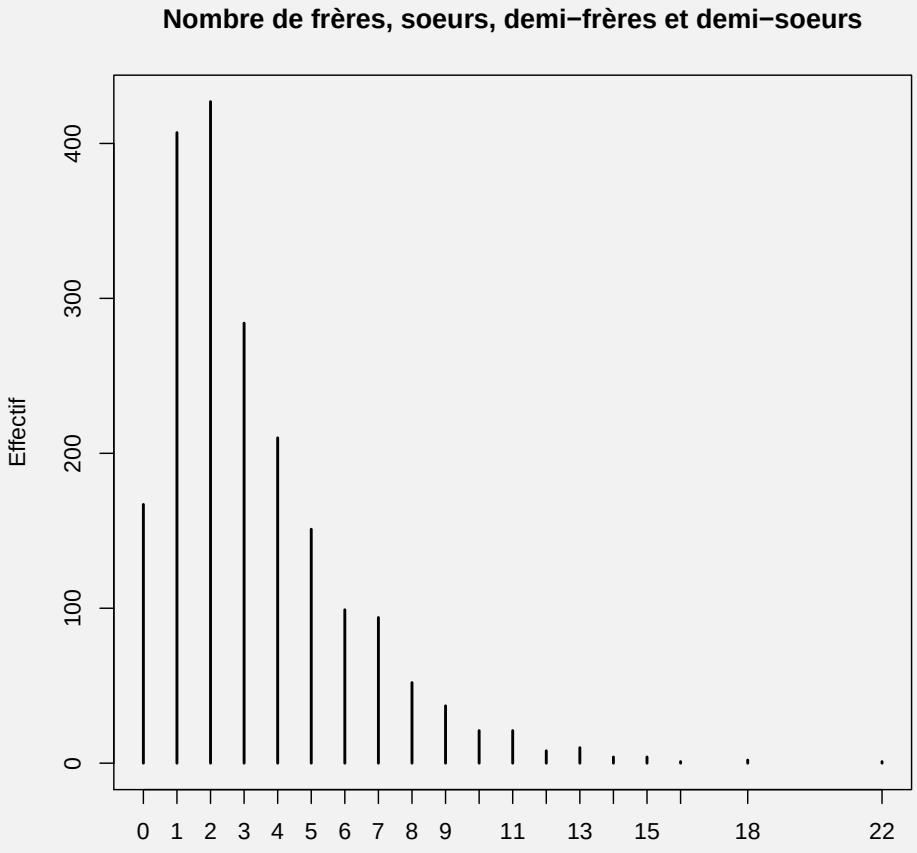


FIGURE 3.7 – Exemple de diagramme en bâtons

```
R> dotchart(t(table(d$clso)), main = "Sentiment d'appartenance à une classe sociale",
+     pch = 19)
```

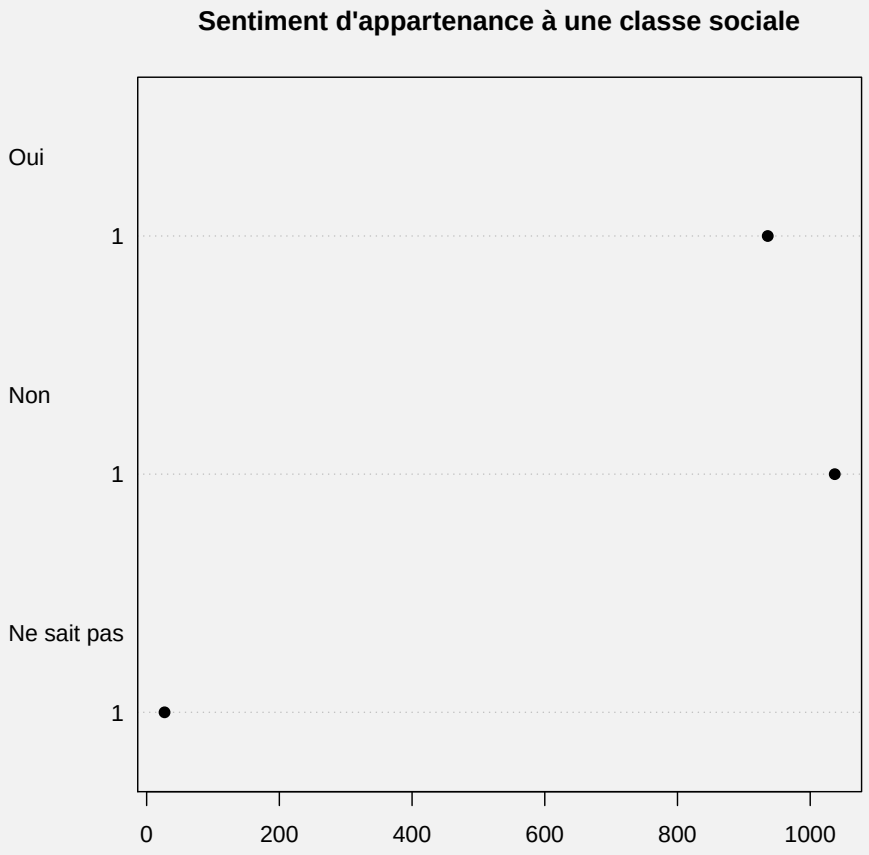


FIGURE 3.8 – Exemple de diagramme de Cleveland

```
R> dotchart(sort(table(d$qualif)), main = "Niveau de qualification")

Warning message: 'x' is neither a vector nor a matrix: using as.numeric(x)
```

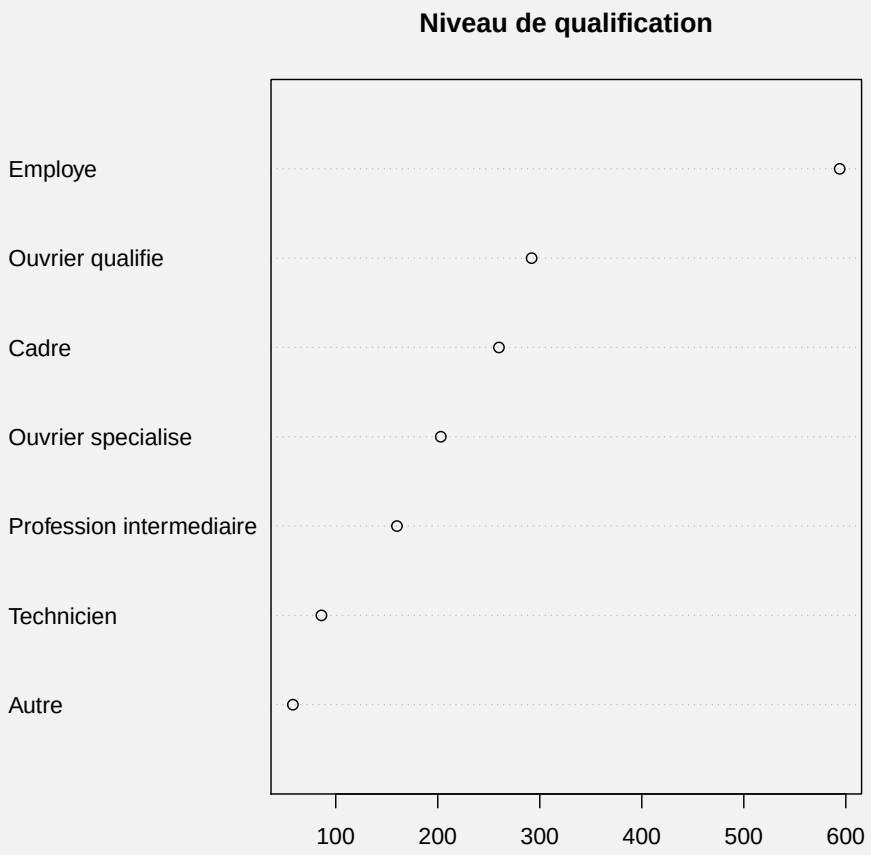


FIGURE 3.9 – Exemple de diagramme de Cleveland ordonné

Exercice 3.8

▷ *Solution page 138*

On s'intéresse maintenant à l'importance accordée par les enquêtés à leur travail (variable `trav.imp`). Faites un tri à plat des effectifs des modalités de cette variable avec la commande `table`. Y'a-t-il des valeurs manquantes ?

Faites un tri à plat affichant à la fois les effectifs et les pourcentages de chaque modalité.

Représentez graphiquement les effectifs des modalités à l'aide d'un diagramme de Cleveland.

Partie 4

Import/export de données

L'import et l'export de données depuis ou vers d'autres applications est couvert en détail dans l'un des manuels officiels (en anglais) nommé *R Data Import/Export* et accessible, comme les autres manuels, à l'adresse suivante :

http://cran.r-project.org/manuals.html

Cette partie est très largement tirée de ce document, et on pourra s'y reporter pour plus de détails.

Importer des données est souvent l'une des première opérations que l'on effectue lorsque l'on débute sous R, et ce n'est pas la moins compliquée. En cas de problème il ne faut donc pas hésiter à demander de l'aide par les différents moyens disponibles (voir partie 10 page 124) avant de se décourager.

Un des points délicats pour l'importation de données dans R concerne le nom des variables. Pour être utilisables dans R ceux-ci doivent être à la fois courts et explicites, ce qui n'est pas le cas dans d'autres applications comme Modalisa par exemple. La plupart des fonctions d'importation s'occupent de convertir les noms de manières à ce qu'ils soient compatibles avec les règles de R (remplacement des espaces par des points par exemple), mais un renommage est souvent à prévoir, soit au sein de l'application d'origine, soit une fois les données importées dans R.

4.1 Accès aux fichiers et répertoire de travail

Dans ce qui suit, puisqu'il s'agit d'importer des données externes, nous allons avoir besoin d'accéder à des fichiers situés sur le disque dur de notre ordinateur.

Par exemple, la fonction `read.table`, très utilisée pour l'import de fichiers texte, prend comme premier argument le nom du fichier à importer, ici `fichier.txt` :

```
R> donnees <- read.table("fichier.txt")
```

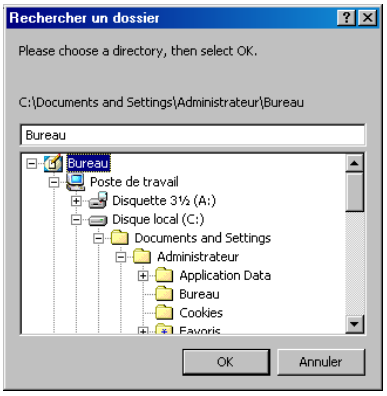


FIGURE 4.1 – Sélection du répertoire de travail avec `selectwd`

Cependant, ceci ne fonctionnera que si le fichier se trouve dans le *répertoire de travail* de R. De quoi s'agit-il ? Tout simplement du répertoire dans lequel R est actuellement en train de s'exécuter. Pour savoir quel est le répertoire de travail actuel, on peut utiliser la fonction `getwd` [1] :

```
R> getwd()

[1] "/home/julien/r/doc/intro"
```

Si on veut modifier le répertoire de travail, on utilise `setwd` en lui indiquant le chemin complet. Par exemple sous Linux :

```
R> setwd("/home/julien/projets/R")
```

Sous Windows le chemin du répertoire est souvent un peu plus compliqué. Vous pouvez alors utiliser
rgrs la fonction `selectwd` de l'extension rgrs [2] en tapant simplement :

```
R> selectwd()
```

Une boîte de dialogue devrait alors s'afficher vous permettant de sélectionner un répertoire sur votre disque. Sous Windows elle devrait ressembler à celle de la figure 4.1 de la présente page.

Sélectionnez le répertoire de travail de votre session R est cliquez sur *Ok* [3]. Vous devriez voir s'afficher le message suivant :

```
Nouveau repertoire de travail : C:/Documents and Settings/Bureau
Pour automatiser ce changement dans un script, utilisez :
setwd("C:/Documents and Settings/Bureau")
```

Si vous travaillez en ligne de commande dans la console, le répertoire de travail a été mis à jour. Si vous travaillez dans un script, il peut être intéressant de rajouter la ligne `setwd` indiquée précédemment au début de votre script pour automatiser cette opération.

1. Le résultat indiqué ici correspond à un système Linux, sous Windows vous devriez avoir quelque chose de la forme `C:/Documents and Settings/` ...
2. Sous Windows, si vous utilisez l'interface graphique par défaut, vous pouvez aussi utiliser la fonction *Changer le répertoire courant* du menu *Fichier*
3. Sous Windows, si vous ne retrouvez pas votre répertoire *Mes documents*, celui-ci se trouve en général dans le répertoire portant le nom de votre utilisateur situé dans le répertoire *Documents and Settings* du lecteur `{C:`. Par exemple `C:\Documents and Settings\Administrateur\Mes Documents\`

Une fois le répertoire de travail fixé, on pourra accéder aux fichiers qui s'y trouvent directement, en spécifiant seulement leur nom. On peut aussi créer des sous-répertoires dans le répertoire de travail ; une potentielle bonne pratique peut être de regrouper tous les fichiers de données dans un sous-répertoire nommé `donnees`. On pourra alors accéder aux fichiers qui s'y trouvent de la manière suivante :

```
R> donnees <- read.table("donnees/fichier.txt")
```

Dans ce qui suit on supposera que les fichiers à importer se trouvent directement dans le répertoire de travail, et on n'indiquera donc que le nom du fichier, sans indication de chemin ou de répertoire supplémentaire.

Si vous utilisez l'environnement de développement RStudio (voir section A.5 page 131), vous pouvez vous débarasser du problème des répertoires de travail en utilisant sa fonctionnalité de gestion de *projets*.

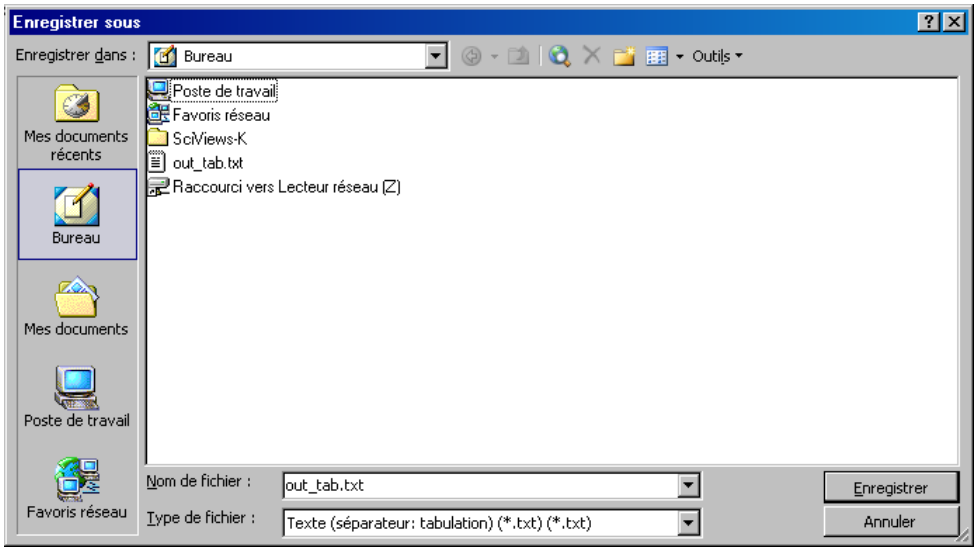
4.2 Import de données depuis un tableur

Il est assez courant de vouloir importer des données saisies ou traitées avec un tableur du type OpenOffice/LibreOffice ou Excel. En général les données prennent alors la forme d'un tableau avec les variables en colonne et les individus en ligne.

	A	B	C	D
1	**Country or Area**	**Year**	**Educational levels**	**Value**
2	Afghanistan	2002	Primary level	3266737
3	Afghanistan	2001	Primary level	773623
4	Afghanistan	2001	Secondary level	362415
5	Afghanistan	2000	Primary level	500068
6	Afghanistan	1999	Primary level	957403
7	Afghanistan	1998	Primary level	1046338
8	Afghanistan	1995	Primary level	1312197
9	Afghanistan	1995	Secondary level	512851
10	Afghanistan	1994	Primary level	1161444
11	Afghanistan	1994	Secondary level	497762
12	Afghanistan	1993	Primary level	786532
13	Afghanistan	1993	Secondary level	332170
14	Afghanistan	1991	Primary level	627888
15	Afghanistan	1991	Secondary level	281928
16	Afghanistan	1990	Primary level	622513
17	Afghanistan	1990	Secondary level	182340

4.2.1 Depuis Excel

La démarche pour importer ces données dans R est d'abord de les enregistrer dans un format de type texte. Sous Excel, on peut ainsi sélectionner *Fichier*, *Enregistrer sous*, puis dans la zone *Type de fichier* choisir soit *Texte (séparateur tabulation)*, soit *CSV (séparateur : point-virgule)*.



Dans le premier cas, on peut importer le fichier en utilisant la fonction `read.delim2`, de la manière suivante :

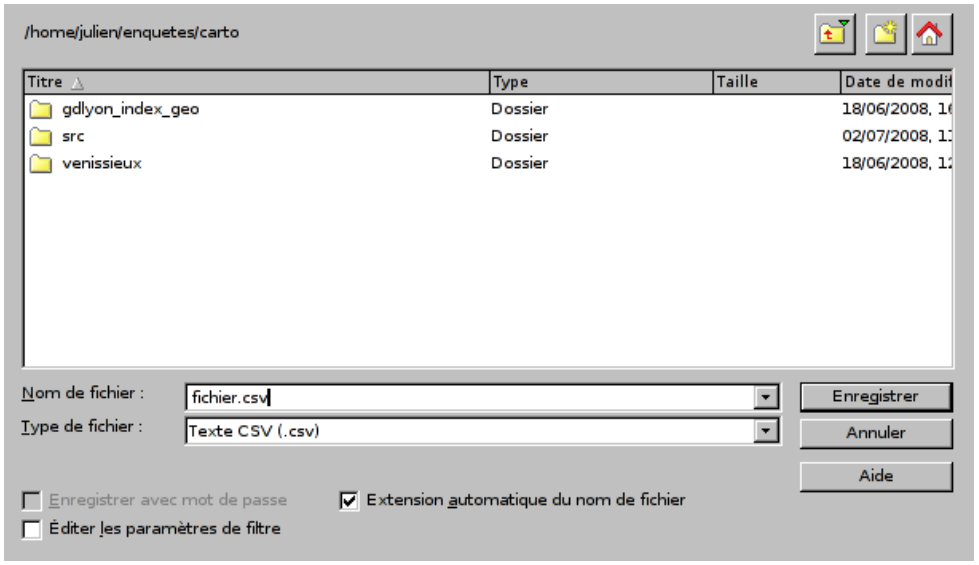
```
R> donnees <- read.delim2("fichier.txt")
```

Dans le second cas, on utilise `read.csv2`, de la même manière :

```
R> donnees <- read.csv2("fichier.csv")
```

4.2.2 Depuis OpenOffice ou LibreOffice

Depuis OpenOffice on procédera de la même manière, en sélectionnant le type de fichier *Texte CSV*.



On importe ensuite les données dans R à l'aide de la fonction `read.csv` :

```
R> read.csv("fichier.csv", dec = ",")
```

4.2.3 Autres sources / en cas de problèmes

Les fonctions `read.csv` et compagnie sont en fait des dérivées de la fonction plus générique `read.table`. Celle-ci contient de nombreuses options permettant d'adapter l'import au format du fichier texte. On pourra se reporter à la page d'aide de `read.table` si on recontre des problèmes ou si on souhaite importer des fichiers d'autres sources.

Parmi les options disponibles, on citera notamment :

`header` indique si la première ligne du fichier contient les noms des variables (valeur `TRUE`) ou non (valeur `FALSE`).

`sep` indique le caractère séparant les champs. En général soit une virgule, soit un point-virgule, soit une tabulation. Pour cette dernière l'option est `sep="\t"`.

`quote` indique le caractère utilisé pour délimiter les champs. En général on utilise soit des guillemets doubles (`quote="\""`) soit rien du tout (`quote=""`).

`dec` indique quel est le caractère utilisé pour séparer les nombres et leurs décimales. Il s'agit le plus souvent de la virgule lorsque les données sont en français (`dec=","`), et le point pour les données anglophones (`dec="."`).

D'autres options sont disponibles, pour gérer le format d'encodage du fichier source ou de nombreux autres paramètres d'importation. On se réfèrera alors à la page d'aide de `read.table` et à la section *Spreadsheet-like data* de *R Data Import/Export* :

http://cran.r-project.org/doc/manuals/R-data.html#Spreadsheet_002dlike-data

4.3 Import depuis d'autres logiciels

La plupart des fonctions permettant l'import de fichiers de données issus d'autres logiciels font partie d'une extension nommée foreign, présente à l'installation de R mais qu'il est nécessaire de charger en mémoire avant utilisation avec l'instruction :

```
R> library(foreign)
```

4.3.1 SAS

Les fichiers au format SAS se présentent en général sous deux format : format SAS export (extension `.xport` ou `.xpt`) ou format SAS natif (extension `.sas7bdat`).

R peut lire directement les fichiers au format export *via* la fonction `read.xport` de l'extension foreign.

Celle-ci s'utilise très simplement, en lui passant le nom du fichier en argument :

```
R> donnees <- read.xport("fichier.xpt")
```

En ce qui concerne les fichiers au format SAS natif, il existe des fonctions permettant de les importer, mais elles nécessitent d'avoir une installation de SAS fonctionnelle sur sa machine (il s'agit des fonctions `read.ssd` de l'extension foreign, et `sas.get` de l'extension Hmisc).

Si on ne dispose que des fichiers au format SAS natif, le plus simple est d'utiliser l'application SAS System Viewer, qui permet de lire des fichiers SAS natif, de les visualiser et de les enregistrer dans un format texte. Cette application est téléchargeable gratuitement, mais ne fonctionne que sous Windows[4] :

http://www.sas.com/apps/demosdownloads/setupcat.jsp?cat=SAS+System+Viewer

Une fois le fichier de données au format SAS natif ouvert on peut l'enregistrer au format texte tabulé. L'import dans R se fait alors avec la commande suivante :

```
R> donnees <- read.delim("fichier.txt", na.strings = ".")
```

4.3.2 SPSS

Les fichiers générés par SPSS sont accessibles depuis R avec la fonction `read.spss` de l'extension `foreign`. Celle-ci peut lire aussi bien les fichiers sauvegardés avec la fonction *Enregistrer* que ceux générés par la fonction *Exporter*.

La syntaxe est également très simple :

```
R> donnees <- read.spss("fichier.sav")
```

Plusieurs options permettant de contrôler l'importation des données sont disponibles. On se reportera à la page d'aide de la fonction pour plus d'informations.

4.3.3 Modalisa

rgrs

L'extension rgrs fournit plusieurs fonctions pour l'import ou l'export de données depuis ou vers Modalisa et pour leur traitement, en particulier concernant les questions à réponses multiples.

L'import de données permet de récupérer des sauvegardes au format ASCII et s'appuie sur la fonction `mls.import`.

On trouvera davantage d'informations à l'adresse suivante :

http://alea.fr.eu.org/pages/R-et-Modalisa

4.3.4 Fichiers `dbf`

L'Insee diffuse ses fichiers détails depuis son site Web au format dBase (extension `.dbf`). Ceux-ci sont directement lisibles dans R avec la fonction `read.dbf` de l'extension `foreign`.

```
R> donnees <- read.dbf("fichier.dbf")
```

La principale limitation des fichiers `dbf` est de ne pas gérer plus de 256 colonnes. Les tables des enquêtes de l'Insee sont donc parfois découpées en plusieurs fichiers `dbf` qu'il convient de fusionner avec la fonction `merge`. L'utilisation de cette fonction est détaillée dans la section 5.6 page 69.

4.4 Autres sources

R offre de très nombreuses autres possibilités pour accéder aux données. Il est ainsi possible d'importer des données depuis d'autres applications qui n'ont pas été évoquées (Stata, S-Plus, etc.), de se connecter

4. Ou sous Linux et Mac OS X avec wine.

à un système de base de données relationelle type MySql, de lire des données *via* ODBC ou des connexions réseau, etc.

Pour plus d'informations on consultera le manuel *R Data Import/Export* :

http://cran.r-project.org/manuals.html

4.5 Exporter des données

R propose également différentes fonctions permettant d'exporter des données vers des formats variés.

- `write.table` est l'équivalent de `read.table` et permet d'enregistrer des tableaux de données au format texte, avec de nombreuses options ;
- `write.foreign`, de l'extension `foreign`, permet d'exporter des données aux formats SAS, SPSS ou Stata ;
- `write.dbf`, de l'extension `foreign`, permet d'exporter des données au format dBase ;
- `mls.export`, de l'extension rgrs, permet d'exporter des données à destination de Modalisa ; *rgrs*
- `save` permet d'enregistrer des objets R sur le disque pour récupération ultérieure ou sur un autre système.

À nouveau, pour plus de détails on se référera aux pages d'aide de ces fonctions et au manuel *R Data Import/Export*.

4.6 Exercices

Exercice 4.9

▷ *Solution page 138*

Saisissez quelques données fictives dans une application de type tableur, enregistrez-les dans un format texte et importez-les dans R.

Vérifiez que l'importation s'est bien déroulée.

Exercice 4.10

▷ *Solution page 139*

L'adresse suivante permet de télécharger un fichier au format dBase contenant une partie des données de l'enquête *EPCV Vie associative* de l'INSEE (2002) :

http://telechargement.insee.fr/fichiersdetail/epcv1002/dbase/epcv1002_BENEVOLAT_dbase.zip

Téléchargez le fichier, décompressez-le et importez les données dans R.

Partie 5

Manipulation de données

Cette partie est un peu aride et pas forcément très intuitive. Elle aborde cependant la base de tous les traitements et manipulation de données sous R, et mérite donc qu'on s'y arrête un moment, ou qu'on y revienne un peu plus tard en cas de saturation...

5.1 Variables

Le type d'objet utilisé par R pour stocker des tableaux de données s'appelle un *data frame*. Celui-ci comporte des observations en ligne et des variables en colonnes. On accède aux variables d'un *data frame* avec l'opérateur `$`.

Dans ce qui suit on travaillera sur le jeu de données tiré de l'enquête *Histoire de vie*, fourni avec l'extension rgrs et décrit dans l'annexe B.3.3, page 134.

```
R> library(rgrs)
R> data(hdv2003)
R> d <- hdv2003
```

Mais aussi sur le jeu de données tiré du recensement 1999, décrit page 135 :

```
R> data(rp99)
```

5.1.1 Types de variables

On peut considérer qu'il existe quatre type de variables dans R :

- les variables **numériques**, ou quantitatives ;
- les **facteurs**, qui prennent leurs valeurs dans un ensemble défini de modalités. Elles correspondent en général aux questions fermées d'un questionnaire ;
- les variables **caractères**, qui contiennent des chaînes de caractères plus ou moins longues. On les utilise pour les questions ouvertes ou les champs libres ;
- les variables **booléennes**, qui ne peuvent prendre que la valeur *vrai* (`TRUE`) ou *faux* (`FALSE`). On les utilise dans R pour les calculs et les recodages.

Pour connaître le type d'une variable donnée, on peut utiliser la fonction `class`.

Résultat de `class`	Type de variable
`factor`	Facteur
`integer`	Numérique
`double`	Numérique
`numeric`	Numérique
`character`	Caractères
`logical`	Booléenne

```
R> class(d$age)

[1] "integer"

R> class(d$sexe)

[1] "factor"

R> class(c(TRUE, TRUE, FALSE))

[1] "logical"
```

La fonction `str` permet également d'avoir un listing de toutes les variables d'un tableau de données et indique le type de chacune d'elle.

5.1.2 Renommer des variables

Une opération courante lorsqu'on a importé des variables depuis une source de données externe consiste à renommer les variables importées. Sous R les noms de variables doivent être à la fois courts et explicites tout en obéissant à certaines règles décrites dans la remarque page 11.

On peut lister les noms des variables d'un *data frame* à l'aide de la fonction `names` :

```
R> names(d)

 [1] "id"           "age"          "sexe"          "nivetud"      "poids"
 [6] "occup"        "qualif"       "freres.soeurs" "clso"         "relig"
[11] "trav.imp"     "trav.satisf"  "hard.rock"     "lecture.bd"   "peche.chasse"
[16] "cuisine"      "bricol"       "cinema"        "sport"        "heures.tv"
```

Cette fonction peut également être utilisée pour renommer l'ensemble des variables. Si par exemple on souhaitait passer les noms de toutes les variables en majuscules, on pourrait faire :

```
R> d.maj <- d
R> names(d.maj) <- c("ID", "AGE", "SEXE", "NIVETUD", "POIDS",
+     "OCCUP", "QUALIF", "FRERES.SOEURS", "CLSO", "RELIG", "TRAV.IMP",
+     "TRAV.SATISF", "HARD.ROCK", "LECTURE.BD", "PECHE.CHASSE", "CUISINE",
+     "BRICOL", "CINEMA", "SPORT", "HEURES.TV")
R> summary(d.maj$SEXE)

Homme Femme
  899  1101
```

Ce type de renommage peut être utile lorsqu'on souhaite passer en revue tous les noms de variables d'un fichier importé pour les corriger le cas échéant. Pour faciliter un peu ce travail pas forcément passionant, on peut utiliser la fonction `dput` :

```
R> dput(names(d))

c("id", "age", "sexe", "nivetud", "poids", "occup", "qualif",
"freres.soeurs", "clso", "relig", "trav.imp", "trav.satisf",
"hard.rock", "lecture.bd", "peche.chasse", "cuisine", "bricol",
"cinema", "sport", "heures.tv")
```

On obtient en résultat la liste des variables sous forme de vecteur déclaré. On n'a plus alors qu'à copier/coller cette chaîne, rajouter `names(d) <-` devant, et modifier un à un les noms des variables.

rgrs

Si on souhaite seulement modifier le nom d'une variable, on peut utiliser la fonction `renomme.variable` de l'extension rgrs. Celle-ci prend en argument le tableau de données, le nom actuel de la variable et le nouveau nom. Par exemple, si on veut renommer la variable `bricol` du tableau de données `d` en `bricolage` :

```
R> d <- renomme.variable(d, "bricol", "bricolage")
R> table(d$bricolage)


 Non  Oui
1147  853
```

5.1.3 Facteurs

Parmi les différents types de variables, les *facteurs* (`factor`) sont à la fois à part et très utilisés, car ils vont correspondre à la plupart des variables issues d'une question fermée dans un questionnaire.

Les facteurs prennent leurs valeurs dans un ensemble de modalités prédéfinies, et ne peuvent en prendre d'autres. La liste des valeurs possibles est donnée par la fonction `levels` :

```
R> levels(d$sexe)

[1] "Homme" "Femme"
```

Si on veut modifier la valeur du sexe du premier individu de notre tableau de données avec une valeur différente, on obient un message d'erreur et une valeur manquante est utilisée à la place :

```
R> d$sexe[1] <- "Chihuahua"

Warning message: invalid factor level, NAs generated

R> d$sexe[1]

[1] <NA>
Levels: Homme Femme
```

On peut très facilement créer un facteur à partir d'une variable de type caractères avec la commande `factor` :

```
R> v <- factor(c("H", "H", "F", "H"))
R> v

[1] H H F H
Levels: F H
```

Par défaut, les niveaux d'un facteur nouvellement créés sont l'ensemble des valeurs de la variable caractères, ordonnées par ordre alphabétique. Cette ordre des niveaux est utilisé à chaque fois qu'on utilise des fonctions comme `table`, par exemple :

```
R> table(v)

v
F H
1 3
```

On peut modifier cet ordre au moment de la création du facteur en utilisant l'option `levels` :

```
R> v <- factor(c("H", "H", "F", "H"), levels = c("H", "F"))
R> table(v)

v
H F
3 1
```

On peut aussi modifier l'ordre des niveaux d'une variable déjà existante :

```
R> d$qualif <- factor(d$qualif, levels = c("Ouvrier specialise",
+     "Ouvrier qualifie", "Employe", "Technicien", "Profession intermediaire",
+     "Cadre", "Autre"))
R> table(d$qualif)


      Ouvrier specialise         Ouvrier qualifie                  Employe
                     203                      292                      594
              Technicien Profession intermediaire                    Cadre
                      86                      160                      260
                  Autre
                      58
```

Par défaut, les valeurs manquantes ne sont pas considérées comme un niveau de facteur. On peut cependant les transformer en niveau en utilisant la fonction `addNA`. Ceci signifie cependant qu'elle ne seront plus considérées comme manquantes par R :

```
R> summary(d$trav.satisf)

  Satisfaction Insatisfaction      Equilibre           NA's
           480            117            451            952

R> summary(addNA(d$trav.satisf))

  Satisfaction Insatisfaction      Equilibre           <NA>
           480            117            451            952
```

5.2 Indexation

L'indexation est l'une des fonctionnalités les plus puissantes mais aussi les plus difficiles à maîtriser de R. Il s'agit d'opérations permettant de sélectionner des sous-ensembles d'observations et/ou de variables en fonction de différents critères. L'indexation peut porter sur des vecteurs, des matrices ou des tableaux de données.

Le principe est toujours le même : on indique, entre crochets et à la suite du nom de l'objet à indexer, une série de conditions indiquant ce que l'on garde ou non. Ces conditions peuvent être de différents types.

5.2.1 Indexation directe

Le mode le plus simple d'indexation consiste à indiquer la position des éléments à conserver. Dans le cas d'un vecteur cela permet de sélectionner un ou plusieurs éléments de ce vecteur.

Soit le vecteur suivant :

```
R> v <- c("a", "b", "c", "d", "e", "f", "g")
```

Si on souhaite le premier élément du vecteur, on peut faire :

```
R> v[1]

[1] "a"
```

Si on souhaite les trois premiers éléments ou les éléments 2, 6 et 7 :

```
R> v[1:3]

[1] "a" "b" "c"

R> v[c(2, 6, 7)]

[1] "b" "f" "g"
```

Si on veut le dernier élément :

```
R> v[length(v)]

[1] "g"
```

Dans le cas de matrices ou de tableaux de données, l'indexation prend deux arguments séparés par une virgule : le premier concerne les *lignes* et le second les *colonnes*. Ainsi, si on veut l'élément correspondant à la troisième ligne et à la cinquième colonne du tableau de données `d` :

```
R> d[3, 5]

[1] 3994
```

On peut également indiquer des vecteurs :

```
R> d[1:3, 1:2]

  id age
1  1  28
2  2  23
3  3  59
```

Si on laisse l'un des deux critères vides, on sélectionne l'intégralité des lignes ou des colonnes. Ainsi si l'on veut seulement la cinquième colonne ou les deux premières lignes :

```
R> d[, 5]

 [1]  2634.4  9738.4  3994.1  5731.7  4329.1  8674.7  6165.8 12891.6  7808.9  2277.2
[11]   704.3  6697.9  7118.5   586.8 11042.1  9958.2  4836.1  1551.5  3141.2 27195.8
[21] 14648.0  8128.1  1281.9 11663.3  8780.3  1700.8  6662.8  3359.5  8536.1 10620.5
[31]  5264.3 14161.8  1339.6  9243.9  4512.3  7871.6  1357.0  7626.3  1630.3  2196.2
[41]  5606.0  8841.3  9113.5  2267.6  7706.3  2446.5  8118.3 10751.5   831.9  6591.6
[51]  1936.9   834.4  3432.5 11354.9  9293.0  6344.1  4899.9  4766.9  3462.8 23732.5
[61]   833.8  8529.4  3190.4  2423.1  5946.0 14991.9  2062.1  5702.1 20604.3  2634.5
 [ reached getOption("max.print") -- omitted 1930 entries ]]

R> d[1:2, ]

  id age  sexe                                                nivetud poids
1  1  28 Femme Enseignement superieur y compris technique superieur  2634
2  2  23 Femme                                                   <NA>  9738
                  occup  qualif freres.soeurs clso                      relig
1 Exerce une profession Employe             8  Oui Ni croyance ni appartenance
2       Etudiant, eleve    <NA>             2  Oui Ni croyance ni appartenance
       trav.imp    trav.satisf hard.rock lecture.bd peche.chasse cuisine bricol
1 Peu important Insatisfaction       Non        Non          Non     Oui    Non
2          <NA>           <NA>       Non        Non          Non     Non    Non
  cinema sport heures.tv
1    Non   Non         0
2    Oui   Oui         1
```

Enfin, si on préfixe les arguments avec le signe « - », ceci signifie « tous les éléments sauf ceux indiqués ». Si par exemple on veut tous les éléments de `v` sauf le premier :

```
R> v[-1]

[1] "b" "c" "d" "e" "f" "g"
```

Bien sûr, tous ces critères se combinent et on peut stocker le résultat dans un nouvel objet. Dans cet exemple `d2` contiendra les trois premières lignes de `d` mais sans les colonnes 2, 6 et 8.

```
R> d2 <- d[1:3, -c(2, 6, 8)]
```

5.2.2 Indexation par nom

Un autre mode d'indexation consiste à fournir non pas un numéro mais un nom sous forme de chaîne de caractères. On l'utilise couramment pour sélectionner les variables d'un tableau de données. Ainsi, les

deux fonctions suivantes sont équivalentes :

```
R> d$clso

 [1] Oui         Oui         Non         Non         Oui         Non
 [7] Oui         Non         Oui         Non         Oui         Oui
[13] Oui         Oui         Oui         Non         Non         Non
[19] Non         Non         Oui         Oui         Non         Non
[25] Non         Oui         Non         Non         Non         Oui
[31] Non         Oui         Oui         Non         Non         Oui
[37] Oui         Non         Non         Oui         Non         Non
[43] Oui         Non         Non         Non         Non         Oui
[49] Oui         Non         Non         Non         Oui         Non
[55] Oui         Oui         Non         Non         Oui         Non
[61] Non         Oui         Oui         Oui         Oui         Non
[67] Oui         Non         Non         Ne sait pas
 [ reached getOption("max.print") -- omitted 1930 entries ]]
Levels: Oui Non Ne sait pas

R> d[, "clso"]

 [1] Oui         Oui         Non         Non         Oui         Non
 [7] Oui         Non         Oui         Non         Oui         Oui
[13] Oui         Oui         Oui         Non         Non         Non
[19] Non         Non         Oui         Oui         Non         Non
[25] Non         Oui         Non         Non         Non         Oui
[31] Non         Oui         Oui         Non         Non         Oui
[37] Oui         Non         Non         Oui         Non         Non
[43] Oui         Non         Non         Non         Non         Oui
[49] Oui         Non         Non         Non         Oui         Non
[55] Oui         Oui         Non         Non         Oui         Non
[61] Non         Oui         Oui         Oui         Oui         Non
[67] Oui         Non         Non         Ne sait pas
 [ reached getOption("max.print") -- omitted 1930 entries ]]
Levels: Oui Non Ne sait pas
```

Là aussi on peut utiliser un vecteur pour sélectionner plusieurs noms et récupérer un « sous-tableau » de données :

```
R> d2 <- d[, c("id", "sexe", "age")]
```

Les noms peuvent également être utilisés pour les observations (lignes) d'un tableau de données si celles-ci ont été munies d'un nom avec la fonction `row.names`. Par défaut les noms de ligne sont leur numéro d'ordre, mais on peut leur assigner comme nom la valeur d'une variable d'identifiant. Ainsi, on peut assigner aux lignes du jeu de données `rp99` le nom des communes correspondantes :

```
R> row.names(rp99) <- rp99$nom
```

On peut alors accéder directement aux communes en donnant leur nom :

```
R> rp99[c("VILLEURBANNE", "OULLINS"), ]

                     nom  code pop.act pop.tot  pop15 nb.rp   agric artis cadres
VILLEURBANNE VILLEURBANNE 69266   57252  124152 103157 55136 0.02096 5.144  13.14
OULLINS           OULLINS 69149   11849   25186  20880 11091 0.10127 4.819  10.20
             interm  empl  ouvr  retr tx.chom  etud dipl.sup dipl.aucun proprio
VILLEURBANNE  25.72 31.42 23.07 36.65   14.82 15.51    9.744      16.90   37.62
OULLINS       27.43 31.53 24.37 41.55   10.64 10.63    7.625      14.32   51.51
               hlm locataire maison
VILLEURBANNE 23.34     32.77  6.533
OULLINS      14.56     29.92 17.708
```

Par contre il n'est pas possible d'utiliser directement l'opérateur « - » comme pour l'indexation directe. On doit effectuer une pirouette un peu compliquée utilisant la fonction `which`. Celle-ci renvoit les positions des éléments satisfaisant une condition. On peut ainsi faire :

```
R> which(names(d) == "qualif")

[1] 7

R> d2 <- d[, -which(names(d) == "qualif")]
```

Pour sélectionner toutes les colonnes sauf celle qui s'appelle `qualif`.

5.2.3 Indexation par conditions

Tests et conditions

Une condition est une expression logique dont le résultat est soit `TRUE` (vrai) soit `FALSE` (faux).

Une condition comprend la plupart du temps un opérateur de comparaison. Les plus courants sont les suivants :

Opérateur	Signification
==	égal à
!=	différent de
>	strictement supérieur à
<	strictement inférieur à
>=	supérieur ou égal à
<=	inférieur ou égal à

Voyons tout de suite un exemple :

```
R> d$sexe == "Homme"

 [1] FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[14] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
[27] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE
[40]  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
[53]  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[66] FALSE  TRUE  TRUE FALSE FALSE
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Que s'est-il passé ? Nous avons fourni à R une condition qui signifie « la valeur de la variable `sexe` vaut "Homme" ». Et il nous a renvoyé un vecteur avec autant d'éléments qu'il y'a d'observations dans `d`, et dont la valeur est `TRUE` si l'observation correspond à un homme, et `FALSE` dans les autres cas.

Prenons un autre exemple. On n'affichera cette fois que les premiers éléments de notre variable d'intérêt à l'aide de la fonction `head` :

```
R> head(d$age)

[1] 28 23 59 34 71 35

R> head(d$age > 40)

[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

On voit bien ici qu'à chaque élément du vecteur `d$age` dont la valeur est supérieure à 40 correspond un élément `TRUE` dans le résultat de la condition.

On peut combiner ou modifier des conditions à l'aide des opérateurs logiques habituels :

Opérateur	Signification
`&`	et logique
`|`	ou logique
`!`	négation logique

Comment les utilise-t-on ? Voyons tout de suite des exemples. Supposons que je veuille déterminer quels sont dans mon échantillon les hommes ouvriers spécialisés :

```
R> d$sexe == "Homme" & d$qualif == "Ouvrier specialise"

 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[14] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[27] FALSE  TRUE FALSE FALSE FALSE FALSE    NA    NA FALSE FALSE FALSE FALSE FALSE
[40] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[53] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE    NA FALSE    NA FALSE
[66] FALSE FALSE FALSE FALSE FALSE
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Si je souhaite identifier les personnes qui bricolent ou qui font la cuisine :

```
R> d$bricol == "Oui" | d$cuisine == "Oui"

 [1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
[14]  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[27]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE
[40]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE
[53] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[66]  TRUE  TRUE  TRUE  TRUE  TRUE
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Si je souhaite isoler les femmes qui ont entre 20 et 34 ans :

```
R> d$sexe == "Femme" & d$age >= 20 & d$age <= 34

 [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[14] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
[27]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[40] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
[53] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[66]  TRUE FALSE FALSE FALSE FALSE
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Si je souhaite récupérer les enquêtés qui ne sont pas cadres, on peut utiliser l'une des deux formes suivantes :

```
R> d$qualif != "Cadre"

 [1]  TRUE    NA  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE    NA  TRUE  TRUE  TRUE  TRUE
[14]    NA    NA  TRUE FALSE    NA  TRUE  TRUE  TRUE  TRUE  TRUE    NA  TRUE FALSE
[27]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE    NA    NA  TRUE  TRUE  TRUE    NA  TRUE
[40] FALSE  TRUE  TRUE FALSE FALSE    NA  TRUE  TRUE    NA  TRUE FALSE  TRUE  TRUE
[53]  TRUE    NA  TRUE  TRUE    NA FALSE  TRUE    NA  TRUE    NA  TRUE    NA  TRUE
[66]  TRUE  TRUE  TRUE  TRUE    NA
 [ reached getOption("max.print") -- omitted 1930 entries ]]

R> !(d$qualif == "Cadre")

 [1]  TRUE    NA  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE    NA  TRUE  TRUE  TRUE  TRUE
[14]    NA    NA  TRUE FALSE    NA  TRUE  TRUE  TRUE  TRUE  TRUE    NA  TRUE FALSE
[27]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE    NA    NA  TRUE  TRUE  TRUE    NA  TRUE
[40] FALSE  TRUE  TRUE FALSE FALSE    NA  TRUE  TRUE    NA  TRUE FALSE  TRUE  TRUE
[53]  TRUE    NA  TRUE  TRUE    NA FALSE  TRUE    NA  TRUE    NA  TRUE    NA  TRUE
[66]  TRUE  TRUE  TRUE  TRUE    NA
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Lorsqu'on mélange « et » et « ou » il est nécessaire d'utiliser des parenthèses pour différencier les blocs. La condition suivante identifie les femmes qui sont soit cadre, soit employée :

```
R> d$sexe == "Femme" & (d$qualif == "Employe" | d$qualif ==
+     "Cadre")

 [1]  TRUE    NA FALSE FALSE  TRUE  TRUE FALSE FALSE    NA FALSE  TRUE FALSE  TRUE
[14]    NA    NA  TRUE FALSE    NA FALSE FALSE  TRUE FALSE  TRUE    NA FALSE FALSE
[27] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE    NA  TRUE
[40] FALSE  TRUE FALSE FALSE  TRUE    NA FALSE FALSE    NA  TRUE  TRUE FALSE  TRUE
[53] FALSE    NA FALSE FALSE    NA  TRUE FALSE    NA  TRUE FALSE FALSE FALSE FALSE
[66]  TRUE FALSE FALSE  TRUE    NA
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

L'opérateur `%in%` peut être très utile : il teste si une valeur fait partie des éléments d'un vecteur. Ainsi on pourrait remplacer la condition précédente par :

```
R> d$sexe == "Femme" & d$qualif %in% c("Employe", "Cadre")

 [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
[14] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
[27] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
[40] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
[53] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[66]  TRUE FALSE FALSE  TRUE FALSE
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Enfin, signalons qu'on peut utiliser les fonctions `table` ou `summary` pour avoir une idée du résultat de notre condition :

```
R> table(d$sexe)


Homme Femme
  899  1101

R> table(d$sexe == "Homme")


FALSE  TRUE
 1101   899

R> summary(d$sexe == "Homme")

   Mode   FALSE    TRUE    NA's
logical    1101     899       0
```

Utilisation pour l'indexation

L'utilisation des conditions pour l'indexation est assez simple : si on indexe un vecteur avec un vecteur booléen, seuls les éléments correspondant à `TRUE` seront conservés.

Ainsi, si on fait :

```
R> dh <- d[d$sexe == "Homme", ]
```

On obtiendra un nouveau tableau de données comportant l'ensemble des variables de `d`, mais seulement les observations pour lesquelles `d$sexe` vaut « Homme ».

La plupart du temps ce type d'indexation s'applique aux lignes, mais on peut aussi l'utiliser sur les colonnes d'un tableau de données. L'exemple suivant, un peu compliqué, sélectionne uniquement les variables dont le nom commence par `a` ou `s` :

```
R> d[, substr(names(d), 0, 1) %in% c("a", "s")]

     age  sexe sport
1     28 Femme   Non
2     23 Femme   Oui
```

```
3     59 Homme   Oui
4     34 Homme   Oui
5     71 Femme   Non
6     35 Femme   Oui
7     60 Femme   Non
8     47 Homme   Non
9     20 Femme   Non
10    28 Homme   Oui
11    65 Femme   Non
12    47 Homme   Oui
13    63 Femme   Non
14    67 Femme   Non
15    76 Femme   Non
16    49 Femme   Non
17    62 Homme   Oui
18    20 Femme   Oui
19    70 Homme   Non
20    39 Femme   Oui
21    30 Femme   Non
22    30 Homme   Non
23    37 Femme   Oui
 [getOption("max.print") est atteint -- 1977 lignes omises ]]
```

On peut évidemment combiner les différents type d'indexation. L'exemple suivant sélectionne les femmes de plus de 40 ans et ne conserve que les variables `qualif` et `bricol`.

```
R> d2 <- d[d$sexe == "Femme" & d$age > 40, c("qualif", "bricol")]
```

Valeurs manquantes dans les conditions

Une remarque importante : quand l'un des termes d'une condition comporte une valeur manquante (`NA`), le résultat de cette condition n'est pas toujours `TRUE` ou `FALSE`, il peut aussi être à son tour une valeur manquante.

```
R> v <- c(1:5, NA)
R> v

[1]  1  2  3  4  5 NA

R> v > 3

[1] FALSE FALSE FALSE  TRUE  TRUE    NA
```

On voit que le test `NA > 3` ne renvoit ni vrai ni faux, mais `NA`.

Le résultat d'une condition peut donc comporter un grand nombre de valeurs manquantes :

```
R> summary(d$trav.satisf == "Satisfaction")

   Mode   FALSE    TRUE    NA's
logical     568     480     952
```

Une autre conséquence importante de ce comportement est qu'on ne peut pas utiliser l'opérateur `== NA` pour tester la présence de valeurs manquantes. On utilisera à la place la fonction *ad hoc* `is.na`.

On comprendra mieux le problème avec l'exemple suivant :

```
R> v <- c(1, NA)
R> v

[1]  1 NA

R> v == NA

[1] NA NA

R> is.na(v)

[1] FALSE  TRUE
```

Pour compliquer encore un peu le tout, lorsqu'on utilise une condition pour l'indexation, lorsque la condition renvoit `NA`, l'élément est sélectionné, comme s'il valait `TRUE`. Ceci aura donc des conséquences pour l'extraction de sous-populations, comme indiqué section 5.3.1 page suivante.

5.2.4 Indexation et assignation

Dans tous les exemples précédents, on a utilisé l'indexation pour extraire une partie d'un vecteur ou d'un tableau de données, en plaçant l'opération d'indexation à droite de l'opérateur `<-`.

Mais l'indexation peut également être placée à gauche de cet opérateur. Dans ce cas, les éléments sélectionnés par l'indexation sont alors remplacés par les valeurs indiquées à droite de l'opérateur `<-`.

Ceci est parfaitement incompréhensible. Prenons donc un exemple simple :

```
R> v <- 1:5
R> v

[1] 1 2 3 4 5

R> v[1] <- 3
R> v

[1] 3 2 3 4 5
```

Cette fois, au lieu d'utiliser quelque chose comme `x <- v[1]`, qui aurait placé la valeur du premier élément de `v` dans `x`, on a utilisé `v[1] <- 3`, ce qui a *mis à jour* le premier élément de `v` avec la valeur 3.

Ceci fonctionne également pour les tableaux de données et pour les différents types d'indexation évoqués précédemment :

```
R> d[257, "sexe"] <- "Homme"
```

Enfin on peut modifier plusieurs éléments d'un seul coup soit en fournissant un vecteur, soit en profitant du mécanisme de recyclage. Les deux commandes suivantes sont ainsi rigoureusement équivalentes :

```
R> d[c(257, 438, 889), "sexe"] <- c("Homme", "Homme", "Homme")
R> d[c(257, 438, 889), "sexe"] <- "Homme"
```

On commence à voir comment l'utilisation de l'indexation par conditions et de l'assignation va nous permettre de faire des recodages.

```
R> d$age[d$age >= 20 & d$age <= 30] <- "20-30 ans"
R> d$age[is.na(d$age)] <- "Inconnu"
```

5.3 Sous-populations

5.3.1 Par indexation

La première manière de construire des sous-populations est d'utiliser l'indexation par conditions. On peut ainsi facilement sélectionner une partie des observations suivant un ou plusieurs critères et placer le résultat dans un nouveau tableau de données.

Par exemple si on souhaite isoler les hommes et les femmes :

```
R> dh <- d[d$sexe == "Homme", ]
R> df <- d[d$sexe == "Femme", ]
R> table(d$sexe)


Homme Femme
  899  1101

R> dim(dh)

[1] 899  20

R> dim(df)

[1] 1101   20
```

On a à partir de là trois tableaux de données, `d` comportant la population totale, `dh` seulement les hommes et `df` seulement les femmes.

On peut évidemment combiner plusieurs critères :

```
R> dh.25 <- d[d$sexe == "Homme" & d$age <= 25, ]
R> dim(dh.25)

[1] 86 20
```

Si on utilise directement l'indexation, il convient cependant d'être extrêmement prudent avec les valeurs manquantes. Comme indiqué précédemment, la présence d'une valeur manquante dans une condition fait que celle-ci est évaluée en `NA` et au final sélectionnée par l'indexation :

```
R> summary(d$trav.satisf)

  Satisfaction Insatisfaction      Equilibre           NA's
           480            117            451            952

R> d.satisf <- d[d$trav.satisf == "Satisfaction", ]
R> dim(d.satisf)

[1] 1432   20
```

Comme on le voit, ici `d.satisf` contient les individus ayant la modalité *Satisfaction* mais aussi ceux ayant une valeur manquante `NA`. C'est pourquoi il faut toujours soit vérifier au préalable qu'on n'a pas de valeurs manquantes dans les variables de la condition, soit exclure explicitement les `NA` de la manière suivante :

```
R> d.satisf <- d[d$trav.satisf == "Satisfaction" & !is.na(d$trav.satisf),
+     ]
R> dim(d.satisf)

[1] 480  20
```

C'est notamment pour cette raison qu'on préfèrera le plus souvent utiliser la fonction `subset`.

5.3.2 Fonction `subset`

La fonction `subset` permet d'extraire des sous-populations de manière plus simple et un peu plus intuitive que l'indexation directe.

Celle-ci prend trois arguments principaux :

- le nom de l'objet de départ ;
- une condition sur les observations (`subset`) ;
- éventuellement une condition sur les colonnes (`select`).

Reprenons tout de suite un exemple déjà vu :

```
R> dh <- subset(d, sexe == "Homme")
R> df <- subset(d, sexe == "Femme")
```

L'utilisation de `subset` présente plusieurs avantages. Le premier est d'économiser quelques touches. On n'est en effet pas obligé de saisir le nom du tableau de données dans la condition sur les lignes. Ainsi les deux commandes suivantes sont équivalentes :

```
R> dh <- subset(d, d$sexe == "Homme")
R> dh <- subset(d, sexe == "Homme")
```

Le second avantage est que `subset` s'occupe du problème des valeurs manquantes évoquées précédemment et les exclut de lui-même, contrairement au comportement par défaut :

```
R> summary(d$trav.satisf)

  Satisfaction Insatisfaction      Equilibre           NA's
           480            117            451            952
```

```
R> d.satisf <- d[d$trav.satisf == "Satisfaction", ]
R> dim(d.satisf)

[1] 1432   20

R> d.satisf <- subset(d, trav.satisf == "Satisfaction")
R> dim(d.satisf)

[1] 480  20
```

Enfin, l'utilisation de l'argument `select` est simplifié pour l'expression de condition sur les colonnes. On peut ainsi spécifier les noms de variable sans guillemets et leur appliquer directement l'opérateur d'exclusion `-` :

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

5.3.3 Fonction `tapply`

Cette section documente une fonction qui peut être très utile, mais pas forcément indispensable au départ.

La fonction `tapply` n'est qu'indirectement liée à la notion de sous-population, mais peut permettre d'éviter d'avoir à créer ces sous-populations dans certains cas.

Son fonctionnement est assez simple, mais pas forcément intuitif. La fonction prend trois arguments : un vecteur, un facteur et une fonction. Elle applique ensuite la fonction aux éléments du vecteur correspondant à un même niveau du facteur. Vite, un exemple !

```
R> tapply(d$age, d$sexe, mean)

Homme Femme
48.16 48.15
```

Qu'est-ce que ça signifie ? Ici `tapply` a sélectionné toutes les observations correspondant à « Homme », puis appliqué la fonction `mean` aux valeurs de `age` correspondantes. Puis elle a fait de même pour les observations correspondant à « Femme ». On a donc ici la moyenne d'âge chez les hommes et chez les femmes.

On peut fournir à peu près n'importe quelle fonction à `tapply` :

```
R> tapply(d$bricol, d$sexe, freq)

$Homme
      n    %
Non 384 42.7
Oui 515 57.3
NA    0  0.0
```

```
$Femme
      n    %
Non 763 69.3
Oui 338 30.7
NA    0  0.0
```

Les arguments supplémentaires fournis à `tapply` sont en fait fournis directement à la fonction appelée.

```
R> tapply(d$bricol, d$sexe, freq, total = TRUE)

$Homme
        n     %
Non   384  42.7
Oui   515  57.3
NA      0   0.0
Total 899 100.0

$Femme
         n     %
Non    763  69.3
Oui    338  30.7
NA       0   0.0
Total 1101 100.0
```

5.4 Recodages

Le recodage de variables est une opération extrêmement fréquente lors du traitement d'enquête. Celui-ci utilise soit l'une des formes d'indexation décrites précédemment, soit des fonctions *ad hoc* de R.

On passe ici en revue différents types de recodage parmi les plus courants. Les exemples s'appuient, comme précédemment, sur l'extrait de l'enquête *Histoire de vie* :

```
R> data(hdv2003)
R> d <- hdv2003
```

5.4.1 Convertir une variable

Il peut arriver qu'on veuille transformer une variable d'un type dans un autre.

Par exemple, on peut considérer que la variable numérique `freres.soeurs` est une « fausse » variable numérique et qu'une représentation sous forme de facteur serait plus adéquate. Dans ce cas il suffit de faire appel à la fonction `factor` :

```
R> d$fs.fac <- factor(d$freres.soeurs)
R> levels(d$fs.fac)

 [1] "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15"
[17] "16" "18" "22"
```

La conversion d'une variable caractères en facteur se fait de la même manière.

La conversion d'un facteur ou d'une variable numérique en variable caractères peut se faire à l'aide de la fonction `as.character` :

```
R> d$fs.char <- as.character(d$freres.soeurs)
R> d$qualif.char <- as.character(d$qualif)
```

La conversion d'un facteur en caractères est fréquemment utilisé lors des recodages du fait qu'il est impossible d'ajouter de nouvelles modalités à un facteur. Par exemple, la première des commandes suivantes génère un message d'avertissement, tandis que les deux autres fonctionnent :

```
R> d$qualif[d$qualif == "Ouvrier specialise"] <- "Ouvrier"
R> d$qualif.char <- as.character(d$qualif)
R> d$qualif.char[d$qualif.char == "Ouvrier specialise"] <- "Ouvrier"
```

Dans le premier cas, le message d'avertissement indique que toutes les modaolités « Ouvrier specialise » de notre variable `qualif` ont été remplacées par des valeurs manquantes `NA`.

Enfin, une variable de type caractères dont les valeurs seraient des nombres peut être convertie en variable numérique avec la fonction `as.numeric`. Si on souhaite convertir un facteur en variable numérique, il faut d'abord le convertir en variable de classe caractère :

```
R> d$fs.num <- as.numeric(as.character(d$fs.fac))
```

5.4.2 Découper une variable numérique en classes

Le premier type de recodage consiste à découper une variable de type numérique en un certain nombre de classes. On utilise pour cela la fonction `cut`.

Celle-ci prend, outre la variable à découper, un certain nombre d'arguments :
- `breaks` indique soit le nombre de classes souhaité, soit, si on lui fournit un vecteur, les limites des classes ;
- `labels` permet de modifier les noms de modalités attribués aux classes ;
- `include.lowest` et `right` influent sur la manière dont les valeurs situées à la frontière des classes seront inclues ou exclues ;
- `dig.lab` indique le nombre de chiffres après la virgule à conserver dans les noms de modalités.

Prenons tout de suite un exemple et tentons de découper notre variable `age` en cinq classes et de placer le résultat dans une nouvelle variable nommée `age5cl` :

```
R> d$age5cl <- cut(d$age, 5)
R> table(d$age5cl)


(17.9,33.8] (33.8,49.6] (49.6,65.4] (65.4,81.2] (81.2,97.1]
        454         628         556         319          43
```

Par défaut R nous a bien créé cinq classes d'amplitudes égales. La première classe va de 16,9 à 32,2 ans (en fait de 17 à 32), etc.

Les frontières de classe seraient plus présentables si elles utilisaient des nombres entiers. On va donc spécifier manuellement le découpage souhaité, par tranches de 20 ans :

```
R> d$age20 <- cut(d$age, c(0, 20, 40, 60, 80, 100))
R> table(d$age20)


  (0,20]  (20,40]  (40,60]  (60,80] (80,100]
      72      660      780      436       52
```

On aurait pu tenir compte des âges extrêmes pour la première et la dernière valeur :

```
R> range(d$age)

[1] 18 97

R> d$age20 <- cut(d$age, c(17, 20, 40, 60, 80, 93))
R> table(d$age20)


(17,20] (20,40] (40,60] (60,80] (80,93]
     72     660     780     436      50
```

Les symboles dans les noms attribués aux classes ont leur importance : `(` signifie que la frontière de la classe est exclue, tandis que `[` signifie qu'elle est incluse. Ainsi, `(20,40]` signifie « strictement supérieur à 20 et inférieur ou égal à 40 ».

On remarque que du coup, dans notre exemple précédent, la valeur minimale, 17, est exclue de notre première classe, et qu'une observation est donc absente de ce découpage. Pour résoudre ce problème on peut soit faire commencer la première classe à 16, soit utiliser l'option `include.lowest=TRUE` :

```
R> d$age20 <- cut(d$age, c(16, 20, 40, 60, 80, 93))
R> table(d$age20)


(16,20] (20,40] (40,60] (60,80] (80,93]
     72     660     780     436      50

R> d$age20 <- cut(d$age, c(17, 20, 40, 60, 80, 93), include.lowest = TRUE)
R> table(d$age20)


[17,20] (20,40] (40,60] (60,80] (80,93]
     72     660     780     436      50
```

On peut également modifier le sens des intervalles avec l'option `right=FALSE`, et indiquer manuellement les noms des modalités avec `labels` :

```
R> d$age20 <- cut(d$age, c(16, 20, 40, 60, 80, 93), right = FALSE,
+     include.lowest = TRUE)
R> table(d$age20)


[16,20) [20,40) [40,60) [60,80) [80,93]
     48     643     793     454      60
```

```
R> d$age20 <- cut(d$age, c(17, 20, 40, 60, 80, 93), include.lowest = TRUE,
+     labels = c("<20ans", "21-40 ans", "41-60ans", "61-80ans", ">80ans"))
R> table(d$age20)


   <20ans 21-40 ans  41-60ans  61-80ans    >80ans
       72       660       780       436        50
```

Enfin, l'extension rgrs propose une fonction `quant.cut` permettant de découper une variable numérique en un nombre de classes donné ayant des efffectifs semblables. Il suffit de lui passer le nombre de classes en argument : *rgrs*

```
R> d$age6cl <- quant.cut(d$age, 6)
R> table(d$age6cl)


    [18,30)     [30,39)     [39,48) [48,55.667) [55.667,66)     [66,97]
        302         337         350         344         305         362
```

`quant.cut` admet les mêmes autres options que `cut` (`include.lowest, right, labels`...).

5.4.3 Regrouper les modalités d'une variable

Pour regrouper les modalités d'une variable qualitative (d'un facteur le plus souvent), on peut utiliser directement l'indexation.

Ainsi, si on veut recoder la variable `qualif` dans une variable `qualif.reg` plus « compacte », on peut utiliser :

```
R> table(d$qualif)


      Ouvrier specialise         Ouvrier qualifie                Technicien
                     203                      292                        86
Profession intermediaire                   Cadre                   Employe
                     160                      260                       594
                  Autre
                      58

R> d$qualif.reg[d$qualif == "Ouvrier specialise"] <- "Ouvrier"
R> d$qualif.reg[d$qualif == "Ouvrier qualifie"] <- "Ouvrier"
R> d$qualif.reg[d$qualif == "Employe"] <- "Employe"
R> d$qualif.reg[d$qualif == "Profession intermediaire"] <- "Intermediaire"
R> d$qualif.reg[d$qualif == "Technicien"] <- "Intermediaire"
R> d$qualif.reg[d$qualif == "Cadre"] <- "Cadre"
R> d$qualif.reg[d$qualif == "Autre"] <- "Autre"
R> table(d$qualif.reg)


       Autre         Cadre       Employe Intermediaire       Ouvrier
          58           260           594           246           495
```

On aurait pu représenter ce recodage de manière plus compacte, notamment en commençant par copier le contenu de `qualif` dans `qualif.reg`, ce qui permet de ne pas s'occuper de ce qui ne change pas. Il est cependant nécessaire de ne pas copier `qualif` sous forme de facteur, sinon on ne pourrait ajouter de nouvelles modalités. On copie donc la version *caractères* de `qualif` grâce à la fonction `as.character` :

```
R> d$qualif.reg <- as.character(d$qualif)
R> d$qualif.reg[d$qualif == "Ouvrier specialise"] <- "Ouvrier"
R> d$qualif.reg[d$qualif == "Ouvrier qualifie"] <- "Ouvrier"
R> d$qualif.reg[d$qualif == "Profession intermediaire"] <- "Intermediaire"
R> d$qualif.reg[d$qualif == "Technicien"] <- "Intermediaire"
R> table(d$qualif.reg)


        Autre         Cadre       Employe Intermediaire       Ouvrier
           58           260           594           246           495
```

On peut faire une version encore plus compacte en utilisant l'opérateur logique *ou* (`|`) :

```
R> d$qualif.reg <- as.character(d$qualif)
R> d$qualif.reg[d$qualif == "Ouvrier specialise" | d$qualif ==
+     "Ouvrier qualifie"] <- "Ouvrier"
R> d$qualif.reg[d$qualif == "Profession intermediaire" |
+     d$qualif == "Technicien"] <- "Intermediaire"
R> table(d$qualif.reg)


        Autre         Cadre       Employe Intermediaire       Ouvrier
           58           260           594           246           495
```

Enfin, pour terminer ce petit tour d'horizon, on peut également remplacer l'opérateur `|` par `%in%`, qui peut parfois être plus lisible :

```
R> d$qualif.reg <- as.character(d$qualif)
R> d$qualif.reg[d$qualif %in% c("Ouvrier specialise", "Ouvrier qualifie")] <-
"Ouvrier"
R> d$qualif.reg[d$qualif %in% c("Profession intermediaire",
+     "Technicien")] <- "Intermediaire"
R> table(d$qualif.reg)


        Autre         Cadre       Employe Intermediaire       Ouvrier
           58           260           594           246           495
```

Dans tous les cas le résultat obtenu est une variable de type *caractère*. On pourra la convertir en *facteur* par un simple :

```
R> d$qualif.reg <- factor(d$qualif.reg)
```

Si on souhaite recoder les valeurs manquantes, il suffit de faire appel à la fonction `is.na` :

```
R> table(d$trav.satisf)
```

```
  Satisfaction Insatisfaction      Equilibre
           480            117            451

R> d$trav.satisf.reg <- as.character(d$trav.satisf)
R> d$trav.satisf.reg[is.na(d$trav.satisf)] <- "Valeur manquante"
R> table(d$trav.satisf.reg)


       Equilibre   Insatisfaction     Satisfaction Valeur manquante
             451              117              480              952
```

5.4.4 Variables calculées

La création d'une variable numérique à partir de calculs sur une ou plusieurs autres variables numériques se fait très simplement.

Supposons que l'on souhaite calculer une variable indiquant l'écart entre le nombre d'heures passées à regarder la télévision et la moyenne globale de cette variable. On pourrait alors faire :

```
R> range(d$heures.tv, na.rm = TRUE)

[1]  0 12

R> mean(d$heures.tv, na.rm = TRUE)

[1] 2.247

R> d$ecart.heures.tv <- d$heures.tv - mean(d$heures.tv,
+     na.rm = TRUE)
R> range(d$ecart.heures.tv, na.rm = TRUE)

[1] -2.247  9.753

R> mean(d$ecart.heures.tv, na.rm = TRUE)

[1] 4.715e-17
```

Autre exemple tiré du jeu de données `rp99` : si on souhaite calculer le pourcentage d'actifs dans chaque commune, on peut diviser la population active `pop.act` par la population totale `pop.tot`.

```
R> rp99$part.actifs <- rp99$pop.act/rp99$pop.tot * 100
```

5.4.5 Combiner plusieurs variables

La combinaison de plusieurs variables se fait à l'aide des techniques d'indexation déjà décrites précédemment. Le plus compliqué est d'arriver à formuler des conditions parfois complexes de manière rigoureuse.

On peut ainsi vouloir combiner plusieurs variables qualitatives en une seule :

```
R> d$act.manuelles <- NA
R> d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Oui"] <- "Cuisine et Bricolage"
R> d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Non"] <- "Cuisine seulement"
R> d$act.manuelles[d$cuisine == "Non" & d$bricol == "Oui"] <- "Bricolage seulement"
R> d$act.manuelles[d$cuisine == "Non" & d$bricol == "Non"] <- "Ni cuisine ni
bricolage"
R> table(d$act.manuelles)

    Bricolage seulement    Cuisine et Bricolage       Cuisine seulement
                    437                     416                     465
Ni cuisine ni bricolage
                    682
```

On peut également combiner variables qualitatives et variables quantitatives :

```
R> d$age.sexe <- NA
R> d$age.sexe[d$sexe == "Homme" & d$age < 40] <- "Homme moins de 40 ans"
R> d$age.sexe[d$sexe == "Homme" & d$age >= 40] <- "Homme plus de 40 ans"
R> d$age.sexe[d$sexe == "Femme" & d$age < 40] <- "Femme moins de 40 ans"
R> d$age.sexe[d$sexe == "Femme" & d$age >= 40] <- "Femme plus de 40 ans"
R> table(d$age.sexe)

Femme moins de 40 ans  Femme plus de 40 ans Homme moins de 40 ans
                  376                   725                   315
 Homme plus de 40 ans
                  584
```

Les combinaisons de variables un peu complexes nécessitent parfois un petit travail de réflexion. En particulier, l'ordre des commandes de recodage ont parfois une influence dans le résultat final.

5.4.6 Variables scores

Une variable score est une variable calculée en additionnant des poids accordés aux modalités d'une série de variables qualitatives.

Pour prendre un exemple tout à fait arbitraire, imaginons que nous souhaitons calculer un score d'activités extérieures. Dans ce score on considère que le fait d'aller au cinéma « pèse » 10, celui de pêcher ou chasser vaut 30 et celui de faire du sport vaut 20. On pourrait alors calculer notre score de la manière suivante :

```
R> d$score.ext <- 0
R> d$score.ext[d$cinema == "Oui"] <- d$score.ext[d$cinema ==
+     "Oui"] + 10
R> d$score.ext[d$peche.chasse == "Oui"] <- d$score.ext[d$peche.chasse ==
+     "Oui"] + 30
R> d$score.ext[d$sport == "Oui"] <- d$score.ext[d$sport ==
+     "Oui"] + 20
R> table(d$score.ext)

  0  10  20  30  40  50  60
800 342 229 509  31  41  48
```

Cette notation étant un peu lourde, on peut l'alléger un peu en utilisant la fonction `ifelse`. Celle-ci prend en argument une condition et deux valeurs. Si la condition est vraie elle retourne la première valeur, sinon elle retourne la seconde.

```
R> d$score.ext <- 0
R> d$score.ext <- ifelse(d$cinema == "Oui", 10, 0) + ifelse(d$peche.chasse ==
+     "Oui", 30, 0) + ifelse(d$sport == "Oui", 20, 0)
R> table(d$score.ext)


  0  10  20  30  40  50  60
800 342 229 509  31  41  48
```

5.4.7 Vérification des recodages

Il est très important de vérifier, notamment après les recodages les plus complexes, qu'on a bien obtenu le résultat escompté. Les deux points les plus sensibles étant les valeurs manquantes et les erreurs dans les conditions.

Pour vérifier tout cela le plus simple est sans doute de faire des tableaux croisés entre la variable recodée et celles ayant servi au recodage, à l'aide de la fonction `table`, et de vérifier le nombre de valeurs manquantes dans la variable recodée avec `summary`, `freq` ou `table`.

Par exemple :

```
R> d$act.manuelles <- NA
R> d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Oui"] <- "Cuisine et Bricolage"
R> d$act.manuelles[d$cuisine == "Oui" & d$bricol == "Non"] <- "Cuisine seulement"
R> d$act.manuelles[d$cuisine == "Non" & d$bricol == "Oui"] <- "Bricolage seulement"
R> d$act.manuelles[d$cuisine == "Non" & d$bricol == "Non"] <- "Ni cuisine ni
bricolage"
R> table(d$act.manuelles, d$cuisine)


                         Non Oui
  Bricolage seulement     437   0
  Cuisine et Bricolage      0 416
  Cuisine seulement         0 465
  Ni cuisine ni bricolage 682   0

R> table(d$act.manuelles, d$bricol)


                         Non Oui
  Bricolage seulement       0 437
  Cuisine et Bricolage      0 416
  Cuisine seulement       465   0
  Ni cuisine ni bricolage 682   0
```

5.5 Tri de tables

On a déjà évoqué l'existence de la fonction `sort`, qui permet de trier les éléments d'un vecteur.

```
R> sort(c(2, 5, 6, 1, 8))

[1] 1 2 5 6 8
```

On peut appliquer cette fonction à une variable, mais celle-ci ne permet que d'ordonner les valeurs de cette variable, et pas l'ensemble du tableau de données dont elle fait partie. Pour cela nous avons besoin d'une autre fonction, nommée `order`. Celle-ci ne renvoit pas les valeurs du vecteur triées, mais les emplacements de ces valeurs.

Un exemple pour comprendre :

```
R> order(c(15, 20, 10))

[1] 3 1 2
```

Le résultat renvoyé signifie que la plus petite valeur est la valeur située en 3ème position, suivie de celle en 1ère position et de celle en 2ème position. Tout cela ne paraît pas passionnant à première vue, mais si on mélange ce résultat avec un peu d'indexation directe, ça devient intéressant...

```
R> order(d$age)

 [1]  162  215  346  377  511  646  852  916 1211 1213 1261 1333 1395 1447 1600 1774
[17] 1937   38  100  134  196  204  256  257  349  395  407  427  453  578  726  969
[33] 1052 1056 1077 1177 1234 1250 1342 1377 1381 1382 1540 1559 1607 1634 1689 1983
[49]    9   18   25  231  335  347  358  488  496  642  826  922 1023 1042 1156 1175
[65] 1290 1384 1464 1467 1608 1661
 [ reached getOption("max.print") -- omitted 1930 entries ]]
```

Ce que cette fonction renvoit, c'est l'ordre dans lequel on doit placer les éléments de `age`, et donc par extension les lignes de `d`, pour que la variable soit triée par ordre croissant. Par conséquent, si on fait :

```
R> d.tri <- d[order(d$age), ]
```

Alors on a trié les lignes de `d` par ordre d'âge croissant ! Et si on fait un petit :

```
R> head(d.tri, 3)

     id age  sexe nivetud poids           occup qualif freres.soeurs clso
162 162  18 Homme    <NA>  4983 Etudiant, eleve   <NA>             2  Non
215 215  18 Homme    <NA>  4631 Etudiant, eleve   <NA>             2  Oui
                          relig trav.imp trav.satisf hard.rock lecture.bd
162  Appartenance sans pratique     <NA>        <NA>       Non        Non
215 Ni croyance ni appartenance     <NA>        <NA>       Non        Non
    peche.chasse cuisine bricol cinema sport heures.tv fs.fac fs.char qualif.char
162          Non     Non    Non    Non   Oui         3      2       2        <NA>
215          Non     Oui    Non    Oui   Oui         2      2       2        <NA>
    fs.num      age5cl  age20  age6cl qualif.reg  trav.satisf.reg ecart.heures.tv
162      2 (17.9,33.8] <20ans [18,30)       <NA> Valeur manquante          0.7534
215      2 (17.9,33.8] <20ans [18,30)       <NA> Valeur manquante         -0.2466
             act.manuelles              age.sexe score.ext
162 Ni cuisine ni bricolage Homme moins de 40 ans        20
215       Cuisine seulement Homme moins de 40 ans        30
 [getOption("max.print") est atteint -- dernière ligne omises ]]
```

On a les caractéristiques des trois enquêtés les plus jeunes.

On peut évidemment trier par ordre décroissant en utilisant l'option `decreasing=TRUE`. On peut donc afficher les caractéristiques des trois individus les plus âgés avec :

```
R> head(d[order(d$age, decreasing = TRUE), ], 3)
```

5.6 Fusion de tables

Lorsqu'on traite de grosses enquêtes, notamment les enquêtes de l'INSEE, on a souvent à gérer des données réparties dans plusieurs tables, soit du fait de la construction du questionnaire, soit du fait de contraintes techniques (fichiers `dbf` ou Excel limités à 256 colonnes, par exemple).

Une opération relativement courante consiste à *fusionner* plusieurs tables pour regrouper tout ou partie des données dans un unique tableau.

Nous allons simuler artificiellement une telle situation en créant deux tables à partir de l'extrait de l'enquête *Histoire de vie* :

```
R> data(hdv2003)
R> d <- hdv2003
R> dim(d)

[1] 2000   20

R> d1 <- subset(d, select = c("id", "age", "sexe"))
R> dim(d1)

[1] 2000    3

R> d2 <- subset(d, select = c("id", "clso"))
R> dim(d2)

[1] 2000    2
```

On a donc deux tableaux de données, `d1` et `d2`, comportant chacun 2000 lignes et respectivement 3 et 2 colonnes. Comment les rassembler pour n'en former qu'un ?

Intuitivement, cela paraît simple. Il suffit de « coller » `d2` à la droite de `d1`, comme dans l'exemple suivant.

Id	V1	V2
1	H	12
2	H	17
3	F	41
4	F	9
⋮	⋮	⋮

+

Id	V3
1	Rouge
2	Bleu
3	Bleu
4	Rouge
⋮	⋮

=

Id	V1	V2	V3
1	H	12	Rouge
2	H	17	Bleu
3	F	41	Bleu
4	F	9	Rouge
⋮	⋮	⋮	⋮

Cela semble fonctionner. La fonction qui permet d'effectuer cette opération sous R s'appelle `cbind`, elle « colle » des tableaux côte à côte en regroupant leurs colonnes [1].

1. L'équivalent de `cbind` pour les lignes s'appelle `rbind`.

```
R> cbind(d1, d2)

       id age  sexe   id        clso
1       1  28 Femme    1         Oui
2       2  23 Femme    2         Oui
3       3  59 Homme    3         Non
4       4  34 Homme    4         Non
5       5  71 Femme    5         Oui
6       6  35 Femme    6         Non
7       7  60 Femme    7         Oui
8       8  47 Homme    8         Non
9       9  20 Femme    9         Oui
10     10  28 Homme   10         Non
11     11  65 Femme   11         Oui
12     12  47 Homme   12         Oui
13     13  63 Femme   13         Oui
14     14  67 Femme   14         Oui
 [getOption("max.print") est atteint -- 1986 lignes omises ]]
```

À part le fait qu'on a une colonne `id` en double, le résultat semble satisfaisant. À première vue seulement. Imaginons maintenant que nous avons travaillé sur `d1` et `d2`, et que nous avons ordonné les lignes de `d1` selon l'âge des enquêtés :

```
R> d1 <- d1[order(d1$age), ]
```

Répétons l'opération de collage :

```
R> cbind(d1, d2)

       id age  sexe   id        clso
162   162  18 Homme    1         Oui
215   215  18 Homme    2         Oui
346   346  18 Femme    3         Non
377   377  18 Homme    4         Non
511   511  18 Homme    5         Oui
646   646  18 Homme    6         Non
852   852  18 Femme    7         Oui
916   916  18 Femme    8         Non
1211 1211  18 Homme    9         Oui
1213 1213  18 Femme   10         Non
1261 1261  18 Homme   11         Oui
1333 1333  18 Femme   12         Oui
1395 1395  18 Homme   13         Oui
1447 1447  18 Femme   14         Oui
 [getOption("max.print") est atteint -- 1986 lignes omises ]]
```

Que constate-t-on ? La présence de la variable `id` en double nous permet de voir que les identifiants ne coïncident plus ! En regroupant nos colonnes nous avons donc attribué à des individus les réponses d'autres individus.

La commande `cbind` ne peut en effet fonctionner que si les deux tableaux ont exactement le même nombre de lignes, et dans le même ordre, ce qui n'est pas le cas ici.

On va donc être obligé de pocéder à une *fusion* des deux tableaux, qui va permettre de rendre à chaque ligne ce qui lui appartient. Pour cela nous avons besoin d'un identifiant qui permet d'identifier chaque ligne de manière unique et qui doit être présent dans tous les tableaux. Dans notre cas, c'est plutôt rapide, il s'agit de la variable `id`.

Une fois l'identifiant identifié[2], on peut utiliser la commande `merge`. Celle-ci va fusionner les deux tableaux en supprimant les colonnes en double et en regroupant les lignes selon leurs identifiants :

```
R> d.complet <- merge(d1, d2, by = "id")
R> head(d.complet)

  id age  sexe clso
1  1  28 Femme  Oui
2  2  23 Femme  Oui
3  3  59 Homme  Non
4  4  34 Homme  Non
5  5  71 Femme  Oui
6  6  35 Femme  Non
```

Ici l'utilisation de la fonction est plutôt simple car nous sommes dans le cas de figure idéal : les lignes correspondent parfaitement et l'identifiant est clairement identifié. Parfois les choses peuvent être un peu plus compliquées :

- parfois les identifiants n'ont pas le même nom dans les deux tableaux. On peut alors les spécifier par les options `by.x` et `by.y` ;
- parfois les deux tableaux comportent des colonnes (hors identifiants) ayant le même nom. `merge` conserve dans ce cas ces deux colonnes mais les renomme en les suffixant par `.x` pour celles provenant du premier tableau, et `.y` pour celles du second ;
- parfois on n'a pas d'identifiant unique préétabli, mais on en construit un à partir de plusieurs variables. On peut alors donner un vecteur en paramètres de l'option `by`, par exemple `by=c("nom","prenom","date.naissance")`.

Une subtilité supplémentaire intervient lorsque les deux tableaux fusionnés n'ont pas exactement les mêmes lignes. Par défaut, `merge` ne conserve que les lignes présentes dans les deux tableaux :

Id	V1
1	H
2	H
3	F

+

Id	V2
1	10
2	15
5	31

=

Id	V1	V2
1	H	10
2	H	15

On peut cependant modifier ce comportement avec les options `all.x=TRUE` et `all.y=TRUE`. La première option indique de conserver toutes les lignes du premier tableau. Dans ce cas `merge` donne une valeur `NA` pour ces lignes aux colonnes provenant du second tableau. Ce qui donnerait :

Id	V1
1	H
2	H
3	F

+

Id	V2
1	10
2	15
5	31

=

Id	V1	V2
1	H	10
2	H	15
3	F	NA

`all.y` fait la même chose en conservant toutes les lignes du second tableau. On peut enfin décider toutes les lignes des deux tableaux en utilisant à la fois `all.x=TRUE` et `all.y=TRUE`, ce qui donne :

2. Si vous me passez l'expression...

Id	V1
1	H
2	H
3	F

+

Id	V2
1	10
2	15
5	31

=

Id	V1	V2
1	H	10
2	H	15
3	F	NA
5	NA	31

Parfois, l'un des identifiants est présent à plusieurs reprises dans l'un des tableaux (par exemple lorsque l'une des tables est un ensemble de ménages et que l'autre décrit l'ensemble des individus de ces ménages). Dans ce cas les lignes de l'autre table sont dupliquées autant de fois que nécessaires :

Id	V1
1	H
2	H
3	F

+

Id	V2
1	10
1	18
1	21
2	11
3	31

=

Id	V1	V2
1	H	10
1	H	18
1	H	21
2	H	11
3	F	31

5.7 Organiser ses scripts

Il ne s'agit pas ici de manipulation de données à proprement parler, mais plutôt d'une conséquence de ce qui a été vu précédemment : à mesure que recodages et traitements divers s'accumulent, votre script R risque de devenir rapidement très long et pas très pratique à éditer.

Il est très courant de répartir son travail entre différents fichiers, ce qui est rendu très simple par la fonction `source`. Celle-ci permet de lire le contenu d'un fichier de script et d'exécuter son contenu.

Prenons tout de suite un exemple. La plupart des scripts R commencent par charger les extensions utiles, par définir le répertoire de travail à l'aide de `setwd`, à importer les données, à effectuer manipulations, traitements et recodages, puis à mettre en oeuvre les analyses. Prenons le fichier fictif suivant :

```
library(rgrs)
library(foreign)

setwd("/home/julien/r/projet")

## IMPORT DES DONNÉES

d1 <- read.dbf("tab1.dbf")
d2 <- read.dbf("tab2.dbf")

d <- merge(d1, d2, by = "id")

## RECODAGES

d$tx.chomage <- as.numeric(d$tx.chomage)

d$pcs[d$pcs == "Ouvrier qualifie"] <- "Ouvrier"
d$pcs[d$pcs == "Ouvrier specialise"] <- "Ouvrier"

d$age5cl <- cut(d$age, 5)

## ANALYSES
```

```
tab <- table(d$tx.chomage, d$age5cl)
tab
chisq.test(tab)
```

Une manière d'organiser notre script [3] pourrait être de placer les opérations d'import des données et celles de recodage dans deux fichiers scripts séparés. Créons alors un fichier nommé `import.R` dans notre répertoire de travail et copions les lignes suivantes :

```
## IMPORT DES DONNÉES

d1 <- read.dbf("tab1.dbf")
d2 <- read.dbf("tab2.dbf")

d <- merge(d1, d2, by = "id")
```

Créons également un fichier `recodages.R` avec le contenu suivant :

```
## RECODAGES

d$tx.chomage <- as.numeric(d$tx.chomage)

d$pcs[d$pcs == "Ouvrier qualifie"] <- "Ouvrier"
d$pcs[d$pcs == "Ouvrier specialise"] <- "Ouvrier"

d$age5cl <- cut(d$age, 5)
```

Dés lors, si nous rajoutons les appels à la fonction `source` qui vont bien, le fichier suivant sera strictement équivalent à notre fichier de départ :

```
library(rgrs)
library(foreign)

setwd("/home/julien/r/projet")

source("import.R")
source("recodages.R")

## ANALYSES

tab <- table(d$tx.chomage, d$age5cl)
tab
chisq.test(tab)
```

Au fur et à mesure du travail sur les données, on placera les recodages que l'on souhaite conserver dans le fichier `recodages.R`.

Cette méthode présente plusieurs avantages :

– bien souvent, lorsqu'on effectue des recodages on se retrouve avec des variables recodées qu'on ne souhaite pas conserver. Si on prend l'habitude de placer les recodages intéressants dans le fichier `recodages.R`, alors il suffit d'exécuter les cinq premières lignes du fichier pour se retrouver avec un tableau de données `d` propre et complet.

3. Ceci n'est qu'une suggestion, la manière d'organiser (ou non) son travail étant bien évidemment très hautement subjectif.

– on peut répartir ses analyses dans différents scripts. Il suffit alors de copier les cinq premières lignes du fichier précédent dans chacun des scripts, et on aura l'assurance de travailler sur exactement les mêmes données.

Le premier point illustre l'une des caractéristiques de R : il est rare que l'on stocke les données modifiées. En général on repart toujours du fichier source original, et les recodages sont conservés sous forme de scripts et recalculés à chaque fois qu'on recommence à travailler. Ceci offre une traçabilité parfaite du traitement effectué sur les données.

5.8 Exercices

Exercice 5.11

▷ *Solution page 139*

Renommer la variable `clso` du jeu de données `hdv2003` en `classes.sociales`, puis la renommer en `clso`.

Exercice 5.12

▷ *Solution page 139*

Réordonner les niveaux du facteur `clso` pour que son tri à plat s'affiche de la manière suivante :

```
tmp
        Non Ne sait pas          Oui
       1037           27          936
```

Exercice 5.13

▷ *Solution page 139*

Affichez :
– les 3 premiers éléments de la variable `cinema`
– les éléments 12 à 30 de la variable `lecture.bd`
– les colonnes 4 et 8 des lignes 5 et 12 du jeu de données `hdv2003`
– les 4 derniers éléments de la variable `age`

Exercice 5.14

▷ *Solution page 140*

Construisez les sous-tableaux suivants avec la fonction `subset` :
– âge et sexe des lecteurs de BD
– ensemble des personnes n'étant pas chômeur (variable `occup`), sans la variable `cinema`
– identifiants des personnes de plus de 45 ans écoutant du hard rock
– femmes entre 25 et 40 ans n'ayant pas fait de sport dans les douze derniers mois
– hommes ayant entre 2 et 4 frères et sœurs et faisant la cuisine ou du bricolage

Exercice 5.15

▷ *Solution page 140*

Calculez le nombre moyen d'heures passées devant la télévision chez les lecteurs de BD, d'abord en construisant les sous-populations, puis avec la fonction `tapply`.

Exercice 5.16

▷ *Solution page 140*

Convertissez la variable `freres.soeurs` en variable de type caractères. Convertissez cette nouvelle variable en facteur. Puis convertissez à nouveau ce facteur en variable numérique. Vérifiez que votre variable finale est identique à la variable de départ.

Exercice 5.17

▷ *Solution page 140*

Découpez la variable `freres.soeurs` :
– en cinq classes d'amplitude égale
– en catégories « de 0 à 2 », « de 2 à 4 », « plus de 4 », avec les étiquettes correspondantes
– en quatre classes d'effectif équivalent
– d'où vient la différence d'effectifs entre les deux découpages précédents ?

Exercice 5.18

▷ *Solution page 141*

Recodez la variable `trav.imp` en `trav.imp2cl` pour obtenir les modalités « Le plus ou aussi important » et « moins ou peu important ». Vérifiez avec des tris à plat et un tableau croisé.

Recodez la variable `relig` en `relig.4cl` en regroupant les modalités « Pratiquant regulier » et « Pratiquant occasionnel » en une seule modalité « Pratiquant », et en remplaçant la modalité « NSP ou NVPR » par des valeurs manquantes. Vérifiez avec un tri croisé.

Exercice 5.19

▷ *Solution page 142*

Créez une variable ayant les modalités suivantes :
– Homme de plus de 40 ans lecteur de BD
– Homme de plus de 30 ans
– Femme faisant du bricolage
– Autre
Vérifier avec des tris croisés.

Exercice 5.20

▷ *Solution page 143*

Ordonner le tableau de données selon le nombre de frères et soeurs croissant. Afficher le sexe des 10 individus regardant le plus la télévision.

Partie 6

Statistique bivariée

On entend par statistique bivariée l'étude des relations entre deux variables, celles-ci pouvant être quantitatives ou qualitatives.

Comme dans la partie précédente, on travaillera sur les jeux de données fournis avec l'extension rgrs et tiré de l'enquête *Histoire de vie* et du recensement 1999 :

rgrs

```
R> data(hdv2003)
R> d <- hdv2003
R> data(rp99)
```

6.1 Deux variables quantitatives

La comparaison de deux variables quantitatives se fait en premier lieu graphiquement, en représentant l'ensemble des couples de valeurs. On peut ainsi représenter les valeurs du nombre d'heures passées devant la télévision selon l'âge (figure 6.1 page ci-contre).

Le fait que des points sont superposés ne facilite pas la lecture du graphique. On peut utiliser une représentation avec des points semi-transparents (figure 6.2 page 78).

Plus sophistiqué, on peut faire une estimation locale de densité et représenter le résultat sous forme de « carte ». Pour cela on commence par isoler les deux variables, supprimer les observations ayant au moins une valeur manquante à l'aide de la fonction `complete.cases`, estimer la densité locale à l'aide de la fonction `kde2d` de l'extension MASS [1] et représenter le tout à l'aide d'une des fonctions `image`, `contour` ou `filled.contour`... Le résultat est donné figure 6.3 page 79.

Dans tous les cas, il n'y a pas de structure très nette qui semble se dégager. On peut tester ceci mathématiquement en calculant le cœfficient de corrélation entre les deux variables à l'aide de la fonction `cor` :

```
R> cor(d$age, d$heures.tv, use = "complete.obs")

[1] 0.1776
```

L'option `use` permet d'éliminer les observations pour lesquelles l'une des deux valeurs est manquante. Le cœfficient de corrélation est très faible.

1. MASS est installée par défaut avec la version de base de R.

```
R> plot(d$age, d$heures.tv)
```

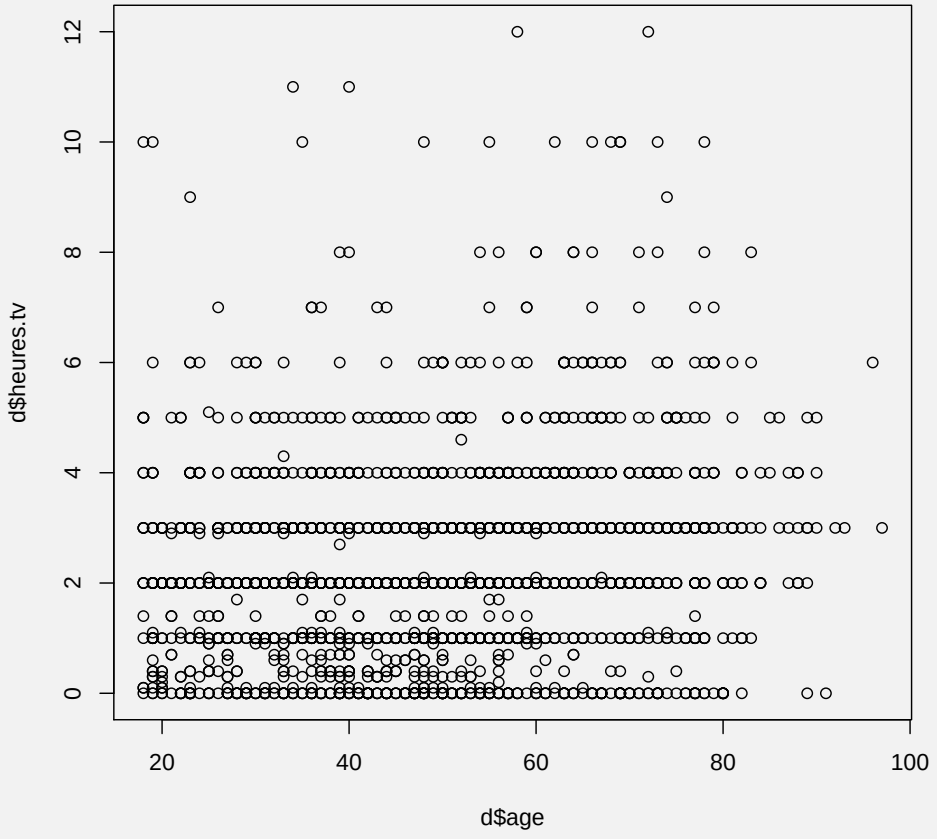


FIGURE 6.1 – Nombre d'heures de télévision selon l'âge

```
R> plot(d$age, d$heures.tv, pch = 19, col = rgb(1, 0, 0,
+     0.1))
```

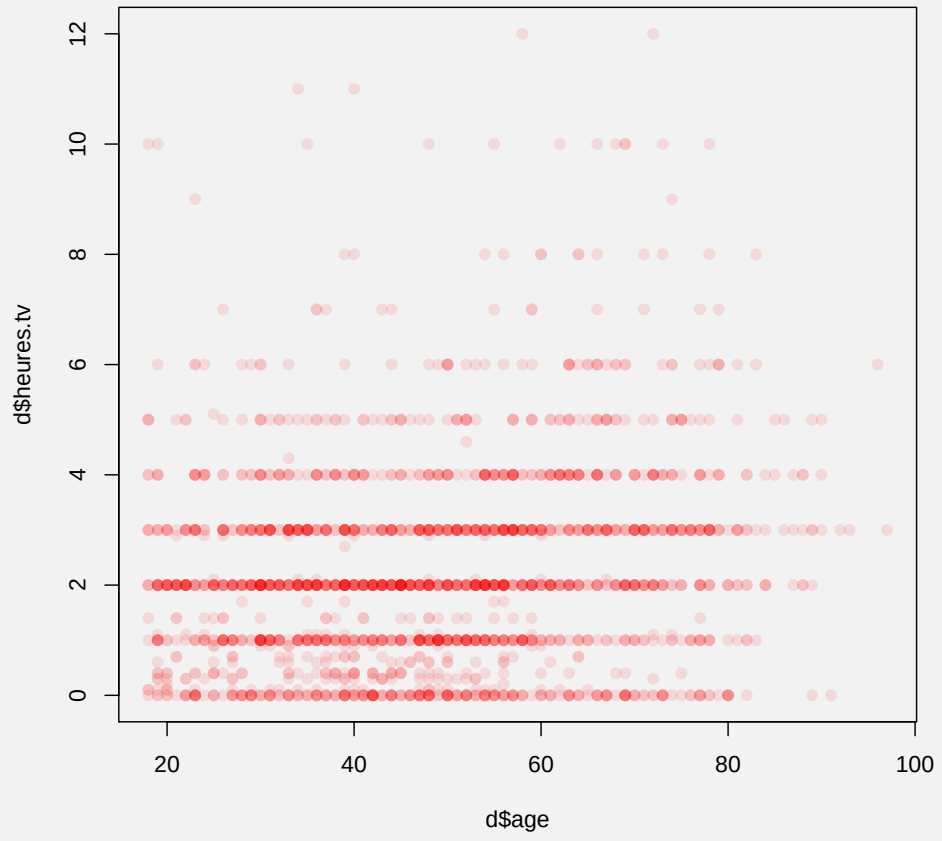


FIGURE 6.2 – Nombre d'heures de télévision selon l'âge avec semi-transparence

```
R> library(MASS)
R> tmp <- d[, c("age", "heures.tv")]
R> tmp <- tmp[complete.cases(tmp), ]
R> filled.contour(kde2d(tmp$age, tmp$heures.tv), color = terrain.colors)
```

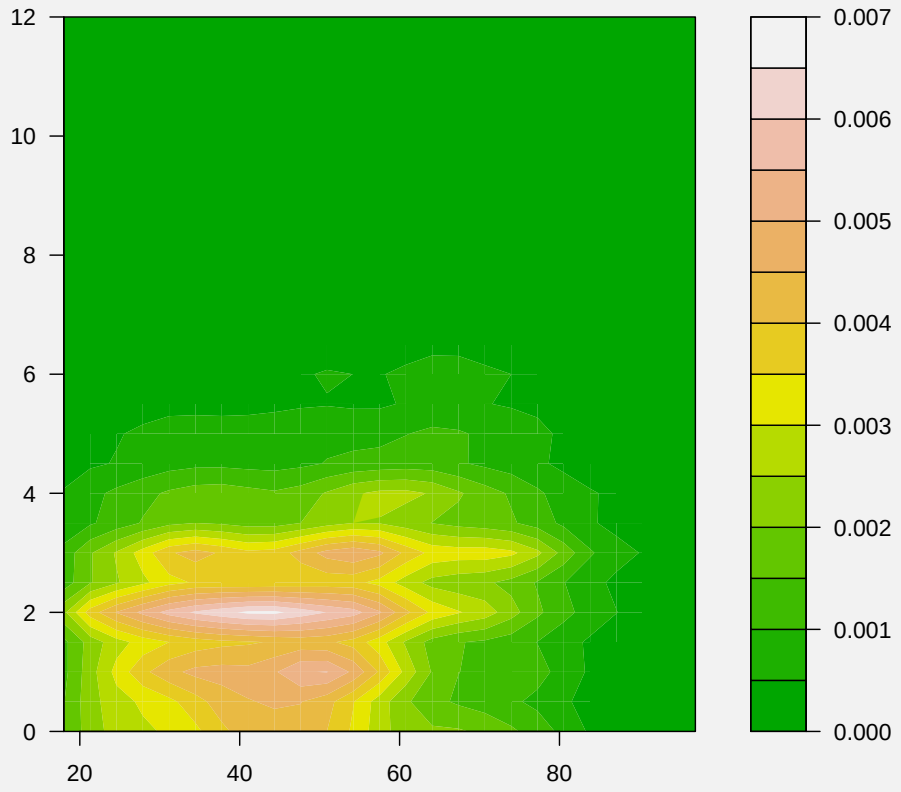


FIGURE 6.3 – Représentation de l'estimation de densité locale

```
R> plot(rp99$dipl.sup, rp99$cadres, ylab = "Part des cadres",
+     xlab = "Part des diplomês du supérieur")
```

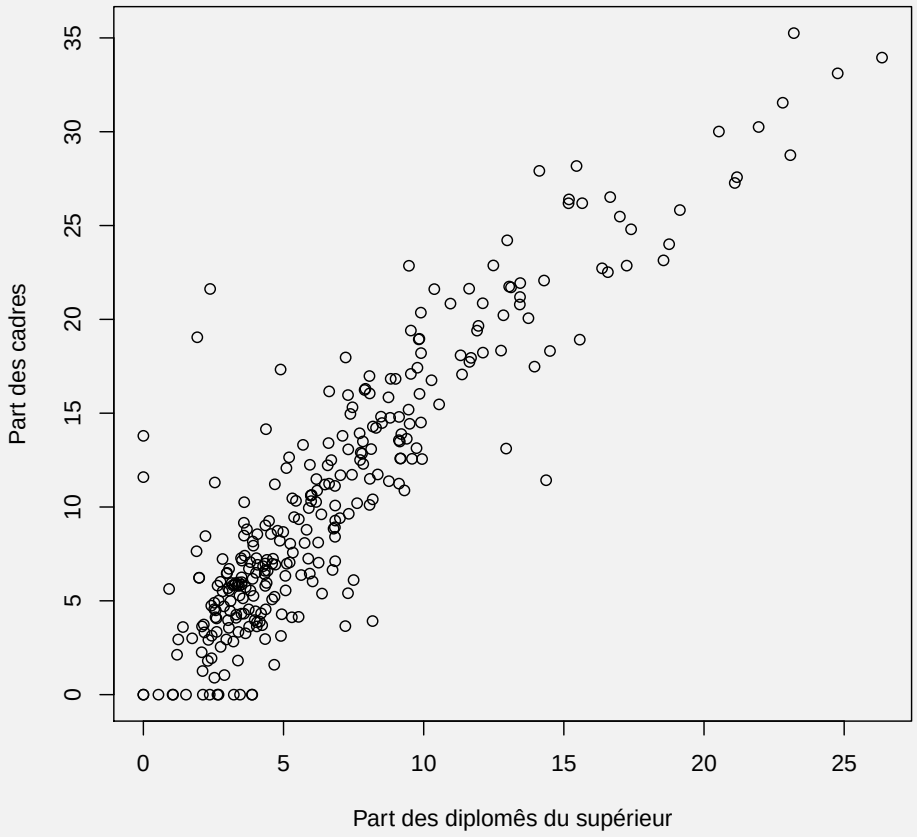


FIGURE 6.4 – Proportion de cadres et proportion de diplômés du supérieur

On va donc s'intéresser plutôt à deux variables présentes dans le jeu de données `rp99`, la part de diplômés du supérieur et la proportion de cadres dans les communes du Rhône en 1999.

À nouveau, commençons par représenter les deux variables (figure 6.4 de la présente page). Ça ressemble déjà beaucoup plus à une relation de type linéaire.

Calculons le coefficient de corrélation :

```
R> cor(rp99$dipl.sup, rp99$cadres)

[1] 0.8975
```

C'est beaucoup plus proche de 1. On peut alors effectuer une régression linéaire complète en utilisant la fonction `lm` :

```
R> reg <- lm(cadres ~ dipl.sup, data = rp99)
R> summary(reg)


Call:
lm(formula = cadres ~ dipl.sup, data = rp99)

Residuals:
   Min     1Q Median     3Q    Max
-9.691 -1.901 -0.182  1.491 17.087

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.2409     0.3299    3.76    2e-04 ***
dipl.sup      1.3835     0.0393   35.20   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.28 on 299 degrees of freedom
Multiple R-squared: 0.806,Adjusted R-squared: 0.805
F-statistic: 1.24e+03 on 1 and 299 DF,  p-value: <2e-16
```

Le résultat montre que les cœfficients sont significativement différents de 0. La part de cadres augmente donc avec celle de diplômés du supérieur (ô surprise). On peut très facilement représenter la droite de régression à l'aide de la fonction `abline` (figure 6.5 page suivante).

6.2 Une variable quantitative et une variable qualitative

Quand on parle de comparaison entre une variable quantitative et une variable qualitative, on veut en général savoir si la distribution des valeurs de la variable quantitative est la même selon les modalités de la variable qualitative. En clair : est ce que l'âge de ceux qui écoutent du hard rock est différent de l'âge de ceux qui n'en écoutent pas ?

Là encore, l'idéal est de commencer par une représentation graphique. Les boîtes à moustaches sont parfaitement adaptées pour cela.

Si on a construit des sous-populations d'individus écoutant ou non du hard rock, on peut utiliser la fonction `boxplot` comme indiqué figure 6.6 page 83.

Mais construire les sous-populations n'est pas nécessaire. On peut utiliser directement la version de `boxplot` prenant une *formule* en argument (figure 6.7 page 84).

À première vue, ô surprise, la population écoutant du hard rock a l'air sensiblement plus jeune. Peut-on le tester mathématiquement ? On peut calculer la moyenne d'âge des deux groupes en utilisant la fonction `tapply` [2] :

```
R> tapply(d$age, d$hard.rock, mean)

  Non   Oui
48.30 27.57
```

2. Fonction décrite page 59.

```
R> plot(rp99$dipl.sup, rp99$cadres, ylab = "Part des cadres",
+     xlab = "Part des diplômés du supérieur")
R> abline(reg, col = "red")
```

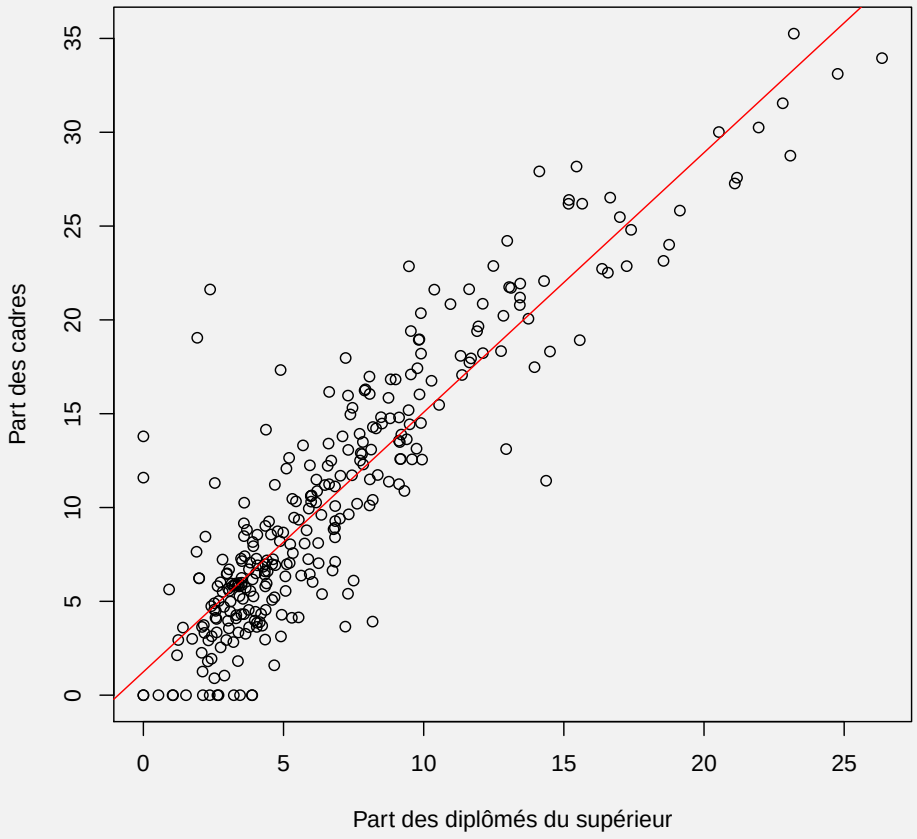


FIGURE 6.5 – Régression de la proportion de cadres par celle de diplômés du supérieur

```
R> d.hard <- subset(d, hard.rock == "Oui")
R> d.non.hard <- subset(d, hard.rock == "Non")
R> boxplot(d.hard$age, d.non.hard$age)
```

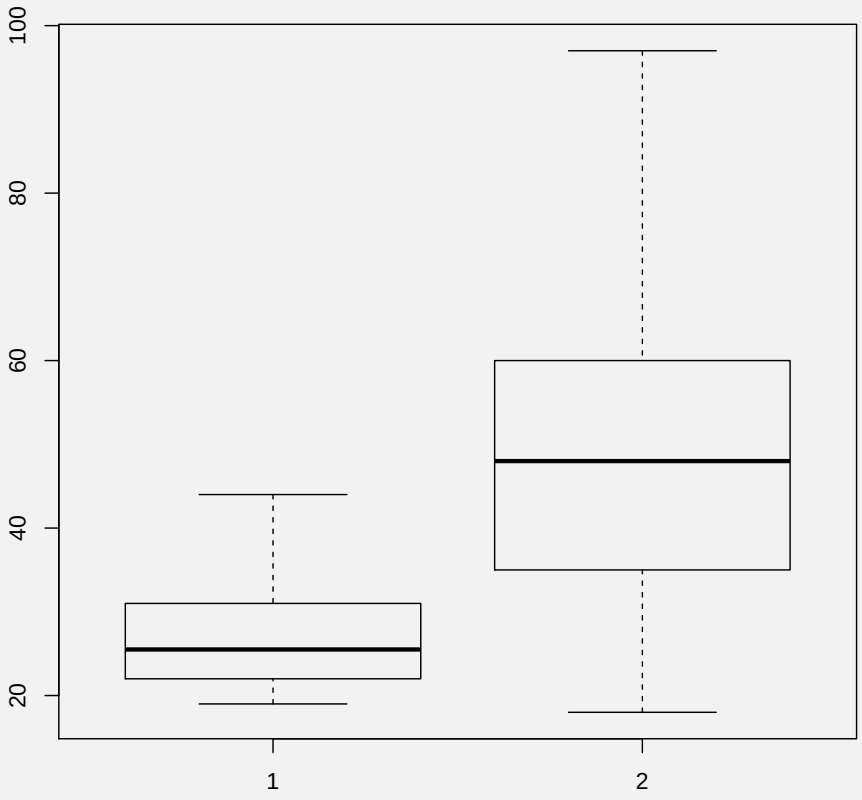


FIGURE 6.6 – *Boxplot* de la répartition des âges (sous-populations)

```
R> boxplot(age ~ hard.rock, data = d)
```

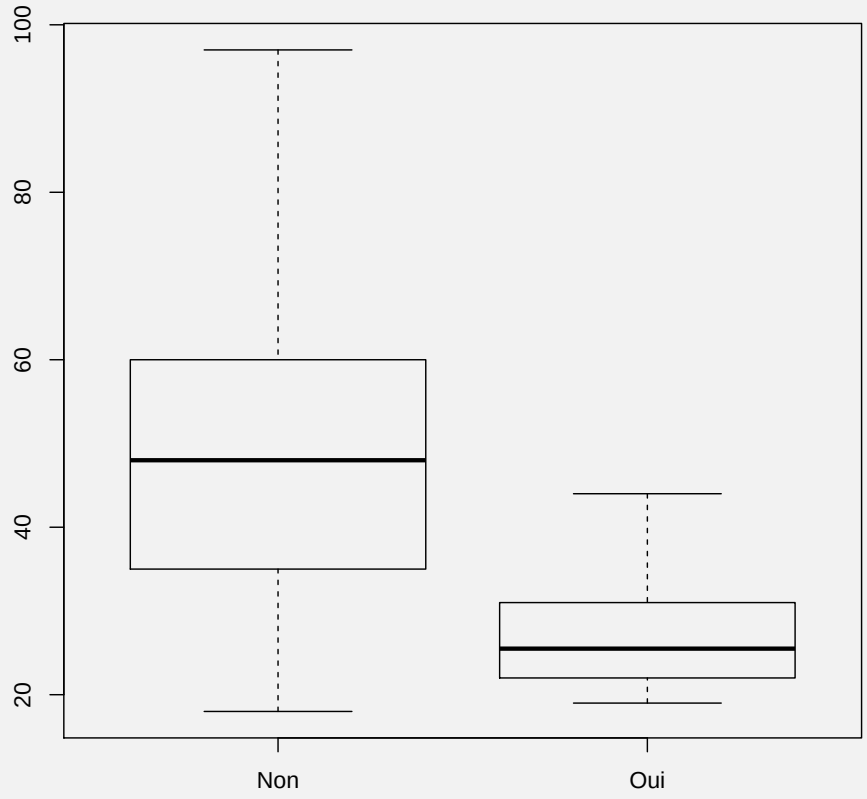


FIGURE 6.7 – *Boxplot* de la répartition des âges (formule)

L'écart est très important. Est-il statistiquement significatif ? Pour cela on peut faire un test t de comparaison de moyennes à l'aide de la fonction `t.test` :

```
R> t.test(d$age ~ d$hard.rock)


Welch Two Sample t-test

data:  d$age by d$hard.rock
t = 9.64, df = 13.85, p-value = 1.611e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 16.11 25.35
sample estimates:
mean in group Non mean in group Oui
            48.30             27.57
```

Le test est extrêmement significatif. L'intervalle de confiance à 95 % de la différence entre les deux moyennes va de 14,5 ans à 21,8 ans.

Nous sommes cependant allés un peu vite en besogne, car nous avons négligé une hypothèse fondamentale du test t : les ensembles de valeur comparés doivent suivre approximativement une loi normale et être de même variance [3]. Comment le vérifier ?

D'abord avec un petit graphique, comme sur la figure 6.8 page suivante.

Ça a l'air à peu près bon pour les « Sans hard rock », mais un peu plus limite pour les fans de *Metallica*, dont les effectifs sont d'ailleurs assez faibles. Si on veut en avoir le cœur net on peut utiliser le test de normalité de Shapiro-Wilk avec la fonction `shapiro.test` :

```
R> shapiro.test(d$age[d$hard.rock == "Oui"])


Shapiro-Wilk normality test

data:  d$age[d$hard.rock == "Oui"]
W = 0.8693, p-value = 0.04104


R> shapiro.test(d$age[d$hard.rock == "Non"])


Shapiro-Wilk normality test

data:  d$age[d$hard.rock == "Non"]
W = 0.9814, p-value = 2.085e-15
```

Visiblement, le test estime que les distributions ne sont pas suffisamment proches de la normalité dans les deux cas.

Et concernant l'égalité des variances ?

3. Concernant cette seconde condition, R propose une option nommée `var.equal` qui permet d'utiliser une approximation dans le cas où les variances ne sont pas égales

```
R> par(mfrow = c(1, 2))
R> hist(d$age[d$hard.rock == "Oui"], main = "Hard rock",
+     col = "red")
R> hist(d$age[d$hard.rock == "Non"], main = "Sans hard rock",
+     col = "red")
```

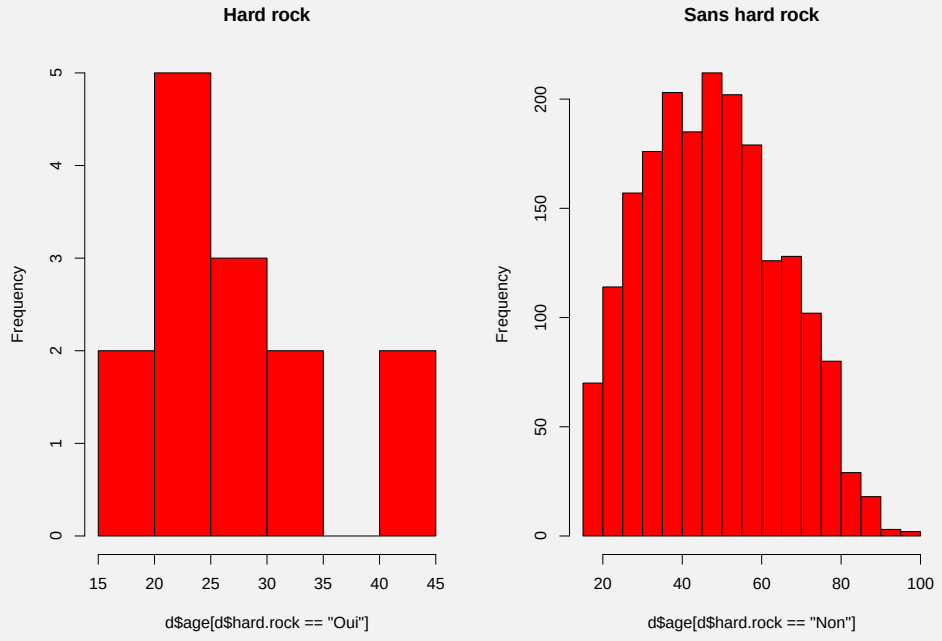


FIGURE 6.8 – Distribution des âges pour appréciation de la normalité

```
R> tapply(d$age, d$hard.rock, var)

   Non    Oui
285.63  62.73
```

L'écart n'a pas l'air négligeable. On peut le vérifier avec le test fourni par la fonction `var.test` :

```
R> var.test(d$age ~ d$hard.rock)


F test to compare two variances

data:  d$age by d$hard.rock
F = 4.554, num df = 1985, denom df = 13, p-value = 0.003217
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 1.752 8.694
sample estimates:
ratio of variances
             4.554
```

La différence est très significative. En toute rigueur le test t n'aurait donc pas pu être utilisé.

Damned ! Ces maudits tests statistiques vont-ils nous empêcher de faire connaître au monde entier notre fabuleuse découverte sur l'âge des fans de *Sepultura* ? Non ! Car voici qu'approche à l'horizon un nouveau test, connu sous le nom de *Wilcoxon/Mann-Whitney*. Celui-ci a l'avantage d'être *non-paramétrique*, c'est à dire de ne faire aucune hypothèse sur la distribution des échantillons comparés. Par contre il ne compare pas des différences de moyennes mais des différences de médianes :

```
R> wilcox.test(d$age ~ d$hard.rock)

Wilcoxon rank sum test with continuity correction

data:  d$age by d$hard.rock
W = 23980, p-value = 2.856e-06
alternative hypothesis: true location shift is not equal to 0
```

Ouf ! La différence est hautement significative[4]. Nous allons donc pouvoir entamer la rédaction de notre article pour la *Revue française de sociologie*.

6.3 Deux variables qualitatives

La comparaison de deux variables qualitatives s'appelle en général un *tableau croisé*. C'est sans doute l'une des analyses les plus fréquentes lors du traitement d'enquêtes en sciences sociales.

6.3.1 Tableau croisé

La manière la plus simple d'obtenir un tableau croisé est d'utiliser la fonction `table` en lui donnant en paramètres les deux variables à croiser. En l'occurrence nous allons croiser un recodage du niveau de

4. Ce test peut également fournir un intervalle de confiance avec l'option `conf.int=TRUE`.

qualification regroupé avec le fait de pratiquer un sport.

On commence par calculer la variable recodée et par afficher le tri à plat des deux variables :

```
R> d$qualreg <- as.character(d$qualif)
R> d$qualreg[d$qualif %in% c("Ouvrier specialise", "Ouvrier qualifie")] <- "Ouvrier"
R> d$qualreg[d$qualif %in% c("Profession intermediaire",
+     "Technicien")] <- "Intermediaire"
R> d$qualreg <- factor(d$qualreg)
R> table(d$qualreg)


        Autre         Cadre       Employe Intermediaire       Ouvrier
           58           260           594           246           495

R> table(d$sport)


 Non  Oui
1277  723
```

Le tableau croisé des deux variables s'obtient de la manière suivante :

```
R> table(d$sport, d$qualreg)


      Autre Cadre Employe Intermediaire Ouvrier
  Non    38   117     401           127     381
  Oui    20   143     193           119     114
```

rgrs On n'a cependant que les effectifs, ce qui rend difficile les comparaisons. L'extension rgrs fournit des fonctions permettant de calculer les pourcentages lignes, colonnes et totaux d'un tableau croisé.

Les pourcentages lignes s'obtiennent avec la fonction `lprop`. Celle-ci s'applique au tableau croisé généré par `table` :

```
R> tab <- table(d$sport, d$qualreg)
R> lprop(tab)


           Autre Cadre Employe Intermediaire Ouvrier Total
  Non        3.6  11.0  37.7    11.9          35.8   100.0
  Oui        3.4  24.3  32.8    20.2          19.4   100.0
  Ensemble   3.5  15.7  35.9    14.9          29.9   100.0
```

Les pourcentages ligne ne nous intéressent guère ici. On ne cherche pas à voir quelle est la proportion de cadres parmi ceux qui pratiquent un sport, mais plutôt quelle est la proportion de sportifs chez les cadres. Il nous faut donc des pourcentages colonnes, que l'on obtient avec la fonction `cprop` :

```
R> cprop(tab)


        Autre Cadre Employe Intermediaire Ouvrier Ensemble
```

```
  Non    65.5  45.0  67.5    51.6          77.0    64.4
  Oui    34.5  55.0  32.5    48.4          23.0    35.6
  Total 100.0 100.0 100.0   100.0         100.0   100.0
```

Dans l'ensemble, le pourcentage de personnes ayant pratiqué un sport est de 35,6 %. Mais cette proportion varie fortement d'une catégorie professionnelle à l'autre : 55,0 % chez les cadres contre 23,0 % chez les ouvriers.

À noter qu'on peut personnaliser l'affichage de ces tableaux de pourcentages à l'aide de différentes options, dont `digits`, qui règle le nombre de décimales à afficher, et `percent`, qui indique si on souhaite ou non rajouter un symbole % dans chaque case du tableau. Cette personnalisation peut se faire directement au moment de la génération du tableau, et dans ce cas elle sera utilisée par défaut :

```
R> ctab <- cprop(tab, digits = 2, percent = TRUE)
R> ctab

        Autre   Cadre   Employe Intermediaire Ouvrier Ensemble
  Non    65.52%  45.00%  67.51%  51.63%        76.97%  64.37%
  Oui    34.48%  55.00%  32.49%  48.37%        23.03%  35.63%
  Total 100.00% 100.00% 100.00% 100.00%       100.00% 100.00%
```

Ou bien ponctuellement en passant les mêmes arguments aux fonctions `print` (pour affichage dans R) ou `copie` (pour export vers un logiciel externe) :

```
R> ctab <- cprop(tab)
R> print(ctab, percent = TRUE)


        Autre  Cadre  Employe Intermediaire Ouvrier Ensemble
  Non    65.5%  45.0%  67.5%   51.6%         77.0%   64.4%
  Oui    34.5%  55.0%  32.5%   48.4%         23.0%   35.6%
  Total 100.0% 100.0% 100.0%  100.0%        100.0%  100.0%
```

6.3.2 χ^2 et dérivés

Pour tester l'existence d'un lien entre les modalités des deux variables, on va utiliser le très classique test du χ^2 [5]. Celui-ci s'obtient grâce à la fonction `chisq.test`, appliquée au tableau croisé obtenu avec `table` [6] :

```
R> chisq.test(tab)


Pearson's Chi-squared test

data:  tab
X-squared = 96.8, df = 4, p-value < 2.2e-16
```

5. On ne donnera pas plus d'indications sur le test du χ^2 ici. Les personnes désirant une présentation plus détaillée pourront se reporter (attention, séance d'autopromotion !) à la page suivante : http://alea.fr.eu.org/pages/khi2.

6. On peut aussi appliquer directement le test en spécifiant les deux variables à croiser *via* `chisq.test(d$qualreg, d$sport)`

Le test est hautement significatif, on ne peut pas considérer qu'il y a indépendance entre les lignes et les colonnes du tableau.

On peut affiner l'interprétation du test en déterminant dans quelle case l'écart à l'indépendance est le plus significatif en utilisant les *résidus* du test. Ceux-ci sont notamment affichables avec la fonction
rgrs `residus` de rgrs :

```
R> residus(tab)


      Autre Cadre Employe Intermediaire Ouvrier
  Non  0.11 -3.89    0.95         -2.49    3.49
  Oui -0.15  5.23   -1.28          3.35   -4.70
```

Les cases pour lesquelles l'écart à l'indépendance est significatif ont un résidu dont la valeur est supérieure à 2 ou inférieure à -2. Ici on constate que la pratique d'un sport est sur-représentée parmi les cadres et, à un niveau un peu moindre, parmi les professions intermédiaires, tandis qu'elle est sous-représentée chez les ouvriers.

Enfin, on peut calculer le cœfficient de contingence de Cramer du tableau, qui peut nous permettre
rgrs de le comparer par la suite à d'autres tableaux croisés. On peut pour cela utiliser la fonction `cramer.v`de rgrs :

```
R> cramer.v(tab)

[1] 0.242
```

6.3.3 Représentation graphique

Enfin, on peut obtenir une représentation graphique synthétisant l'ensemble des résultats obtenus sous la forme d'un graphique en mosaïque, grâce à la fonction `mosaicplot`. Le résultat est indiqué figure 6.9 page suivante.

Comment interpréter ce graphique haut en couleurs[7] ? Chaque rectangle représente une case de tableau. Sa largeur correspond au pourcentage des modalités en colonnes (il y'a beaucoup d'employés et d'ouvriers et très peu d'« autres »). Sa hauteur correspond aux pourcentages-colonnes : la proportion de sportifs chez les cadres est plus élevée que chez les employés. Enfin, la couleur de la case correspond au résidu du test du χ^2 correspondant : les cases en rouge sont sous-représentées, les cases en bleu sur-représentées, et les cases blanches sont statistiquement proches de l'hypothèse d'indépendance.

7. Sauf s'il est imprimé en noir et blanc...

```
R> mosaicplot(qualreg ~ sport, data = d, shade = TRUE, main = "Graphe en mosaïque")
```

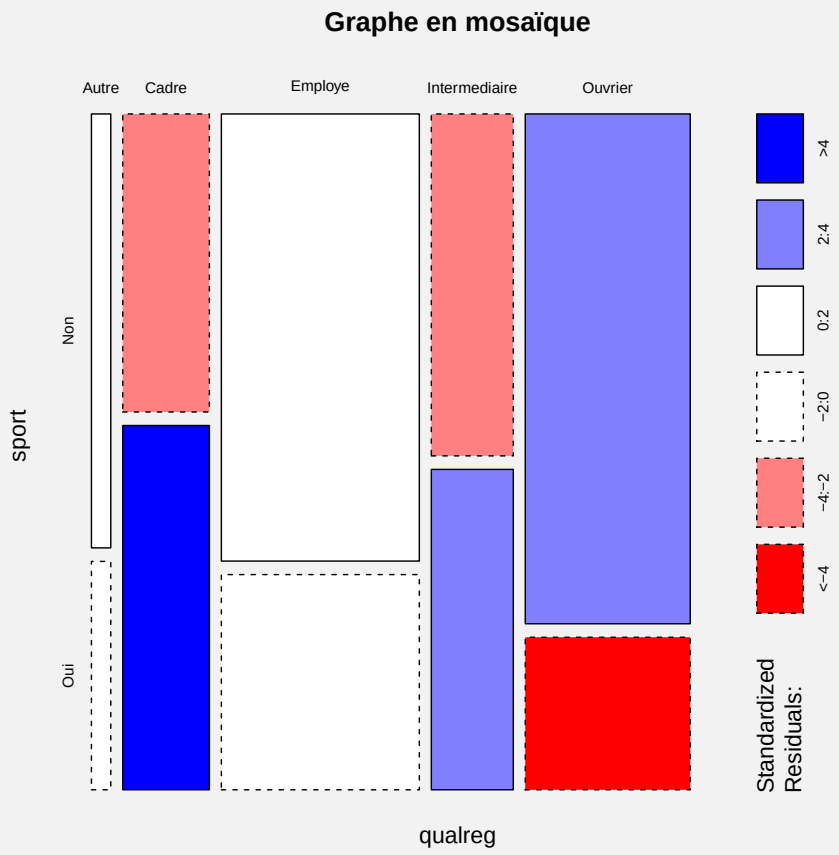


FIGURE 6.9 – Exemple de graphe en mosaïque

Partie 7

Données pondérées

S'il est tout à fait possible de travailler avec des données pondérées sous R, cette fonctionnalité n'est pas aussi bien intégrée que dans la plupart des autres logiciels de traitement statistique. En particulier, il y a plusieurs manières possibles de gérer la pondération.

Dans ce qui suit, on utilisera le jeu de données tiré de l'enquête *Histoire de vie* et notamment sa variable de pondération `poids`[1].

```
R> data(hdv2003)
R> d <- hdv2003
R> range(d$poids)

[1]    78.08 31092.14
```

7.1 Options de certaines fonctions

Tout d'abord, certaines fonctions de R acceptent en argument un vecteur permettant de pondérer les observations (l'option est en général nommée `weights` ou `row.w`). C'est le cas par exemple des méthodes d'estimation de modèles linéaires (`lm`) ou de modèles linéaires généralisés (`glm`), ou dans les analyses de correspondances des extensions ade4 (`dudi.acm`) ou FactoMineR (`MCA`).

Par contre cette option n'est pas présente dans les fonctions de base comme `mean`, `var`, `table` ou `chisq.test`.

7.2 Fonctions de l'extension rgrs

L'extension rgrs propose quelques fonctions permettant de calculer des statistiques simples pondérées[2] :

`wtd.mean` moyenne pondérée

`wtd.var` variance pondérée

`wtd.table` tris à plat et tris croisés pondérés

1. On notera que cette variable est utilisée à titre purement illustratif. Le jeu de données étant un extrait d'enquête et la variable de pondération n'ayant pas été recalculée, elle n'a ici à proprement parler aucun sens.
2. Les fonctions `wtd.mean` et `wtd.var` sont des copies conformes des fonctions du même nom de l'extension Hmisc de Frank Harrel. Hmisc étant une extension « de taille », on a préféré recopié les fonctions pour limiter le poids des dépendances.

On les utilise de la manière suivante :

```
R> mean(d$age)

[1] 48.16

R> wtd.mean(d$age, weights = d$poids)

[1] 46.35

R> wtd.var(d$age, weights = d$poids)

[1] 325.3
```

Pour les tris à plat, on utilise la fonction `wtd.table` à laquelle on passe la variable en paramètre :

```
R> wtd.table(d$sexe, weights = d$poids)

  Homme   Femme
5149382 5921844
```

Pour un tri croisé, il suffit de passer deux variables en paramètres :

```
R> wtd.table(d$sexe, d$hard.rock, weights = d$poids)

          Non     Oui
Homme 5109366   40016
Femme 5872596   49247
```

Ces fonctions admettent les deux options suivantes :

`na.rm` si `TRUE`, on ne conserve que les observations sans valeur manquante

`normwt` si `TRUE`, on normalisate les poids pour que les effectifs totaux pondérés soient les mêmes que les effectifs initiaux. Il faut utiliser cette option, notamment si on souhaite appliquer un test sensible aux effectifs comme le χ^2.

Ces fonctions rendent possibles l'utilisation des statistiques descriptives les plus simples et le traitement des tableaux croisés (les fonctions `lprop`, `cprop` ou `chisq.test` peuvent être appliquées au résultat d'un `wtd.table`) mais restent limitées en termes de tests statistiques ou de graphiques...

7.3 L'extension survey

L'extension survey est spécialement dédiée au traitement d'enquêtes ayant des techniques d'échantillonnage et de pondération potentiellement très complexes. L'extension s'installe comme la plupart des autres :

```
R> install.packages("survey", dep = TRUE)
```

Le site officiel (en anglais) comporte beaucoup d'informations, mais pas forcément très accessibles :

http://faculty.washington.edu/tlumley/survey/

Pour utiliser les fonctionnalités de l'extension, on doit d'abord définir un *design* de notre enquête. C'est-à-dire indiquer quel type de pondération nous souhaitons lui appliquer. Dans notre cas nous utilisons le *design* le plus simple, avec une variable de pondération déjà calculée. Ceci se fait à l'aide de la fonction `svydesign` :

```
R> library(survey)


Attaching package: 'survey'


The following object(s) are masked from 'package:graphics':

    dotchart


R> dw <- svydesign(ids = ~1, data = d, weights = ~d$poids)
```

Cette fonction crée un nouvel objet, que nous avons nommé `dw`. Cet objet n'est pas à proprement parler un tableau de données, mais plutôt un tableau de données *plus* une méthode de pondération. `dw` et `d` sont des objets distincts, les opérations effectuées sur l'un n'ont pas d'influence sur l'autre. On peut cependant retrouver le contenu de `d` depuis `dw` en utilisant `dw$variables` :

```
R> mean(d$age)

[1] 48.16

R> mean(dw$variables$age)

[1] 48.16
```

Lorsque notre *design* est déclaré, on peut lui appliquer une série de fonctions permettant d'effectuer diverses opérations statistiques en tenant compte de la pondération. On citera notamment :

`svymean`, `svyvar`, `svytotal` statistiques univariées

`svytable` tableaux croisés

`svyglm` modèles linéaires généralisés

`svyplot`, `svyhist`, `svyboxplot` fonctions graphiques

D'autres fonctions sont disponibles, comme `svyratio` ou `svyby`, mais elles ne seront pas abordées ici.

Pour ne rien arranger, ces fonctions prennent leurs arguments sous forme de formules, c'est-à-dire pas de la manière habituelle. En général l'appel de fonction se fait en spécifiant d'abord les variables d'intérêt sous forme de formule, puis l'objet *design*.

Voyons tout de suite quelques exemples :

```
R> svymean(~age, dw)

    mean   SE
age 46.3 0.53

R> svyvar(~heures.tv, dw, na.rm = TRUE)
```

```
          variance   SE
heures.tv     2.99 0.18

R> svytable(~sexe, dw)

sexe
  Homme   Femme
5149382 5921844

R> svytable(~sexe + clso, dw)

       clso
sexe        Oui     Non Ne sait pas
  Homme 2658744 2418188       72451
  Femme 2602032 3242389       77423
```

En particulier, les tris à plat se déclarent en passant comme argument le nom de la variable précédé d'un symbole `~`, tandis que les tableaux croisés utilisent les noms des deux variables séparés par un `+` et précédés par un `~`.

On peut récupérer le tableau issu de `svytable` dans un objet et le réutiliser ensuite comme n'importe quel tableau croisé :

```
R> tab <- svytable(~sexe + clso, dw)
R> tab

       clso
sexe        Oui     Non Ne sait pas
  Homme 2658744 2418188       72451
  Femme 2602032 3242389       77423

R> lprop(tab)

          clso
sexe       Oui   Non   Ne sait pas Total
  Homme     51.6  47.0   1.4       100.0
  Femme     43.9  54.8   1.3       100.0
  Ensemble  47.5  51.1   1.4       100.0

R> chisq.test(tab)


Pearson's Chi-squared test

data:  tab
X-squared = 67214, df = 2, p-value < 2.2e-16
```

Les fonctions `lprop`, `cprop` et `residus` de rgrs sont donc tout à fait compatibles avec l'utilisation de survey. La fonction `freq` peut également être utilisée si on lui passe en argument non pas la variable elle-même, mais son tri à plat obtenu avec `svytable` :

rgrs

```
R> par(mfrow = c(2, 2))
R> svyplot(~age + heures.tv, dw, col = "red", main = "Bubble plot")
R> svyhist(~heures.tv, dw, col = "peachpuff", main = "Histogramme")
R> svyboxplot(age ~ 1, dw, main = "Boxplot simple", ylab = "Âge")
R> svyboxplot(age ~ sexe, dw, main = "Boxplot double", ylab = "Âge",
+     xlab = "Sexe")
```

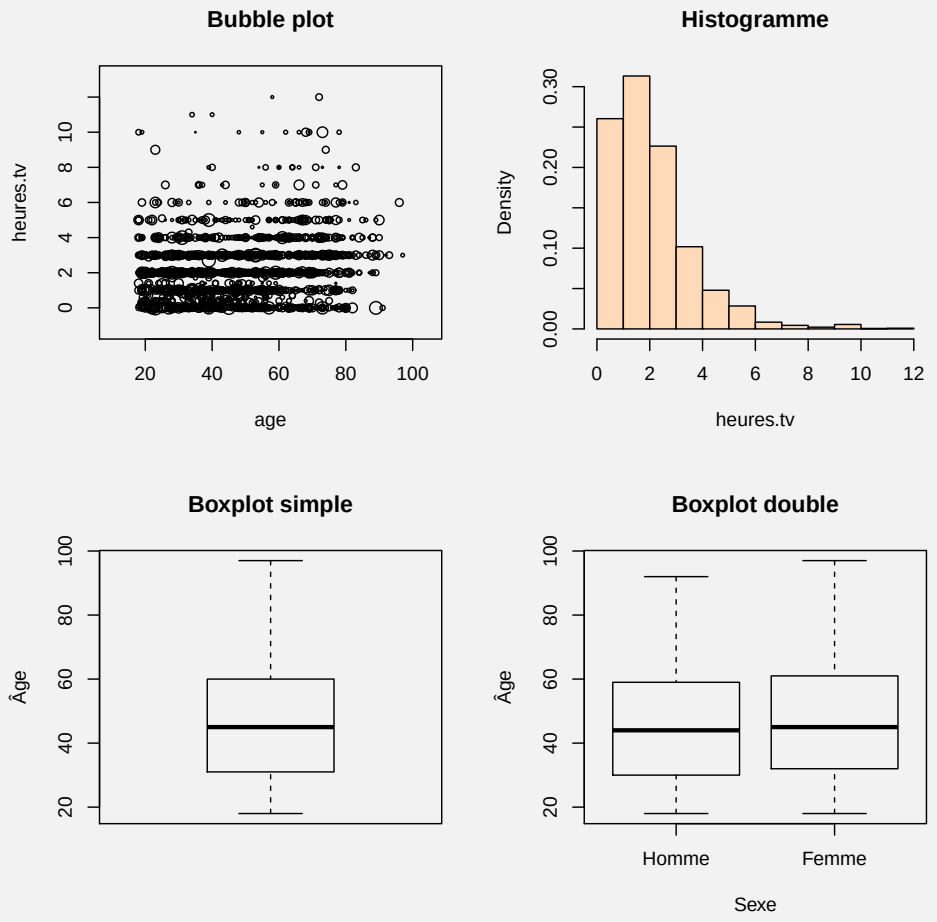


FIGURE 7.1 – Fonctions graphiques de l'extension survey

```
R> tab <- svytable(~peche.chasse, dw)
R> freq(tab, total = TRUE)

              n     %
Non     9716683  87.8
Oui     1354544  12.2
Total 11071226 100.0
```

Enfin, survey est également capable de produire des graphiques à partir des données pondérées. Des exemples sont donnés figure 7.1 de la présente page.

7.4 Conclusion

En attendant mieux, la gestion de la pondération sous R n'est sans doute pas ce qui se fait de plus pratique et de plus simple. On pourra quand même donner les conseils suivants :

- utiliser les options de pondération des fonctions usuelles ou les fonctions de l'extension rgrs pour les cas les plus simples ;
- si on utilise survey, effectuer tous les recodages et manipulations sur les données non pondérées autant que possible ;
- une fois les recodages effectués, on déclare le *design* et on fait les analyses en tenant compte de la pondération ;
- surtout ne jamais modifier les variables du *design*. Toujours effectuer recodages et manipulations sur les données pondérées, puis redéclarer le *design* pour que les mises à jour effectuées soient disponibles pour l'analyse ;

Partie 8

Cartographie

Cette partie aborde l'utilisation de R pour la création de cartes simples permettant la représentation d'effectifs, de proportions ou de variables qualitatives pour des zonages géographiques (régions, communes, Iris...). Ceci ne constitue qu'une infime partie des possibilités de l'analyse spatiale de données.

Par ailleurs, le parti pris est ici de tout effectuer à l'intérieur de R, sans faire appel à des applications externes spécialisées comme Quantum GIS, gvSig ou GRASS.

rgrs Les fonctions présentées ici font partie pour la plupart de l'extension rgrs , mais elles ne sont que des interfaces visant à faciliter l'utilisation de fonctions disponibles dans des extensions spécialisées, en particulier l'extension sp.

8.1 Données spatiales

Sous R, une carte est un objet comme un autre, seulement un peu plus compliqué. Le stockage des données utilisé dans cette partie repose sur la classe d'objets nommée `SpatialPolygonsDataFrame`, définie par l'extension sp. Ce type d'objet, particulièrement complexe, peut contenir à la fois des données de type spatial (sous la forme d'une liste de polygones) et des données classiques sous la forme d'un tableau de données.

8.1.1 Exemple d'objet spatial

rgrs L'extension rgrs fournit un exemple d'objet de ce type, nommé `lyon`, et qui contient le contour des 9 arrondissements de cette commune. On peut le charger dans R de la manière suivante :

```
R> library(rgrs)
R> data(lyon)
```

Nous avons désormais à notre disposition un objet nommé `lyon` que nous pouvons tout de suite représenter graphiquement à l'aide de la fonction `plot`. Le résultat est indiqué figure 8.1 page suivante.

Si l'on veut étudier la structure de l'objet `lyon`, par exemple en effectuant un `str(lyon)`, on se rend vite compte de la complexité de ce type d'objets. En fait `lyon` est lui-même composé de plusieurs « sous-objets » (*slots*) accessibles avec l'opérateur `@`. Nous décrirons les trois principaux :

`lyon@data` est un tableau de données dont chaque ligne correspond à un des polygones (c'est à dire ici à un arrondissement de Lyon) et qui lui associe un certain nombre de données (identifiant, nom de l'arrondissement, etc.) ;

```
R> sp::plot(lyon)
```

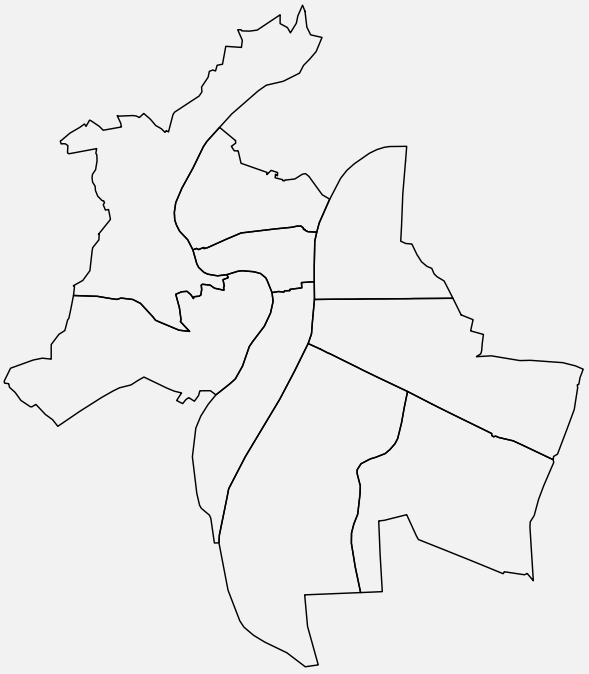


FIGURE 8.1 – `plot` d'un objet de type spatial

lyon@polygons est une liste de polygones définissant les contours de chaque arrondissement ;
proj4string est un objet décrivant la projection géographique utilisée pour les données spatiales.

On n'accèdera en général jamais directement à ces sous-objets, à part éventuellement **data**, que l'on peut manipuler comme n'importe quel tableau de données :

```
R> str(lyon@data)

'data.frame': 9 obs. of  2 variables:
 $ DepCom : Factor w/ 301 levels "69001","69002",..: 293 294 295 296 297 298 299 300 301
 $ Nom_Com: chr  "LYON 1ER" "LYON 2E" "LYON 3E" "LYON 4E" ...

R> lyon@data$Nom_Com

[1] "LYON 1ER" "LYON 2E"  "LYON 3E"  "LYON 4E"  "LYON 5E"  "LYON 6E"  "LYON 7E"
[8] "LYON 8E"  "LYON 9E"
```

8.1.2 Importer des données spatiales

Cette section est relativement technique et peut-être sautée si l'on souhaite juste avoir un aperçu des fonctionnalités présentées en utilisant les données incluses dans rgrs.

La conversion et l'import de données de type spatial sont des opérations relativement complexes, notamment du fait qu'elles mettent en jeu des notions de projection pas toujours faciles à comprendre et à maîtriser pour des non-spécialistes [1].

R propose cependant de nombreux outils pour importer des données de différents formats, notamment *via* les extensions maptools et rgdal.

On n'entrera pas ici dans le détail de ces opérations (que nous ne maîtrisons guère). Mais voici cependant, à titre indicatif, la marche à suivre pour importer dans R des données de l'IGN telles que fournies par le Centre Maurice Halbwachs (contours Iris de différents départements par exemple).

Les données étant fournies au format MapInfo, la première opération consiste à les convertir en format ESRI Shapefile. Sous Linux cela se fait très facilement grâce à **ogr2ogr** avec une commande du type :

```
ogr2ogr -f "ESRI Shapefile" 69_iris.shp 69_iris.mid
```

Ceci devrait vous générer trois fichiers portant le même nom mais avec les extensions **shp**, **shx** et **dbf**. L'import dans R peut alors s'effectuer de la manière suivante :

```
library(maptools)
library(rgdal)
library(foreign)

## Projection d'origine
proj.string <- "+init=epsg:27572 +proj=lcc
+lat_1=45.90287723937\n+lat_2=47.69712276063 +lat_0=46.8
+lon_0=2.337229104484\n+x_0=600000 +y_0=2200000 +units=m +pm=greenwich"
```

1. Et en premier lieu par l'auteur de ces lignes.

```
## Projection d'arrivée
proj.string.geo <- "+proj=longlat +datum=WGS84"

## Import du fichier
rhone.iris <- readShapePoly("69_iris.shp", proj4string = CRS(proj.string))

## Transformation de la projection
rhone.iris <- spTransform(rhone.iris, CRS(proj.string.geo))

## Conversion en Unicode
rhone.iris$Nom_Com <- iconv(rhone.iris$Nom_Com, from = "latin1",
    to = "utf8")
rhone.iris$Nom_Iris <- iconv(rhone.iris$Nom_Iris, from = "latin1",
    to = "utf8")

## Sauvegarde
save(rhone.iris, file = "rhone_iris.rda")
```

On peut ensuite charger le contenu du fichier `rhone_iris.rda` à l'aide de la fonction `load` dans un autre script.

8.2 Cartes simples

Dans ce qui suit on se base sur la carte des arrondissements de Lyon et sur l'extrait du recensement 1999 pour les communes du Rhône. Ces données peuvent être chargées avec les commandes suivantes :

```
R> data(lyon)
R> data(rp99)
```

8.2.1 Représentation de proportions

Nous disposons donc, d'un côté, d'un objet spatial représentant les arrondissements de Lyon, et de l'autre d'un tableau de données contenant un extrait du recensement de 1999 pour les communes du Rhône. Si on regarde un peu plus attentivement la structure de ces deux objets :

```
R> str(lyon@data)

'data.frame': 9 obs. of  2 variables:
 $ DepCom : Factor w/ 301 levels "69001","69002",..: 293 294 295 296 297 298 299 300 301
 $ Nom_Com: chr  "LYON 1ER" "LYON 2E" "LYON 3E" "LYON 4E" ...

R> head(rp99$code, 20)

 [1] 69001 69002 69003 69004 69005 69006 69007 69008 69009 69010 69012 69013 69014
[14] 69015 69016 69017 69018 69019 69020 69021
```

On se rend compte que les deux objets peuvent être « joints » grâce à un champ contenant le code INSEE de chaque commune ou arrondissement. Ce champ se nomme `DepCom` pour l'objet `lyon`, et `code` pour l'objet `rp99`.

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code")

Loading required package: sp

Loading required package: RColorBrewer
```

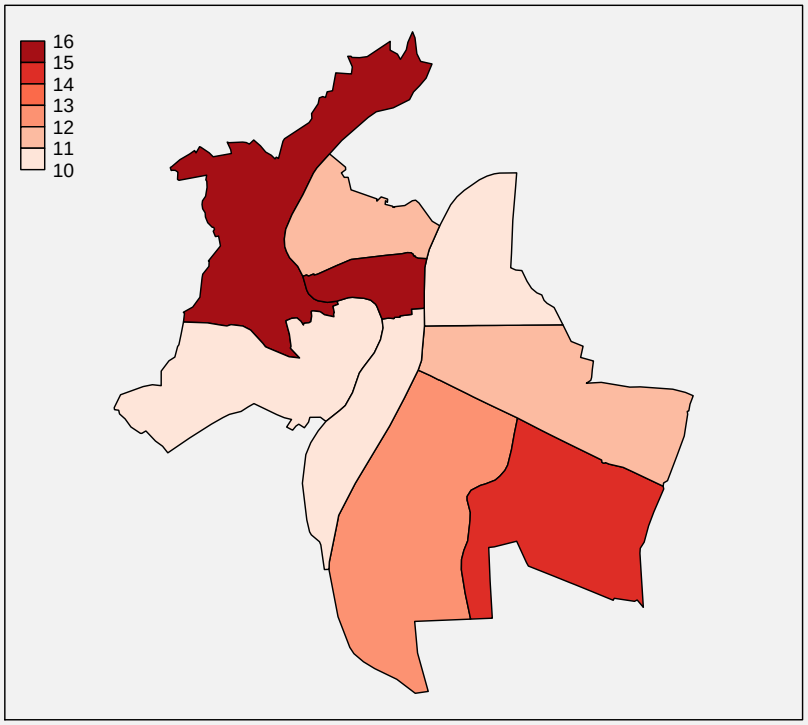


FIGURE 8.2 – Exemple d'utilisation de `carte.prop`

Nous avons dès lors tout ce qu'il nous faut pour afficher une première carte, en l'occurrence le taux de chômage par arrondissement en 1999. On utilise pour cela la fonction `carte.prop` de `rgrs`. Le code et le résultat sont indiqués figure 8.2 de la présente page.

La fonction `carte.prop` admet les arguments suivants :

- le premier argument est l'objet de type spatial contenant les données cartographiques, ici `lyon` ;
- le second argument est le tableau de données contenant les variables à cartographier, ici `rp99` ;
- le troisième argument est le nom de la variable à représenter, ici `"tx.chom"` ;
- l'argument `sp.key` correspond au nom du champ de jointure dans l'objet spatial ;
- l'argument `data.key` correspond au nom du champ de jointure dans le tableau de données.

Le nombre de classes de valeurs et les limites de ces classes sont calculés automatiquement. On peut

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", nbcuts = 3)
```

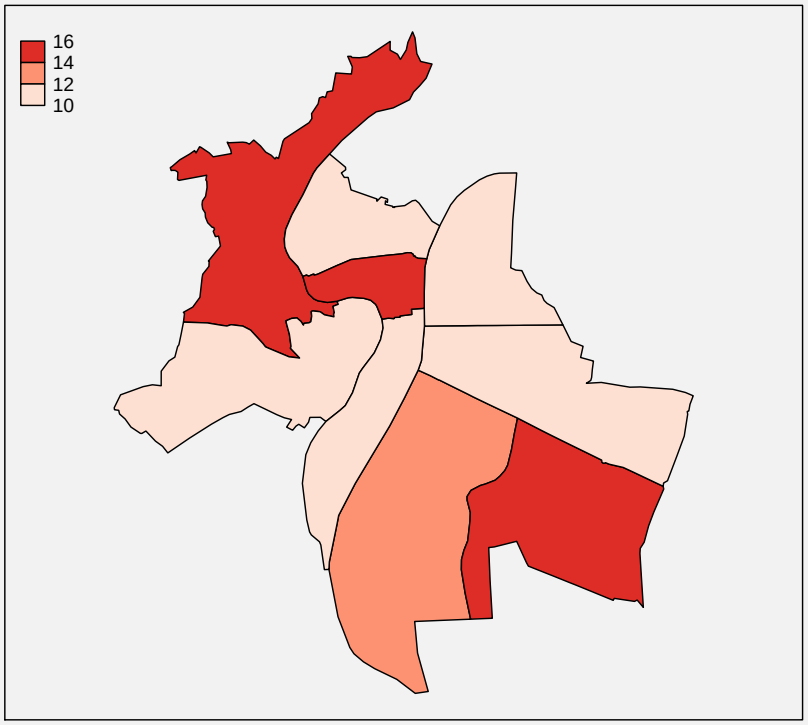


FIGURE 8.3 – Utilisation de l'argument `nbcuts` de `carte.prop`

cependant spécifier soit un nombre de classes à l'aide de l'argument `nbcuts` [2], soit les limites de ces classes avec l'argument `at`, comme indiqué figure 8.3 de la présente page, et figure 8.4 page suivante.

De nombreuses options sont disponibles pour personnaliser l'affichage de la carte. On pourra citer :

`main` titre de la carte ;

`sub` sous-titre de la carte ;

`posleg` position de la légende, spécifiée sous forme de d'une chaîne de caractères : `bottomright`, `topleft`, `left`, `center`, ... ;

`diverg` si `TRUE`, indique que la carte représente à la fois des valeurs négatives et positives ;

`palette.pos` palette de couleur utilisée pour les valeurs positives ;

`palette.neg` palette de couleur utilisée pour les valeurs négatives ;

`palette` palette de couleur spécifiée manuellement ;

2. Du fait que la fonction essaye d'établir des limites de classes « propres », correpsondant par exemple à des nombres entiers, le nombre de classes indiqué n'est pas toujours respecté.

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", at = c(10, 10.5, 11, 11.5, 12, 15))
```

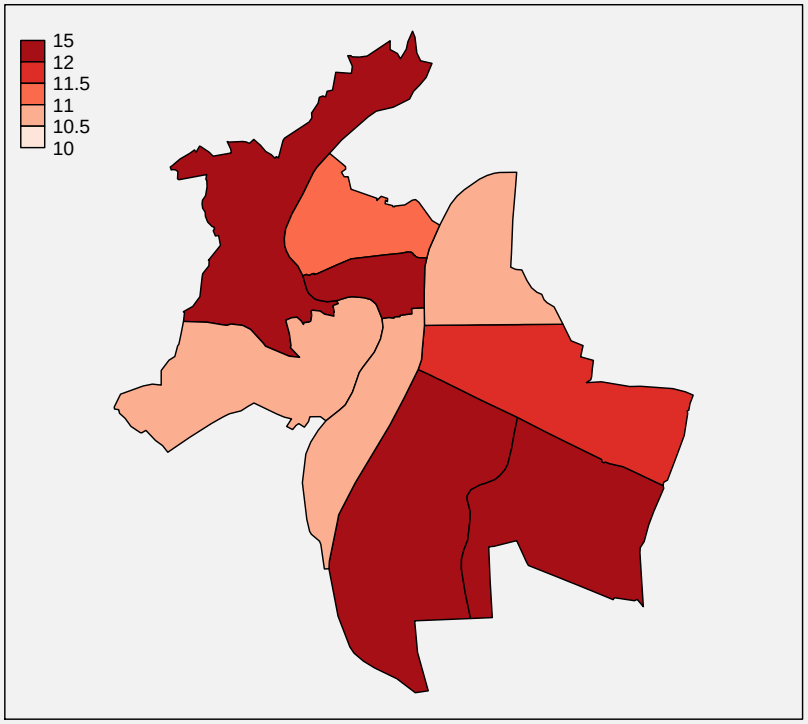


FIGURE 8.4 – Utilisation de l'argument `at` de `carte.prop`

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", main = "Taux de chômage1999", sub = "Source : INSEE, RP
1999",
+     palette.pos = "RdPu", posleg = "topright")
```

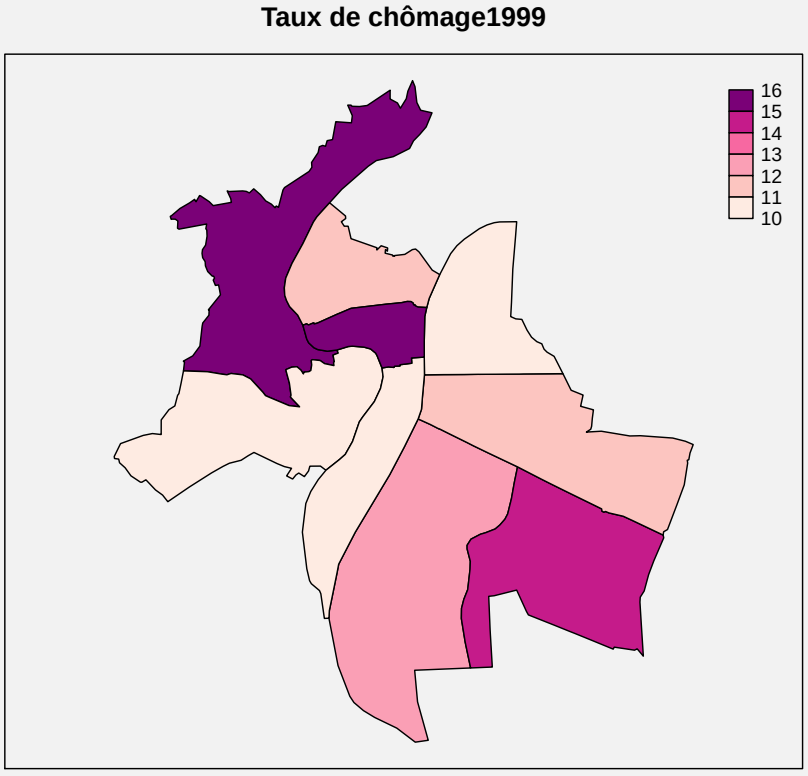


FIGURE 8.5 – Personnalisations de l'affichage de `carte.prop`

Enfin, tout argument supplémentaire est transmis à la fonction `spplot` pour l'affichage de la carte.

Les palettes utilisées en argument de `palette.pos` et `palette.neg` sont celles définies par l'extension RColorBrewer, elle-même issue du projet *Colorbrewer* :

http://www.colorbrewer.org

Le site du projet propose notamment un outil qui permet de visualiser et de choisir une palette de manière interactive :

http://www.personal.psu.edu/cab38/ColorBrewer/ColorBrewer.html

Les noms de palettes passés en argument de `palette.pos` et `palette.neg` sont les mêmes que ceux utilisés sur ce site.

Un exemple d'utilisation de ces différents paramètres est donné figure 8.5 de la présente page.

```
R> carte.eff(lyon, rp99, "pop.act", sp.key = "DepCom", data.key = "code")
```

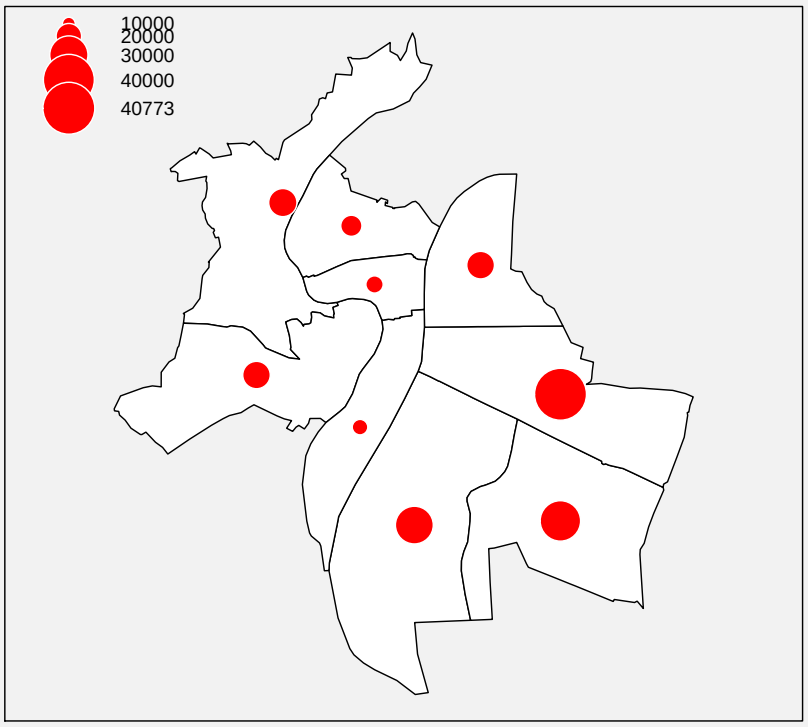


FIGURE 8.6 – Exemple d'utilisation de `carte.eff`

8.2.2 Représentation d'effectifs

Contrairement à la reprséentation de proportions, qui s'effectue en affectant une couleur à chaque polygone, les effectifs ou les populations sont en général représentés par un symbole de taille variable. Ce type de carte peut être obtenue avec la fonction `carte.eff`.

Son utilisation est très semblable à celle de `carte.prop` pour ses arguments principaux. Un exemple utilisant les options par défaut et représentant la population active de chaque arrondissement est donné figure 8.6 de la présente page.

Les options `nbcuts` et `at` sont disponibles de la même manière que pour `carte.prop`, mais elles n'agissent ici que sur la présentation de la légende.

D'autres options de personnalisation sont également disponibles. On retrouve les options `main`, `sub` et `posleg` déjà décrites pour `carte.prop`, ainsi que les options suivantes :

`col.bg` couleur des symboles (rouge par défaut) ;

`col.border` couleur de la bordure des symboles (blanc par défaut) ;

`cex` facteur d'agrandissement des symboles ;

```
R> carte.eff(lyon, rp99, "pop.act", sp.key = "DepCom", data.key = "code",
+     main = "Population active en 1999", sub = "Source : INSEE, RP 1999",
+     pch = 23, cex = 10, col.bg = "blue", col.border = "yellow",
+     posleg = "topright")
```

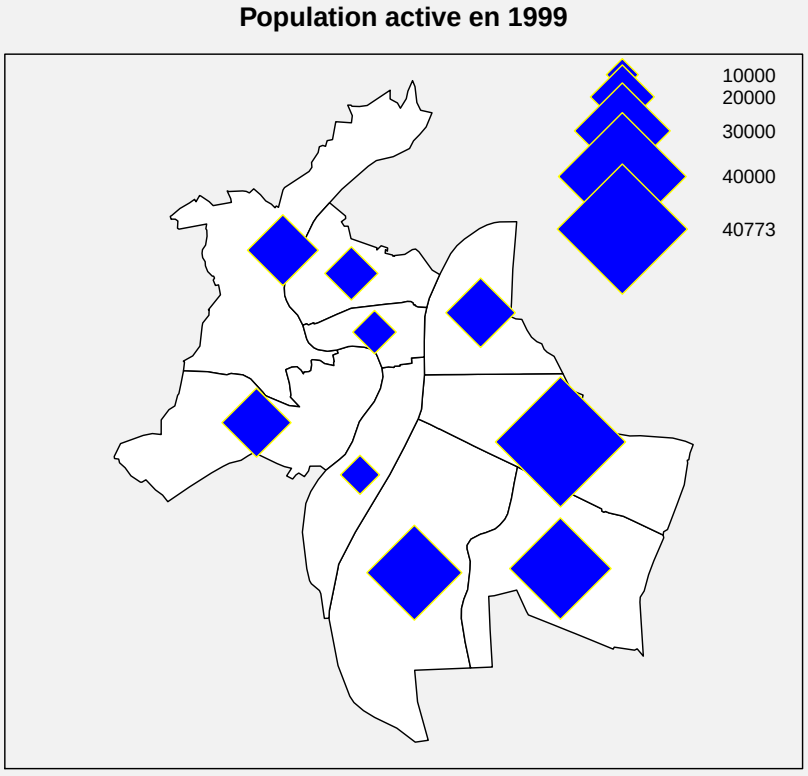


FIGURE 8.7 – Personnalisations de l'affichage de `carte.eff`

`pch` symbole utilisé ;

`plot.polygons` si `FALSE`, on n'affiche que les symboles et pas les polygones.

Un exemple d'utilisation de ces différents paramètres est donné figure 8.7 de la présente page.

8.2.3 Représentation d'une variable qualitative

La représentation d'une variable qualitative (typiquement le résultat d'une classification) se fait généralement de la même manière que la représentation d'une proportion, mais en utilisant une palette de couleurs contrastées et n'induisant pas de « hiérarchie » entre les objets représentés. Ce type de carte peut être obtenue avec la fonction `carte.qual`.

Là encore, son utilisation est très semblable à celle de `carte.prop` et `carte.eff`. Un exemple utilisant les options par défaut et représentant la population active de chaque arrondissement est donné figure 8.8

```
R> rp99$qual <- sample(c("A", "B", "C", "D", "E"), nrow(rp99),
+     replace = TRUE)
R> carte.qual(lyon, rp99, "qual", sp.key = "DepCom", data.key = "code")
```

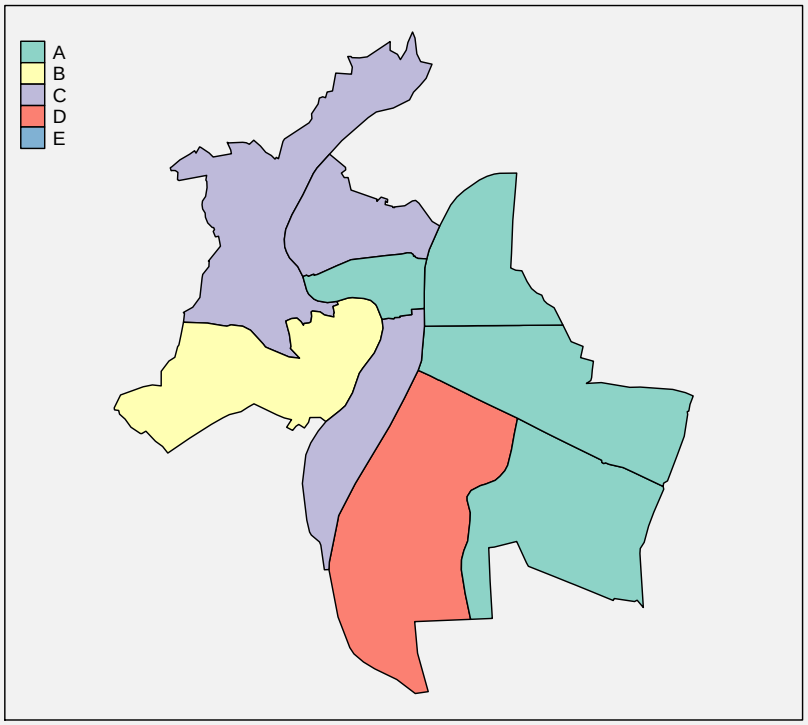


FIGURE 8.8 – Exemple d'utilisation de `carte.qual`

de la présente page. Le tableau de données `rp99` ne contenant pas de variable de type qualitative, on a simulé les données de manière tout à fait aléatoire.

Là encore, plusieurs options sont disponibles pour personnaliser l'affichage de la carte. Ce sont en fait les mêmes options que celles de `carte.prop` (`main`, `sub`, `posleg`, `palette`...), ainsi que l'option `palette.qual`, qui permet de sélectionner une palette de l'extension `RColorBrewer`. Un exemple de personnalisation est donné figure 8.9 page suivante.

8.3 Ajout d'éléments à une carte

Il est tout à fait possible de superposer d'autres éléments graphiques à une carte. On utilise alors en général directement la fonction `plot` munie de l'argument `add=TRUE`.

```
R> carte.qual(lyon, rp99, "qual", sp.key = "DepCom", data.key = "code",
+     main = "Catégories d'arrondissements", sub = "Source : aléatoire",
+     posleg = "bottomright", palette.qual = "Set2")
```

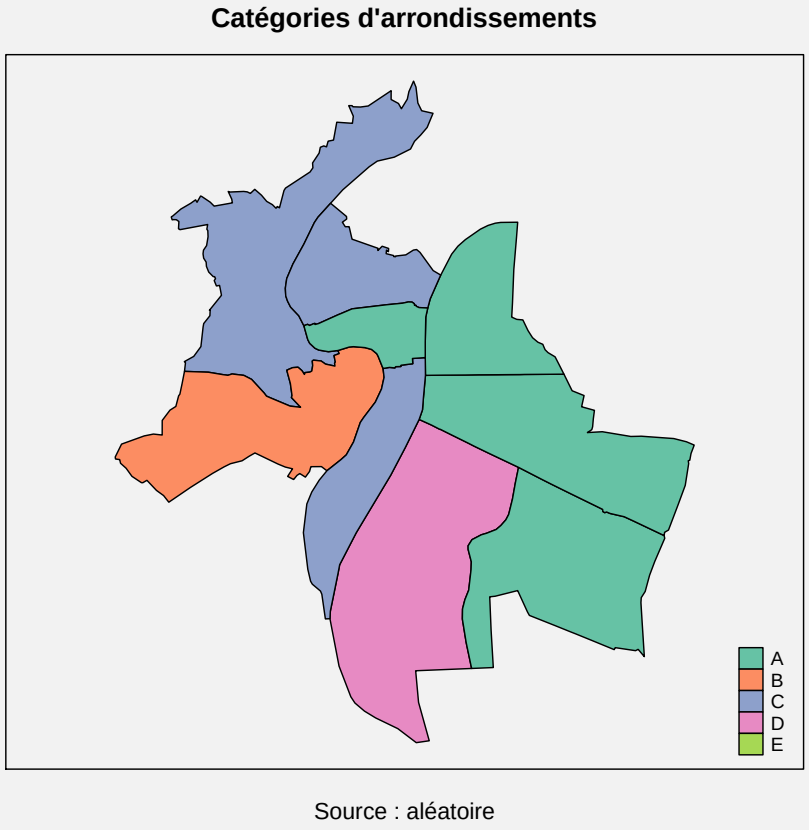


FIGURE 8.9 – Exemple de personnalisation de `carte.qual`

8.3.1 Bordure

Il peut être intéressant d'ajouter une bordure à une carte soit pour des raisons esthétiques (par exemple pour délimiter son contour), soit pour mettre en valeur une zone particulière.

Imaginons par exemple que nous souhaitons mettre en valeur le premier arrondissement de Lyon sur notre carte. Un objet de type spatial, malgré sa complexité, supporte l'indexation d'une manière semblable aux tableaux de données. On peut donc sélectionner un sous-ensemble de notre carte des arrondissements de Lyon de la manière suivante :

```
R> lyon1 <- lyon[lyon@data$Nom_Com == "LYON 1ER", ]
```

Cette commande sélectionne le polygone dont le nom est `LYON 1ER` et le place dans un nouvel objet. Il est alors très simple de rajouter une bordure autour de cet arrondissement en utilisant la fonction `plot`, comme indiqué figure 8.10 page ci-contre.

Si on souhaite délimiter le contour de notre carte avec une bordure, on doit effectuer une opération supplémentaire, qui consiste à « fusionner » l'ensemble des polygones de notre carte pour n'en garder que le contour. Cette opération peut se faire dans R à l'aide de la fonction `unionSpatialPolygons` de l'extension maptools. Celle-ci permet de fusionner les polygones d'une carte en fonction des valeurs d'une variable : les polygones ayant la même valeur sont alors regroupés pour n'en former plus qu'un.

Dans notre cas, nous voulons fusionner tous les polygones, nous pouvons donc créer une variable artificielle (ici nommée `fusion`) contenant la même valeur pour toutes nos zones, puis appliquer la fonction `unionSpatialPolygons`, ce qui donne :

```
R> library(maptools)

Loading required package: foreign

Loading required package: lattice

R> gpclibPermit()

General Polygon Clipper Library for R (version 1.5-1)
Type 'class ? gpc.poly' for help


R> fusion <- rep(1, nrow(lyon@data))
R> lyon.contour <- unionSpatialPolygons(lyon, fusion)
```

On peut ensuite utiliser l'objet ainsi calculé pour ajouter une bordure globale à notre carte, comme dans la figure 8.11 page 112.

8.3.2 Labels

L'ajout de labels est souvent indispensable pour augmenter la lisibilité d'une carte et permettre le repérage des zonages géographiques représentés. La fonction `carte.labels` est faite pour cela.

Elle accepte comme principaux arguments :

- le nom de l'objet spatial ;
- un vecteur de chaînes de caractères contenant les labels ;

On peut spécifier les coordonnées de placement des labels via l'argument `coords`. Si cet argument vaut `NULL` (ce qui est le cas par défaut), la position des labels est automatiquement calculée en fonction du polygone auquel il appartient (ce qui n'exclut cependant pas les chevauchements).

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", main = "Taux de chômage 1999")
R> lyon1 <- lyon[lyon@data$Nom_Com == "LYON 1ER", ]
R> plot(lyon1, lwd = 5, border = "red", add = TRUE)
```

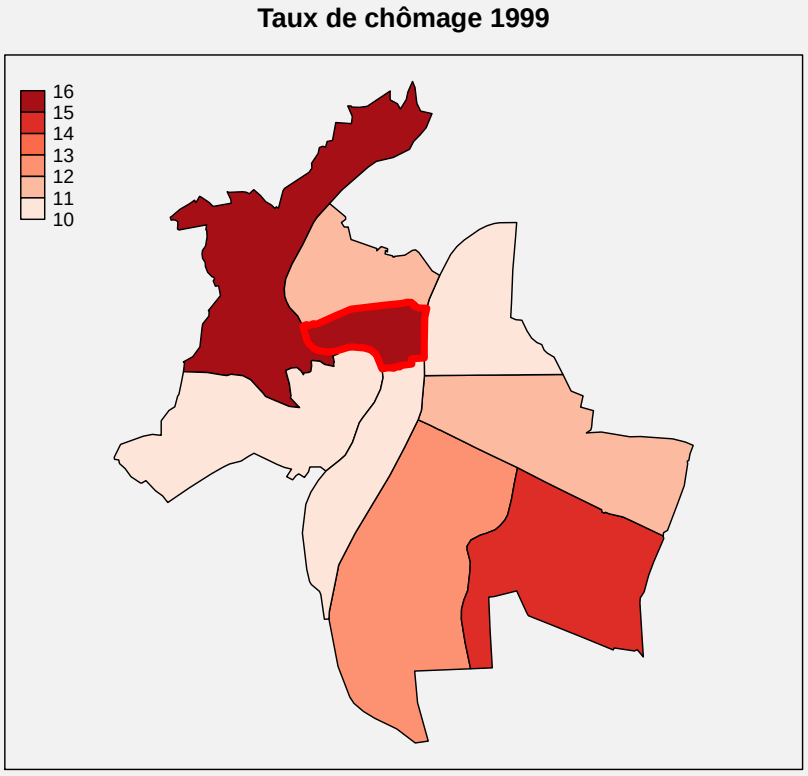


FIGURE 8.10 – Exemple d'ajout d'une bordure autour d'une zone

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", main = "Taux de chômage 1999")
R> plot(lyon1, lwd = 5, border = "red", add = TRUE)
R> plot(lyon.contour, lwd = 3, border = "black", add = TRUE)
```

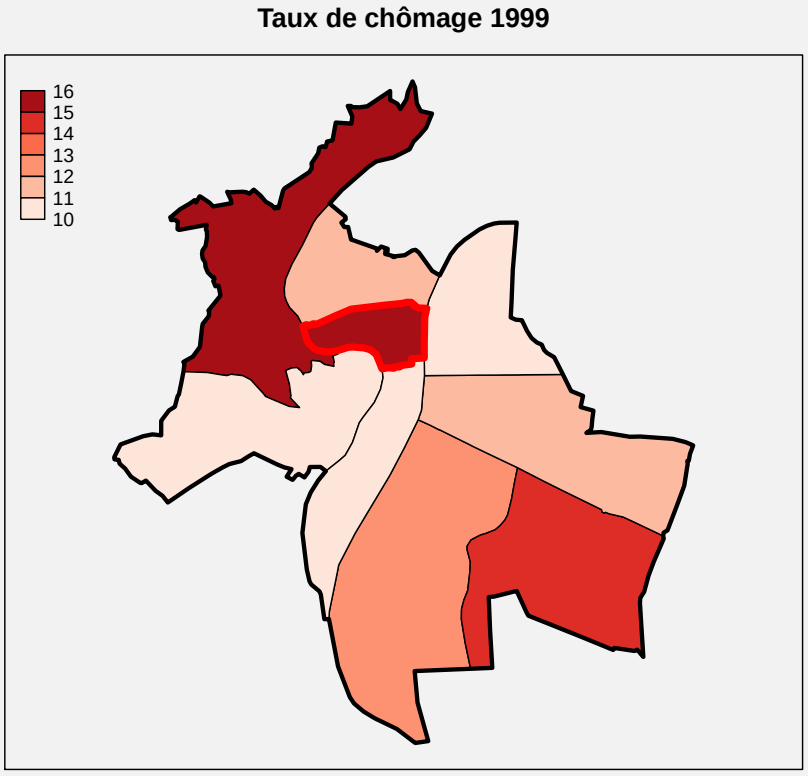


FIGURE 8.11 – Exemple d'ajout d'une bordure globale

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", main = "Taux de chômage 1999")
R> carte.labels(lyon, lyon@data$Nom_Com)
```

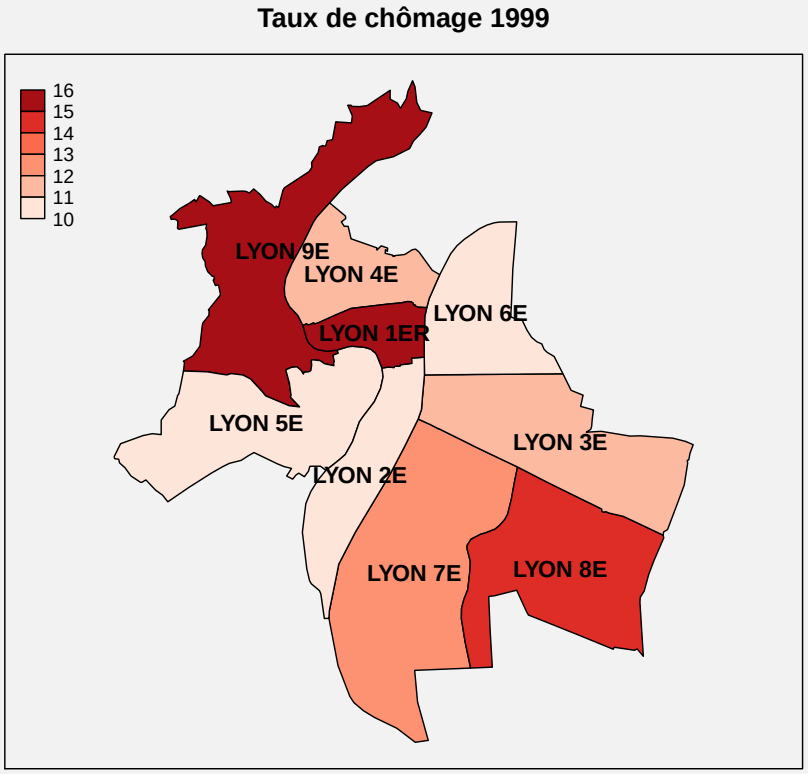


FIGURE 8.12 – Exemple d'ajout de labels

La figure 8.12 de la présente page montre comment ajouter les noms des arrondissements à notre carte de Lyon.

Plusieurs options sont disponibles pour personnaliser l'affichage des labels, notamment :

cex facteur d'agrandissement ;

font style de police de caractère (gras par défaut). Voir la page d'aide de **par** pour plus de détails ;

col couleur du texte ;

outline, outline.decal, outline.col permettent d'ajouter une « bordure » autour des labels.

La figure 8.13 page suivante montre comment ajouter la valeur du taux de chômage à notre carte, en leur ajoutant une bordure blanche.

```
R> carte.prop(lyon, rp99, "tx.chom", sp.key = "DepCom",
+     data.key = "code", main = "Taux de chômage 1999")
R> lyon.tx.chom <- round(rp99$tx.chom[rp99$code %in% lyon@data$DepCom],
+     1)
R> carte.labels(lyon, lyon.tx.chom, outline = TRUE)
```

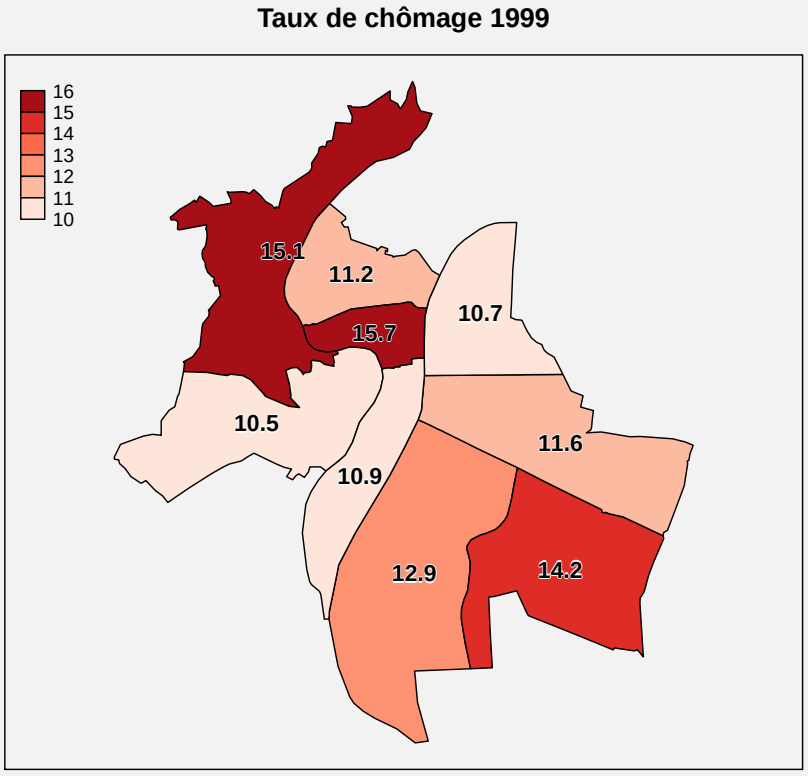


FIGURE 8.13 – Exemple d'ajout de labels personnalisés

Partie 9

Exporter les résultats

Cette partie décrit comment, une fois les analyses réalisées, on peut exporter les résultats (tableaux et graphiques) dans un traitement de texte ou une application externe.

9.1 Export manuel de tableaux

Les tableaux générés par R (et plus largement, tous les types d'objets) peuvent être exportés pour inclusion dans un traitement de texte à l'aide de la fonction `copie` de l'extension rgrs[1]. *rgrs*

Il suffit pour cela de lui passer en argument le tableau ou l'objet qu'on souhaite exporter. Dans ce qui suit on utilisera le tableau suivant, placé dans un objet nommé `tab` :

```
R> data(hdv2003)
R> tab <- table(hdv2003$sexe, hdv2003$bricol)
R> tab



        Non Oui
  Homme 384 515
  Femme 763 338
```

9.1.1 Copier/coller vers Excel et Word *via* le presse-papier

La première possibilité est d'utiliser les options par défaut de `copie`. Celle-ci va alors transformer le tableau (ou l'objet) en `HTML` et placer le résultat dans le presse papier du système. Ceci ne fonctionne malheureusement que sous Windows[2].

```
R> copie(tab)

Loading required package: R2HTML
```

On peut ensuite récupérer le résultat dans une feuille Excel en effectuant un simple *Coller*.

1. Celle-ci nécessite que l'extension `R2HTML` soit également installée sur le système *via* `install.packages("R2HTML",dep=TRUE)`.
2. En fait cela fonctionne aussi sous Linux si le programme `xclip` est installé et accessible. Cela fonctionne peut-être aussi sous Mac OS X mais n'a pas pu être testé.

	A	B	C
1		**Non**	**Oui**
2	Homme	383	511
3	Femme	771	335

On peut ensuite sélectionner le tableau sous Excel, le copier et le coller dans Word :

¤	Non¤	Oui¤
Homme¤	383	511¤
Femme¤	771	335¤

9.1.2 Export vers Word ou OpenOffice/LibreOffice *via* un fichier

L'autre possibilité ne nécessite pas de passer par Excel, et fonctionne sous Word, OpenOffice et LibreOffice sur toutes les plateformes.

Elle nécessite de passer à la fonction `copie` l'option `file=TRUE` qui enregistre le contenu de l'objet dans un fichier plutôt que de le placer dans le presse-papier :

```
R> copie(tab, file = TRUE)
```

Par défaut le résultat est placé dans un fichier nommé `temp.html` dans le répertoire courant, mais on peut modifier le nom et l'emplacement avec l'option `filename` :

```
R> copie(tab, file = TRUE, filename = "exports/tab1.html")
```

On peut ensuite l'intégrer directement dans Word ou dans OpenOffice en utilisant le menu *Insertion* puis *Fichier* et en sélectionnant le fichier de sortie généré précédemment.

	Non	Oui
Homme	383	511
Femme	771	335

9.2 Export de graphiques

9.2.1 Export *via* l'interface graphique (Windows ou Mac OS X)

L'export de graphiques est très simple si on utilise l'interface graphique sous Windows. En effet, les fenêtres graphiques possèdent un menu *Fichier* qui comporte une entrée *Sauver sous* et une entrée *Copier dans le presse papier*.

L'option *Sauver sous* donne le choix entre plusieurs formats de sortie, vectoriels (Metafile, Postscript) ou bitmaps (jpeg, png, tiff, etc.). Une fois l'image enregistrée on peut ensuite l'inclure dans n'importe quel document ou la retravailler avec un logiciel externe.

Une image *bitmap* est une image stockée sous forme de points, typiquement une photographie. Une image *vectorielle* est une image enregistrée dans un langage de description, typiquement un schéma ou une figure. Le second format présente l'avantage d'être en général beaucoup plus léger et d'être redimensionnable à l'infini sans perte de qualité. Pour plus d'informations voir http://fr.wikipedia.org/wiki/Image_matricielle et http://fr.wikipedia.org/wiki/Image_vectorielle.

L'option *Copier dans le presse papier* permet de placer le contenu de la fenêtre dans le presse-papier soit dans un format vectoriel soit dans un format bitmap. On peut ensuite récupérer le résultat dans un traitement de texte ou autre avec un simple *Coller*.

Des possibilités similaires sont offertes par l'interface sous Mac OS X, mais avec des formats proposés un peu différents.

9.2.2 Export avec les commandes de R

On peut également exporter les graphiques dans des fichiers de différents formats directement avec des commandes R. Ceci a l'avantage de fonctionner sur toutes les plateformes, et de faciliter la mise à jour du graphique exporté (on n'a qu'à relancer les commandes concernées pour que le fichier externe soit mis à jour).

La première possibilité est d'exporter le contenu d'une fenêtre déjà existante à l'aide de la fonction `dev.copy`. On doit fournir à celle-ci le format de l'export (option `device`) et le nom du fichier (option `file`). Par exemple :

```
R> boxplot(rnorm(100))
R> dev.copy(device = png, file = "export.png")
R> dev.off()
```

Les formats de sortie possibles varient selon les plateformes, mais on retrouve partout les formats bitmap bmp, jpeg, png, tiff, et les formats vectoriels postscript ou pdf. La liste complète disponible pour votre installation de R est disponible dans la page d'aide de `Devices` :

```
R> ?Devices
```

L'autre possibilité est de rediriger directement la sortie graphique dans un fichier, avant d'exécuter la commande générant la figure. On doit pour cela faire appel à l'une des commandes permettant cette redirection. Les plus courantes sont `bmp`, `png`, `jpeg` et `tiff` pour les formats bitmap, `postscript`, `pdf`, `svg`[3] et `win.metafile`[4] pour les formats vectoriels.

Ces fonctions prennent différentes options permettant de personnaliser la sortie graphique. Les plus courantes sont `width` et `height` qui donnent la largeur et la hauteur de l'image générée (en pixels pour les images bitmap, en pouces pour les images vectorielles), et `pointsize` qui donne la taille de base des polices de caractère utilisées.

```
R> png(file = "out.png", width = 800, height = 700)
R> plot(rnorm(100))
R> dev.off()
R>
R> pdf(file = "out.pdf", width = 9, height = 9, pointsize = 10)
R> plot(rnorm(150))
R> dev.off()
```

3. Ne fonctionne pas sous Windows.
4. Ne fonctionne que sous Windows.

Il est nécessaire de faire un appel à la fonction `dev.off` après génération du graphique pour que le résultat soit bien écrit dans le fichier de sortie (dans le cas contraire on se retrouve avec un fichier vide).

9.3 Génération automatique de rapports avec OpenOffice ou LibreOffice

Les méthodes précédentes permettent d'exporter tableaux et graphiques, mais cette opération reste manuelle, un peu laborieuse et répétitive, et surtout elle ne permet pas de mise à jour facile des documents externes en cas de modification des données analysées ou du code.

R et son extension odfWeave permettent de résoudre en partie ce problème. Le principe de base est d'inclure du code R dans un document de type traitement de texte, et de procéder ensuite au remplacement automatique du code par le résultat sous forme de texte, de tableau ou de figure.

9.3.1 Prérequis

odfWeave ne fonctionne qu'avec des documents au format OpenDocument (extension `.odf`), donc en particulier avec OpenOffice ou LibreOffice mais pas avec Word. L'utilisation d'OpenOffice est cependant très proche de celle de Word, et les documents générés peuvent être ensuite ouverts sous Word pour édition.

L'installation de l'extension se fait de manière tout à fait classique :

```
R> install.packages("odfWeave", dep = TRUE)
```

Un autre prérequis est de disposer d'applications permettant de compresser et décompresser des fichiers au format `zip`. Or ceci n'est pas le cas par défaut sous Windows. Pour les récupérer, téléchargez l'archive à l'adresse suivante :

http://alea.fr.eu.org/public/files/zip.zip

Décompressez-là et placez les deux fichiers qu'elle contient (`zip.exe` et `unzip.exe`) dans votre répertoire système, c'est à dire en général soit `c:\windows`, soit `c:\winnt`.

9.3.2 Exemple

Prenons tout de suite un petit exemple. Soit le fichier OpenOffice représenté figure 9.1 page ci-contre.

On voit qu'il contient à la fois du texte mis en forme (sous forme de titre notamment) mais aussi des passages plus ésotériques qui ressemblent plutôt à du code R.

Ce code est séparé du reste du texte par les caractères `«»=`, en haut, et `@`, en bas.

Créons maintenant un nouveau fichier R dans le même répertoire que notre fichier OpenOffice, et mettons-y le contenu suivant :

```
R> library(odfWeave)
R> odfWeave("odfWeave_exemple1.odt", "odfWeave_exemple1_out.odt")
```

Puis exécutons le tout... Nous devrions alors avoir un nouveau fichier nommé `odfWeave_exemple1_out.odt` dans notre répertoire de travail. Si on l'ouvre avec OpenOffice, on obtient le résultat indiqué figure 9.2 page suivante.

Que constate-t-on ? Le passage contenant du code R a été remplacé par le code R en question, de couleur bleue, et par son résultat, en rouge.

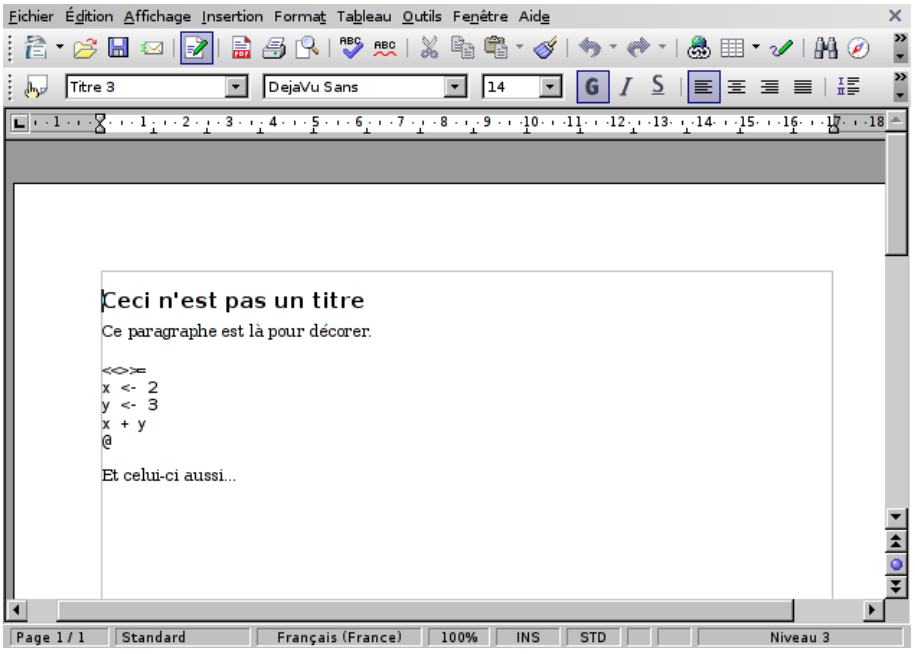


FIGURE 9.1 – Exemple de fichier odfWeave

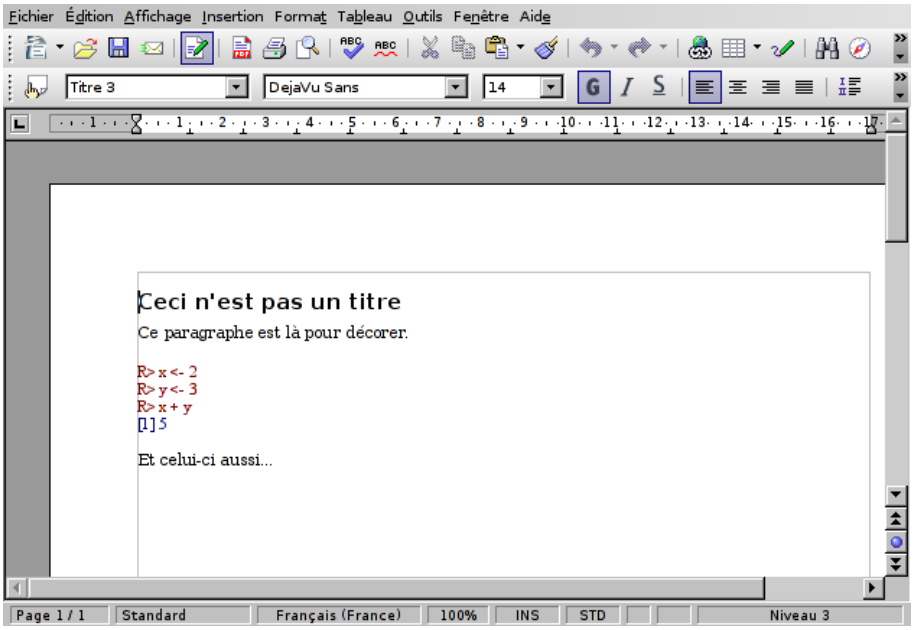


FIGURE 9.2 – Résultat de l'exemple de la figure 9.1

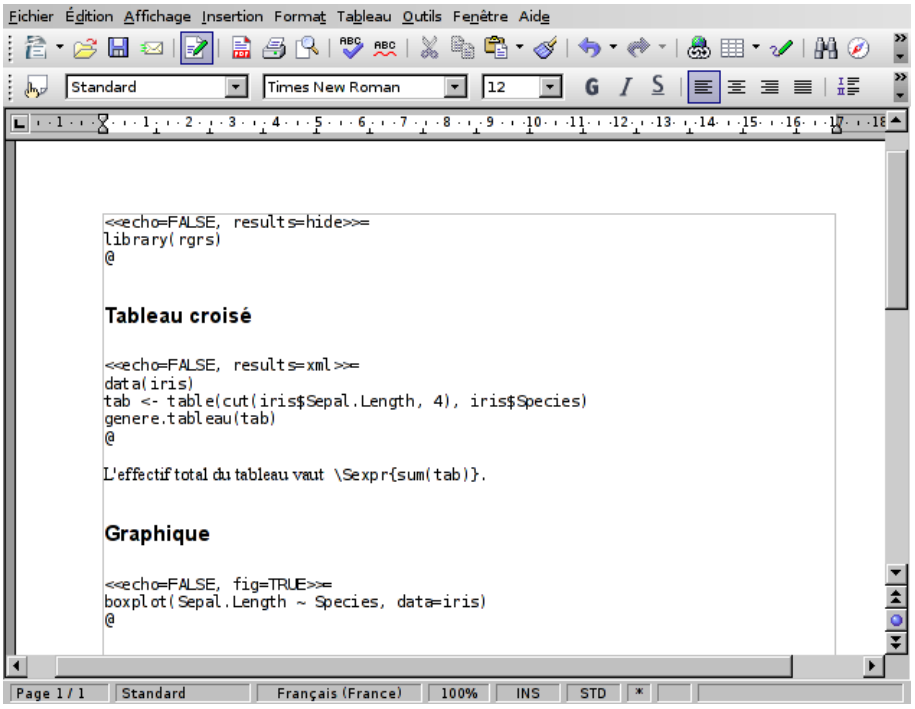


FIGURE 9.3 – Un fichier odfWeave un peu plus compliqué

Tout ceci est bien sympathique mais un peu limité. La figure 9.3 de la présente page, montre un exemple plus complexe, dont le résultat est indiqué figure 9.4, page suivante.

Le premier bloc de code R contient des options entre les séparateurs « et »=. L'option `echo=FALSE` supprime l'affichage du code R (en bleu) dans le document résultat. L'option `results=hide` supprime l'affichage du résultat du code (en rouge). Au final, le code `library(rgrs)` est exécuté, mais caché dans le document final.

Dans le deuxième bloc, l'option `results=xml` indique que le résultat du code ne sera pas du simple texte mais un objet déjà au format OpenOffice (en l'occurrence un tableau). Le code lui-même est ensuite
rgrs assez classique, sauf la dernière instruction `genere.tableau`, qui, appliquée à un objet de type `table`, produit le tableau mis en forme dans le document résultat.

Plus loin, on a dans le cours du texte une chaîne `\Sexprsum(tab)` qui a été remplacée par le résultat du code qu'elle contient.

Enfin, dans le dernier bloc, l'option `fig=TRUE` indique que le résultat sera cette fois une image. Et le bloc est bien remplacé par la figure correspondante dans le document final.

9.3.3 Utilisation

Le principe est donc le suivant : un document OpenOffice classique, avec du texte mis en forme, stylé et structuré de manière tout à fait libre, à l'intérieur duquel se trouve du code R. Ce code est délimité par les caractères «»= (avant le code) et @ (après le code). On peut indiquer des options concernant le bloc de code R entre les caractères « et » de la chaîne ouvrante. Parmi les options possibles les plus importantes sont :

`eval` si `TRUE` (par défaut), le bloc de code est exécuté. Sinon il est seulement affiché et ne produit pas de résultat.

`echo` si `TRUE` (par défaut), le code R du bloc est affiché dans le document résultat (par défaut en bleu). Si `FALSE`, le code est masqué.

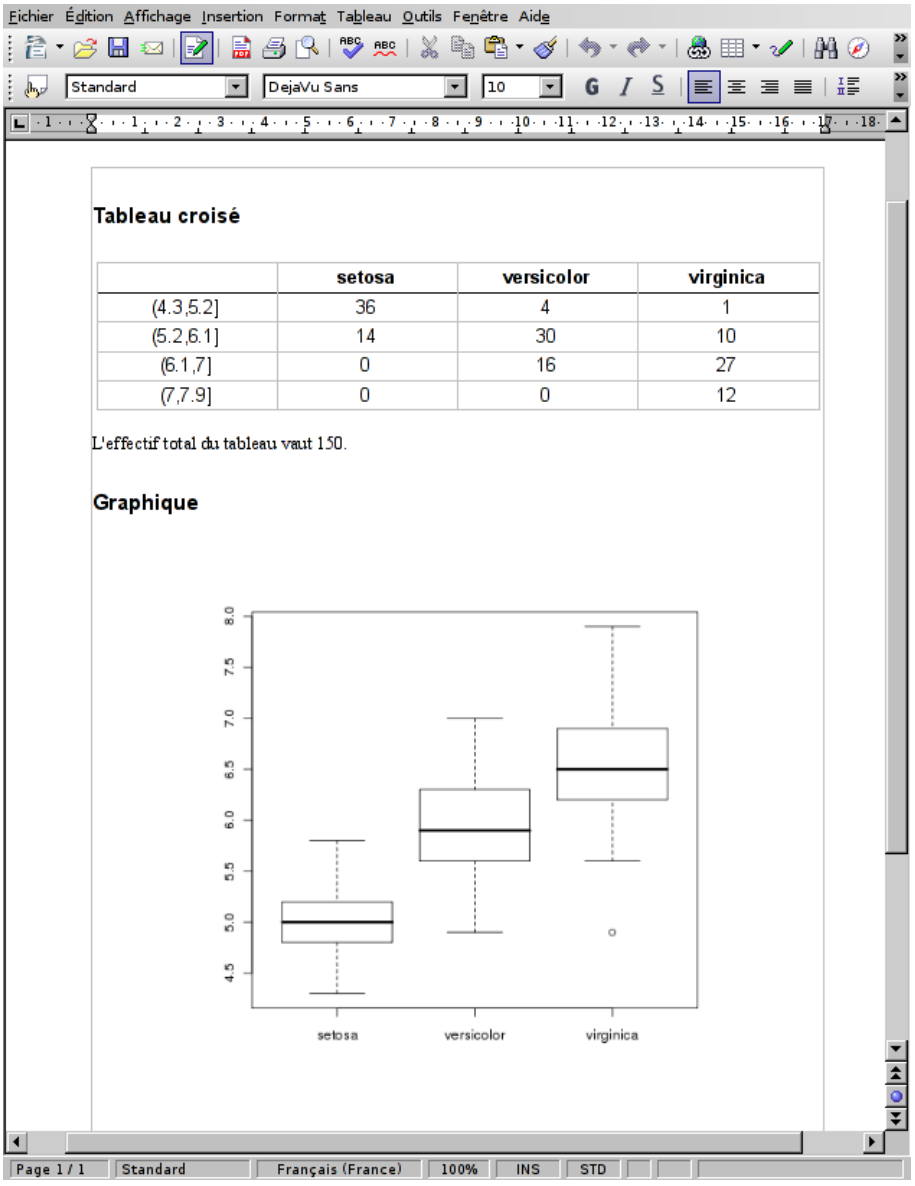


FIGURE 9.4 – Résultat de l'exemple de la figure 9.3

results indique le type de résultat renvoyé par le bloc. Si l'option vaut `verbatim` (par défaut), le résultat de la commande est affiché tel quel (par défaut en rouge). Si elle vaut `xml`, le résultat attendu est un objet OpenOffice : c'est l'option qu'on utilisera lorsqu'on fait appel à la fonction `genere.tableau`. Si l'option vaut `hide`, le résultat est masqué.

fig si `TRUE`, indique que le résultat du code est une image.

En résumé, si on souhaite utiliser un bloc pour charger des extensions sans que des traces apparaissent dans le document final, on utilise `«echo=FALSE,results=hide»=`. Si on veut afficher un tableau généré par `genere.tableau`, on utilise `«echo=FALSE,results=xml»=`. Si on souhaite insérer un graphique, on utilise `«echo=FALSE,fig=TRUE»=`. Si on souhaite afficher du code R et son résultat « tel quel », on utilise simplement `«»=`

rgrs

La fonction `genere.tableau` fait partie de l'extension rgrs . Elle transforme l'objet qu'on lui passe en paramètres dans un format mis en forme lisible par OpenOffice[5].

Elle permet de transformer les objets de type suivant :

- `table` à une ou deux dimensions (tri à plat ou tableau croisé obtenu avec les fonctions `table`, `lprop`, `cprop`...
- `data.frame`, y compris les tableaux de données ou le résultat de la commande `freq`.
- `vector`
- `matrix`

Pour générer le document résultat, on doit lancer une session R utilisant comme répertoire de travail celui où se trouve le document OpenOffice source, et exécuter les deux commandes suivantes :

```
R> library(odfWeave)
R> odfWeave("fichier_source.odt", "fichier_resultat.odt")
```

En pratique, on répartit en général son travail entre différents fichiers R qu'on appelle ensuite dans le document OpenOffice à l'aide de la fonction `source` histoire de limiter le code R dans le document au strict minimum. Par exemple, si on a regroupé le chargement des données et les recodages dans un fichier nommé `recodages.R`, on pourra utiliser le code suivant en début de document :

```
<<echo=FALSE,results=hide>>=
source("recodages.R")
@
```

Et se contenter dans la suite de générer les tableaux et graphiques souhaités.

Il existe un conflit entre les extensions R2HTML et odfWeave qui peut empêcher la seconde de fonctionner correctement si la première est chargée en mémoire. En cas de problème on pourra enlever l'extension R2HTML avec la commande `detach(package:R2HTML)`.

Enfin, différentes options sont disponibles pour personnaliser le résultat obtenu, et des commandes permettent de modifier le style d'affichage des tableaux et autres éléments générés. Pour plus d'informations, on se référera à la documentation de l'extension :

http://cran.r-project.org/web/packages/odfWeave/index.html

et notamment au document d'introduction en anglais :

http://cran.r-project.org/web/packages/odfWeave/vignettes/odfWeave.pdf

5. En fait la fonction `genere.tableau` ne fait rien par elle-même, elle se contente de simplifier l'appel à la fonction `odfTable` de l'extension odfWeave.

9.4 Génération automatique de rapports avec LaTeX

Des fonctionnalités similaires à celles offertes par l'extension odfWeave sont fournies pour LaTeX par des extensions comme Sweave ou knitr [6], permettant de générer dynamiquement des documents contenant du code R avec un rendu typographique de haute qualité.

Comme cette extension nécessite l'apprentissage de LaTeX, elle dépasse le cadre de ce document.

6. C'est cet outil qui a permis de générer le document que vous avez sous les yeux.

Partie 10

Où trouver de l'aide

10.1 Aide en ligne

R dispose d'une aide en ligne très complète, mais dont l'usage n'est pas forcément très simple. D'une part car elle est intégralement en anglais, d'autre part car son organisation prend un certain temps à être maîtrisée.

10.1.1 Aide sur une fonction

La fonction la plus utile est sans doute celle qui permet d'afficher la page d'aide liée à une ou plusieurs fonctions. Celle-ci permet de lister les arguments de la fonction, d'avoir des informations détaillées sur son fonctionnement, les résultats qu'elle retourne, etc.

Pour accéder à l'aide de la fonction `mean`, par exemple, il vous suffit de saisir directement :

```
R> help("mean")
```

Ou sa forme abrégée `?mean`.

Chaque page d'aide comprend plusieurs sections, en particulier :

Description donne un résumé en une phrase de ce que fait la fonction

Usage indique la ou les manières de l'utiliser

Arguments détaille tous les arguments possibles et leur signification

Value indique la forme du résultat renvoyé par la fonction

Details apporte des précisions sur le fonctionnement interne de la fonction

Note pour des remarques éventuelles

References pour des références bibliographiques ou des URL associées

See Also *très utile*, renvoit vers d'autres fonctions semblables ou liées, ce qui peut être très utile pour découvrir ou retrouver une fonction dont on a oublié le nom

Examples série d'exemples d'utilisation

Les exemples peuvent être directement exécutés en utilisant la fonction `example` :

```
R> example(mean)
```

```
meanR> x <- c(0:10, 50)

meanR> xm <- mean(x)

meanR> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

10.1.2 Naviguer dans l'aide

La fonction `help.start` permet d'afficher le contenu de l'aide en ligne au format HTML dans votre navigateur Web. Pour comprendre ce que cela signifie, saisissez simplement :

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

Ceci devrait lancer votre navigateur favori et afficher une page vous permettant alors de naviguer parmi les différentes extensions installées, d'afficher les pages d'aide des fonctions, de consulter les manuels, d'effectuer des recherches, etc.

À noter qu'à partir du moment où vous avez lancé `help.start()`, les pages d'aide demandées avec `help("lm")` ou `?plot` s'afficheront désormais dans votre navigateur.

Si vous souhaitez rechercher quelque chose dans le contenu de l'aide directement dans la console, vous pouvez utiliser la fonction `help.search`, qui renvoit une liste des pages d'aide contenant les termes recherchés. Par exemple :

```
R> help.search("logistic")
```

10.2 Ressources sur le Web

De nombreuses ressources existent en ligne, mais la plupart sont en anglais.

10.2.1 Moteur de recherche

Le fait que le logiciel s'appelle R ne facilite malheureusement pas les recherches sur le Web... La solution à ce problème a été trouvée grâce à la constitution d'un moteur de recherche *ad hoc* à partir de Google, nommé Rseek :

http://www.rseek.org/

Les requêtes saisies dans Rseek sont exécutées dans des corpus prédéfinis liés à R, notamment les documents et manuels, les listes de discussion ou le code source du programme.

Les requêtes devront cependant être formulées en anglais.

10.2.2 Ressources officielles

La documentation officielle de R est accessible en ligne depuis le site du projet :

http://www.r-project.org/

Les liens de l'entrée *Documentation* du menu de gauche vous permettent d'accéder à différentes ressources.

Les manuels sont des documents complets de présentation de certains aspects de R. Ils sont accessibles en ligne, ou téléchargeables au format PDF :

http://cran.r-project.org/manuals.html

On notera plus particulièrement *An introduction to R*, normalement destiné aux débutants, mais qui nécessite quand même un minimum d'aisance en informatique et en statistiques :

http://cran.r-project.org/doc/manuals/R-intro.html

R Data Import/Export explique notamment comment importer des données depuis d'autres logiciels :

http://cran.r-project.org/doc/manuals/R-data.html

Les FAQ regroupent des questions fréquemment posées et leurs réponses. À lire donc ou au moins à parcourir avant toute chose :

http://cran.r-project.org/faqs.html

La FAQ la plus utile est la FAQ généraliste sur R :

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

Mais il existe également une FAQ dédiée aux questions liées à Windows, et une autre à la plateforme Mac OS X.

Les manuels et les FAQ sont accessibles même si vous n'avez pas d'accès à Internet en utilisant la fonction `help.start()` décrite précédemment.

Le Wiki est un site dont les pages sont éditées par les utilisateurs, à la manière de *Wikipédia*. N'importe quel visiteur du site peut ainsi rajouter ou modifier des informations sur tel aspect de l'utilisation du logiciel :

http://wiki.r-project.org/

R-announce est la liste de diffusion électronique officielle du projet. Elle ne comporte qu'un nombre réduit de messages (quelques-uns par mois tout au plus) et diffuse les annonces concernant de nouvelles versions de R ou d'autres informations particulièrement importantes. On peut s'y abonner à l'adresse suivante :

https://stat.ethz.ch/mailman/listinfo/r-announce

R Journal est la « revue » officielle du projet R, qui a succédé début 2009 à la lettre de nouvelles *R News*. Elle paraît entre deux et cinq fois par an et contient des informations sur les nouvelles versions du logiciel, des articles présentant des extensions, des exemples d'analyse... Les parutions sont annoncées sur la liste de diffusion *R-announce*, et les numéros sont téléchargeables à l'adresse suivante :

http://journal.r-project.org/

Autres documents On trouvera de nombreux documents dans différentes langues, en général au format PDF, dans le répertoire suivant :

http://cran.r-project.org/doc/contrib/

Parmi ceux-ci, les cartes de référence peuvent être très utiles, ce sont des aides-mémoire recensant les fonctions les plus courantes :

http://cran.r-project.org/doc/contrib/Short-refcard.pdf

On notera également un document d'introduction en anglais progressif et s'appuyant sur des méthodes statistiques relativement simples :

http://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf

Pour les utilisateurs déjà habitués à SAS ou SPSS, le livre *R for SAS and SPSS Users* et le document gratuit qui en est tiré peuvent être de bonnes ressources, tout comme le site Web *Quick-R* :

http://rforsasandspssusers.com/

http://www.statmethods.net/

10.2.3 Revue

La revue *Journal of Statistical Software* est une revue électronique anglophone, dont les articles sont en accès libre, et qui traite de l'utilisation de logiciels d'analyse de données dans un grand nombre de domaines. De nombreux articles (la majorité) sont consacrés à R et à la présentation d'extensions plus ou moins spécialisées.

Les articles qui y sont publiés prennent souvent la forme de tutoriels plus ou moins accessibles mais qui fournissent souvent une bonne introduction et une ressource riche en informations et en liens.

Adresse de la revue :

http://www.jstatsoft.org/

10.2.4 Ressources francophones

Il existe des ressources en français sur l'utilisation de R, mais peu sont réellement destinées aux débutants, elles nécessitent en général des bases à la fois en informatique et en statistique.

Le document le plus abordable et le plus complet est sans doute *R pour les débutants*, d'Emmanuel Paradis, accessible au format PDF :

http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf

La somme de documentation en français la plus importante liée à R est sans nulle doute celle mise à disposition par le *Pôle bioinformatique lyonnais*. Leur site propose des cours complets de statistique utilisant R :

http://pbil.univ-lyon1.fr/R/enseignement.html

La plupart des documents sont assez pointus niveau mathématique et plutôt orientés biostatistique, mais on trouvera des documents plus introductifs ici :

http://pbil.univ-lyon1.fr/R/html/cours1

Dans tous les cas la somme de travail et de connaissances mise à disposition librement est impressionnante...

Enfin, le site de Vincent Zoonekynd comprend de nombreuses notes prises au cours de sa découverte du logiciel. On notera cependant que l'auteur est normalien et docteur en mathématiques...

http://zoonek2.free.fr/UNIX/48_R_2004/all.html

10.3 Où poser des questions

La communauté des utilisateurs de R est très active et en général très contente de pouvoir répondre aux questions (nombreuses) des débutants et à celles (tout aussi nombreuses) des utilisateurs plus expérimentés.

Dans tous les cas, les règles de base à respecter avant de poser une question sont toujours les mêmes : avoir cherché soi-même la réponse auparavant, notamment dans les FAQ et dans l'aide en ligne, et poser sa question de la manière la plus claire possible, de préférence avec un exemple de code posant problème.

10.3.1 Liste `R-soc`

Une liste de discussion a été créée spécialement pour permettre aide et échanges autour de l'utilisation de R en sciences sociales. Elle est hébergée par le CRU et on peut s'y abonner à l'adresse suivante :

`https://listes.cru.fr/sympa/subscribe/r-soc`

Grâce aux services offerts par le site `gmane.org`, la liste est également disponible sous d'autres formes (forum Web, blog, `NNTP`, fils RSS) permettant de lire et de poster sans avoir à s'inscrire et à recevoir les messages sous forme de courrier électronique.

Pour plus d'informations :

`http://dir.gmane.org/gmane.comp.lang.r.user.french`

10.3.2 StackOverflow

Le site *StackOverflow* (qui fait partie de la famille des sites *StackExchange*) comprend une section (anglophone) dédiée à R qui permet de poser des questions et en général d'obtenir des réponses assez rapidement :

`http://stackoverflow.com/questions/tagged/r`

La première chose à faire, évidemment, est de vérifier que sa question n'a pas déjà été posée.

10.3.3 Forum Web en français

Le Cirad a mis en ligne un forum dédié aux utilisateurs de R, très actif :

`http://forums.cirad.fr/logiciel-R/index.php`

Les questions diverses et variées peuvent être posées dans la rubrique *Questions en cours* :

`http://forums.cirad.fr/logiciel-R/viewforum.php?f=3`

Il est tout de même conseillé de faire une recherche rapide sur le forum avant de poser une question, pour voir si la réponse ne s'y trouverait pas déjà.

10.3.4 Canaux IRC (chat)

L'IRC, ou *Internet Relay Chat* est le vénérable ancêtre toujours très actif des messageries instantanées actuelles. Un canal (en anglais) est notamment dédié aux échanges autour de R (`#R`).

Si vous avez déjà l'habitude d'utiliser IRC, il vous suffit de pointer votre client préféré sur Freenode (`irc.freenode.net`) puis de rejoindre l'un des canaux en question.

Sinon, le plus simple est certainement d'utiliser l'interface Web de Mibbit, accessible à l'adresse :

`http://www.mibbit.com/`

Dans le champ *Connect to IRC*, sélectionnez *Freenode.net*, puis saisissez un pseudonyme dans le champ *Nick* et `#R` dans le champ *Channel*. Vous pourrez alors discuter directement avec les personnes présentes.

Le canal `#R` est normalement peuplé de personnes qui seront très heureuses de répondre à toutes les questions, et en général l'ambiance y est très bonne. Une fois votre question posée, n'hésitez pas à être patient et à attendre quelques minutes, voire quelques heures, le temps qu'un des habitués vienne y faire un tour.

10.3.5 Listes de discussion officielles

La liste de discussion d'entraide (par courrier électronique) officielle du logiciel R s'appelle `R-help`. On peut s'y abonner à l'adresse suivante, mais il s'agit d'une liste avec de nombreux messages :

`https://stat.ethz.ch/mailman/listinfo/r-help`

Pour une consultation ou un envoi ponctuels, le mieux est sans doute d'utiliser les interfaces Web fournies par gmane :

`http://blog.gmane.org/gmane.comp.lang.r.general`

`R-help` est une liste avec de nombreux messages, suivie par des spécialistes de R, dont certains des développeurs principaux. Elle est cependant à réserver aux questions particulièrement techniques qui n'ont pas trouvé de réponses par d'autres biais.

Dans tous les cas, il est nécessaire avant de poster sur cette liste de bien avoir pris connaissance du *posting guide* correspondant :

`http://www.r-project.org/posting-guide.html`

Plusieurs autres listes plus spécialisées existent également, elles sont listées à l'adresse suivante :

`http://www.r-project.org/mail.html`

Annexe A

Installer R

A.1 Installation de R sous Windows

Nous ne couvrons ici que l'installation de R sous Windows. Rappelons qu'en tant que logiciel libre, R est librement et gratuitement installable par quiconque.

La première chose à faire est de télécharger la dernière version du logiciel. Pour cela il suffit de se rendre à l'adresse suivante :

http://cran.cict.fr/bin/windows/base/release.htm

Vous allez alors vous voir proposer le téléchargement d'un fichier nommé `R-2.X.X-win32.exe` (les `X` étant remplacés par les numéros de la dernière version disponible). Une fois ce fichier sauvegardé sur votre poste, exécutez-le et procédez à l'installation du logiciel : celle-ci s'effectue de manière tout à fait classique, c'est-à-dire en cliquant un certain nombre de fois [1] sur le bouton *Suivant*.

Une fois l'installation terminée, vous devriez avoir à la fois une magnifique icône R sur votre bureau ainsi qu'une non moins magnifique entrée R dans les programmes de votre menu *Démarrer*. Il ne vous reste donc plus qu'à lancer le logiciel pour voir à quoi il ressemble.

A.2 Installation de R sous Mac OS X

R fonctionne pour les versions de Mac OS X 10.2 ultérieures. Néanmoins l'installateur par défaut nécessite au minimum une version 10.4.4 (*Tiger*).

L'installation est très simple :

1. Se rendre à la page suivante : http://cran.r-project.org/bin/macosx/
2. Télécharger le fichier nommé `R-2.X.Y.dmg`
3. Double cliquer sur le fichier téléchargé. Une fenêtre devrait s'ouvrir, contenant le programme d'installation.
4. Il vous suffit alors de double cliquer sur le programme d'installation et de suivre les instructions.

A.3 Mise à jour de R sous Windows

La méthode conseillée pour mettre à jour R sur les plateformes Windows est la suivante [2] :

1. Voire un nombre de fois certain. Vous pouvez laisser les options par défaut à chaque étape de l'installation.
2. Méthode conseillée dans l'entrée correspondante de la FAQ de R pour Windows : http://cran.r-project.org/bin/windows/rw-FAQ.html#What_0027s-the-best-way-to-upgrade_003f

1. Désinstaller R. Pour cela on pourra utiliser l'entrée *Uninstall R* présente dans le groupe R du menu *Démarrer*.
2. Installer la nouvelle version comme décrit précédemment.
3. Se rendre dans le répertoire d'installation de R, en général `C:\Program Files\R`. Sélectionner le répertoire de l'ancienne installation de R et copier le contenu du dossier nommé `library` dans le dossier du même nom de la nouvelle installation. En clair, si vous mettez à jour de R 2.6.2 vers R 2.7.1, copiez tout le contenu du répertoire `C:\Program Files\R\R-2.6.2\library` dans `C:\Program Files\R\R-2.7.1\library`.
4. Lancez la nouvelle version de R et exécuter la commande `update.packages` pour mettre à jour les extensions.

A.4 Interfaces graphiques

L'interface par défaut sous Windows est celle présentée figure 2.1 page 8. Il en existe d'autres, plus ou moins sophistiquées, qui vont de la simple coloration syntaxique à des interfaces plus complètes se rapprochant de modèles du type SPSS. Une liste des projets en cours est disponible sur la page suivante :

http://www.sciviews.org/_rgui/ (en anglais)

Le projet RStudio tend à s'imposer comme l'environnement de développement de référence pour R, d'autant qu'il a l'avantage d'être libre, gratuit et multiplateforme. Son installation est décrite section A.5 de la présente page.

Au final, ce document se basant toujours sur une utilisation de R basée sur la saisie de commandes textuelles, l'interface choisie importe peu. Celles-ci ne diffèrent que par le niveau de confort ou d'efficacité supplémentaires qu'elles apportent.

A.5 RStudio

RStudio est un environnement de développement intégré libre, gratuit, et qui fonctionne sour Windows, Mac OS X et Linux. Il fournit un éditeur de script avec coloration syntaxique, des fonctionnalités pratiques d'édition et d'exécution du code, un affichage simultané du code, de la console R, des fichiers, graphiques et pages d'aide, une gestion des extensions, une intégration avec des systèmes de contrôle de versions comme git, etc.

Il est en développement actif et de nouvelles fonctionnalités sont ajoutées régulièrement. Son seul défaut est d'avoir une interface uniquement anglophone.

Pour avoir un aperçu de l'interface de RStudio, on pourra se référer à la page *Screenshots* du site du projet :

http://www.rstudio.org/screenshots/

L'installation de RStudio est très simple, il suffit de se rendre sur la page de téléchargement et de sélectionner le fichier correspondant à son système d'exploitation :

http://www.rstudio.org/download/desktop

L'installation s'effectue ensuite de manière tout à fait classique.

Annexe B

Extensions

B.1 Présentation

L'installation par défaut du logiciel R contient le cœur du programme ainsi qu'un ensemble de fonctions de base fournissant un grand nombre d'outils de traitement de données et d'analyse statistiques.

R étant un logiciel libre, il bénéficie d'une forte communauté d'utilisateurs qui peuvent librement contribuer au développement du logiciel en lui ajoutant des fonctionnalités supplémentaires. Ces contributions prennent la forme d'extensions (*packages*) pouvant être installées par l'utilisateur et fournissant alors diverses fonctions supplémentaires.

Il existe un très grand nombre d'extensions (environ 1500 à ce jour), qui sont diffusées par un réseau baptisé CRAN (*Comprehensive R Archive Network*).

La liste de toutes les extensions disponibles sur le CRAN est disponible ici :

http://cran.r-project.org/web/packages/

Pour faciliter un peu le repérage des extensions, il existe un ensemble de regroupements thématiques (économétrie, finance, génétique, données spatiales...) baptisés *Task views* :

http://cran.r-project.org/web/views/

On y trouve notamment une *Task view* dédiée aux sciences sociales, listant de nombreuses extensions potentiellement utiles pour les analyses statistiques dans ce champ disciplinaire :

http://cran.r-project.org/web/views/SocialSciences.html

B.2 Installation des extensions

Les interfaces graphiques sous Windows ou Mac OS X permettent la gestion des extensions par le biais de boîtes de dialogues (entrées du menu *Packages* sous Windows par exemple). Nous nous contenterons ici de décrire cette gestion *via* la console.

On notera cependant que l'installation et la mise à jour des extensions nécessite d'être connecté à l'Internet.

L'installation d'une extension se fait par la fonction `install.packages`, à qui on fournit le nom de l'extension. Ici on souhaite installer l'extension ade4 :

```
R> install.packages("ade4", dep = TRUE)
```

L'option `dep=TRUE` indique à R de télécharger et d'installer également toutes les extensions dont l'extension choisie dépend pour son fonctionnement.

En général R va alors vous demander de choisir un *miroir* depuis lequel récupérer les données nécessaires. Choisissez de préférence un miroir le plus proche possible de l'endroit où vous vous trouvez [1].

Une fois l'extension installée, elle peut être appelée depuis la console ou un fichier script avec la commande :

```
R> library(ade4)
```

À partir de là, on peut utiliser les fonctions de l'extension, consulter leur page d'aide en ligne, accéder aux jeux de données qu'elle contient, etc.

Pour mettre à jour l'ensemble des extensions installées, une seule commande suffit :

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

Si on souhaite désinstaller une extension précédemment installée, on peut utiliser la fonction `remove.packages` :

```
R> remove.packages("ade4")
```

Il est important de bien comprendre la différence entre `install.packages` et `library`. La première va chercher les extensions sur l'Internet et les installe en local sur le disque dur de l'ordinateur. On n'a besoin d'effectuer cette opération qu'une seule fois. La seconde lit les informations de l'extension sur le disque dur et les met à disposition de R. On a besoin de l'exécuter à chaque début de session ou de script.

B.3 L'extension rgrs

rgrs est une extension pour R comprenant quelques fonctions potentiellement utiles pour l'utilisation du logiciel en sciences sociales. Pour l'instant elle comporte essentiellement des fonctions pour les tableaux croisés, l'export de résultats et pour le travail avec des fichiers issus de Modalisa [2].

B.3.1 Installation

L'installation nécessite d'avoir une connexion active à Internet. Depuis la version 0.2-6, l'extension est hébergée sur le CRAN (*Comprehensive R Archive Network*), le réseau officiel de diffusion des extensions de R. Elle est donc installable de manière très simple, comme n'importe quelle autre extension, par un simple :

1. Ayant déjà rencontré des soucis avec le miroir lyonnais, j'ai tendance à utiliser celui de Toulouse.
2. À noter que les fonctions en question ne sont en général que des interfaces facilitant l'utilisation de fonctions déjà existantes.

```
R> install.packages("rgrs", dep = TRUE)
```

L'extension s'utilise alors de manière classique grâce à l'instruction `library` en début de session ou de fichier R :

```
R> library(rgrs)
```

À noter que l'extension n'est disponible que pour les versions les plus récentes de R. Ainsi, depuis la sortie de la version 2.8, elle n'est plus installable de la manière décrite précédemment pour les versions 2.7 de R. Il est alors conseillé de mettre son installation de R à jour.

B.3.2 Fonctions et utilisation

Pour plus de détails sur la liste des fonctions de l'extension et son utilisation, on pourra se reporter aux pages Web suivantes :

http://alea.fr.eu.org/pages/rgrs

Un document PDF regroupant les pages d'aide en ligne de l'extension est notamment disponible :

http://cran.r-project.org/web/packages/rgrs/rgrs.pdf

Ainsi qu'une page décrivant plus particulièrement l'utilisation des fonctions facilitant l'importation et le traitement de données issues de Modalisa :

http://alea.fr.eu.org/pages/R-et-Modalisa

B.3.3 Le jeu de données `hdv2003`

L'extension rgrs contient plusieurs jeux de données (*dataset*) destinés à l'apprentissage de R.

`hdv2003` est un extrait comportant 2000 individus et 20 variables provenant de l'enquête *Histoire de Vie* réalisée par l'INSEE en 2003.

L'extrait est tiré du fichier détail mis à disposition librement (ainsi que de nombreux autres) par l'INSEE à l'adresse suivante :

http://www.insee.fr/fr/themes/detail.asp?ref_id=fd-HDV03

Les variables retenues ont été parfois partiellement recodées. La liste des variables est la suivante :

Variable	Description
`id`	Identifiant (numéro de ligne)
`poids`	Variable de pondération [3]
`age`	Âge
`sexe`	Sexe
`nivetud`	Niveau d'études atteint
`occup`	Occupation actuelle
`qualif`	Qualification de l'emploi actuel
`freres.soeurs`	Nombre total de frères, sœurs, demi-frères et demi-sœurs
`clso`	Sentiment d'appartenance à une classe sociale
`relig`	Pratique et croyance religieuse
`trav.imp`	Importance accordée au travail
`trav.satisf`	Satisfaction ou insatisfaction au travail
`hard.rock`	Ecoute du Hard rock ou assimilés
`lecture.bd`	Lecture de bandes dessinées
`peche.chasse`	Pêche ou chasse pour le plaisir au cours des 12 derniers mois
`cuisine`	Cuisine pour le plaisir au cours des 12 derniers mois
`bricol`	Bricolage ou mécanique pour le plaisir au cours des 12 derniers mois
`cinema`	Cinéma au cours des 12 derniers mois
`sport`	Sport ou activité physique pour le plaisir au cours des 12 derniers mois
`heures.tv`	Nombre moyen d'heures passées àregarder la télévision par jour

B.3.4 Le jeu de données `rp99`

`rp99` est issu du recensement de la population de 1999 de l'INSEE. Il comporte une petite partie des résultats pour l'ensemble des communes du Rhône, soit 301 lignes et 21 colonnes

La liste des variables est la suivante :

Variable	Description
`nom`	nom de la commune
`code`	Code de la commune
`pop.act`	Population active
`pop.tot`	Population totale
`pop15`	Population des 15 ans et plus
`nb.rp`	Nombre de résidences principales
`agric`	Part des agriculteurs dans la population active
`artis`	Part des artisans, commerçants et chefs d'entreprises
`cadres`	Part des cadres
`interm`	Part des professions intermédiaires
`empl`	Part des employés
`ouvr`	Part des ouvriers
`retr`	Part des retraités
`tx.chom`	Part des chômeurs
`etud`	Part des étudiants
`dipl.sup`	Part des diplômés du supérieur
`dipl.aucun`	Part des personnes sans diplôme
`proprio`	Part des propriétaires parmi les résidences principales
`hlm`	Part des logements HLM parmi les résidences principales
`locataire`	Part des locataires parmi les résidences principales
`maison`	Part des maisons parmi les résidences principales

3. Comme il s'agit d'un extrait du fichier, cette variable de pondération n'a en toute rigueur aucune valeur statistique. Elle a été tout de même incluse à des fins « pédagogiques ».

Annexe C

Solutions des exercices

Exercice 2.1, page 15

```
R> c(12, 13, 14, 15, 16)

[1] 12 13 14 15 16
```

Exercice 2.2, page 15

```
R> c(1, 2, 3, 4)

[1] 1 2 3 4

R> 1:4

[1] 1 2 3 4

R> c(1, 2, 3, 4, 8, 9, 10, 11)

[1]  1  2  3  4  8  9 10 11

R> c(1:4, 8:11)

[1]  1  2  3  4  8  9 10 11

R> c(2, 4, 6, 8)

[1] 2 4 6 8

R> 1:4 * 2

[1] 2 4 6 8
```

Exercice 2.3, page 16

```
R> chef <- c(1200, 1180, 1750, 2100)
R> conjoint <- c(1450, 1870, 1690, 0)
R> nb.personnes <- c(4, 2, 3, 2)
R> (chef + conjoint)/nb.personnes

[1]  662.5 1525.0 1146.7 1050.0
```

Exercice 2.4, page 16

```
R> chef <- c(1200, 1180, 1750, 2100)
R> min(chef)

[1] 1180

R> max(chef)

[1] 2100

R> chef.na <- c(1200, 1180, 1750, NA)
R> min(chef.na)

[1] NA

R> max(chef.na)

[1] NA

R> min(chef.na, na.rm = TRUE)

[1] 1180

R> max(chef.na, na.rm = TRUE)

[1] 1750
```

Exercice 3.5, page 32

```
R> library(rgrs)
R> data(hdv2003)
R> df <- hdv2003
R> str(df)

'data.frame': 2000 obs. of  20 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ age          : int  28 23 59 34 71 35 60 47 20 28 ...
 $ sexe         : Factor w/ 2 levels "Homme","Femme": 2 2 1 1 2 2 2 1 2 1 ...
```

```
$ nivetud      : Factor w/ 8 levels "N'a jamais fait d'etudes",..: 8 NA 3 8 3 6 3 6 NA 7 ...
$ poids        : num  2634 9738 3994 5732 4329 ...
$ occup        : Factor w/ 7 levels "Exerce une profession",..: 1 3 1 1 4 1 6 1 3 1 ...
$ qualif       : Factor w/ 7 levels "Ouvrier specialise",..: 6 NA 3 3 6 6 2 2 NA 7 ...
$ freres.soeurs: int  8 2 2 1 0 5 1 5 4 2 ...
$ clso         : Factor w/ 3 levels "Oui","Non","Ne sait pas": 1 1 2 2 1 2 1 2 1 2 ...
$ relig        : Factor w/ 6 levels "Pratiquant regulier",..: 4 4 4 3 1 4 3 4 3 2 ...
$ trav.imp     : Factor w/ 4 levels "Le plus important",..: 4 NA 2 3 NA 1 NA 4 NA 3 ...
$ trav.satisf  : Factor w/ 3 levels "Satisfaction",..: 2 NA 3 1 NA 3 NA 2 NA 1 ...
$ hard.rock    : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...
$ lecture.bd   : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 1 1 1 1 ...
$ peche.chasse : Factor w/ 2 levels "Non","Oui": 1 1 1 1 1 1 2 2 1 1 ...
$ cuisine      : Factor w/ 2 levels "Non","Oui": 2 1 1 2 1 1 2 2 1 1 ...
$ bricol       : Factor w/ 2 levels "Non","Oui": 1 1 1 2 1 1 1 2 1 1 ...
$ cinema       : Factor w/ 2 levels "Non","Oui": 1 2 1 2 1 2 1 1 2 2 ...
$ sport        : Factor w/ 2 levels "Non","Oui": 1 2 2 2 1 2 1 1 1 2 ...
$ heures.tv    : num  0 1 0 2 3 2 2.9 1 2 2 ...
```

Exercice 3.6, page 32

Utilisez la fonction suivante et corrigez manuellement les erreurs :

```
R> df.ok <- edit(df)
```

Attention à ne pas utiliser la fonction `fix` dans ce cas, celle-ci modifierait directement le contenu de `df`.

Puis utilisez la fonction `head` :

```
R> head(df.ok, 4)
```

Exercice 3.7, page 32

```
R> summary(df$age)
R> hist(df$age, breaks = 10, main = "Répartition des âges",
+     xlab = "Âge", ylab = "Effectif")
R> boxplot(df$age)
R> plot(table(df$age), main = "Répartition des âges",
+     xlab = "Âge", ylab = "Effectif")
```

Exercice 3.8, page 36

```
R> table(df$trav.imp)
R> summary(df$trav.imp)
R> freq(df$trav.imp)
R> dotchart(table(df$trav.imp))
```

Exercice 4.9, page 43

Utilisez la fonction `read.table` ou l'un de ses dérivés, en fonction du tableur utilisé et du format d'enregistrement.

Pour vérifier que l'importation s'est bien passée, on peut utiliser les fonctions `str`, `dim`, éventuellement `edit` et faire quelques tris à plat.

Exercice 4.10, page 43

Utilisez la fonction `read.dbf` de l'extension foreign.

Exercice 5.11, page 74

```
R> library(rgrs)
R> data(hdv2003)
R> d <- hdv2003
R> d <- renomme.variable(d, "clso", "classes.sociales")
R> d <- renomme.variable(d, "classes.sociales", "clso")
```

Exercice 5.12, page 74

```
R> d$clso <- factor(d$clso, levels = c("Non", "Ne sait pas",
+     "Oui"))
R> table(d$clso)


        Non Ne sait pas         Oui
       1037          27         936
```

Exercice 5.13, page 74

```
R> d$cinema[1:3]

[1] Non Oui Non
Levels: Non Oui

R> d$lecture.bd[12:30]

 [1] Non Non Non Non Non Non Non Non Non Non Non Non Non Non Non Non Non Non Non
Levels: Non Oui

R> d[c(5, 12), c(4, 8)]

                            nivetud freres.soeurs
5  Derniere annee d'etudes primaires             0
12                        2eme cycle             4

R> longueur <- length(d$age)
R> tail(d$age, 4)

[1] 46 24 24 66
```

Exercice 5.14, page 74

```
R> subset(d, lecture.bd == "Oui", select = c(age, sexe))
R> subset(d, occup != "Chômeur", select = -cinema)
R> subset(d, age >= 45 & hard.rock == "Oui", select = id)
R> subset(d, sexe == "Femme" & age >= 25 & age <= 40 & sport ==
+     "Non")
R> subset(d, sexe == "Homme" & freres.soeurs >= 2 & freres.soeurs <=
+     4 & (cuisine == "Oui" | bricol == "Oui"))
```

Exercice 5.15, page 74

```
R> d.bd.oui <- subset(d, lecture.bd == "Oui")
R> d.bd.non <- subset(d, lecture.bd == "Non")
R> mean(d.bd.oui$heures.tv)

[1] 1.764

R> mean(d.bd.non$heures.tv, na.rm = TRUE)

[1] 2.258

R> tapply(d$heures.tv, d$lecture.bd, mean, na.rm = TRUE)

  Non   Oui
2.258 1.764
```

Exercice 5.16, page 74

```
R> d$fs.char <- as.character(d$freres.soeurs)
R> d$fs.fac <- factor(d$fs.char)
R> d$fs.num <- as.numeric(as.character(d$fs.char))
R> table(d$fs.num == d$freres.soeurs)


TRUE
2000
```

Exercice 5.17, page 75

```
R> d$fs1 <- cut(d$freres.soeurs, 5)
R> table(d$fs1)


(-0.022,4.39]    (4.39,8.8]    (8.8,13.2]   (13.2,17.6]     (17.6,22]
         1495           396            97             9             3
```

```
R> d$fs2 <- cut(d$freres.soeurs, breaks = c(0, 2, 4, 19),
+     include.lowest = TRUE, labels = c("de 0 à 2", "de 2 à 4",
+         "plus de 4"))
R> table(d$fs2)


 de 0 à 2  de 2 à 4 plus de 4
     1001       494       504

R> d$fs3 <- quant.cut(d$freres.soeurs, 3)
R> table(d$fs3)


 [0,2)  [2,4) [4,22]
   574    711    715
```

Exercice 5.18, page 75

```
R> d$trav.imp2cl[d$trav.imp == "Le plus important" | d$trav.imp ==
+     "Aussi important que le reste"] <- "Le plus ou aussi important"
R> d$trav.imp2cl[d$trav.imp == "Moins important que le reste" |
+     d$trav.imp == "Peu important"] <- "moins ou peu important"
R> table(d$trav.imp)


           Le plus important Aussi important que le reste
                          29                         259
Moins important que le reste               Peu important
                         708                          52

R> table(d$trav.imp2cl)


Le plus ou aussi important     moins ou peu important
                       288                        760

R> table(d$trav.imp, d$trav.imp2cl)


                             Le plus ou aussi important moins ou peu important
  Le plus important                                  29                      0
  Aussi important que le reste                      259                      0
  Moins important que le reste                        0                    708
  Peu important                                       0                     52
```

```
R> d$relig.4cl <- as.character(d$relig)
R> d$relig.4cl[d$relig == "Pratiquant regulier" | d$relig ==
+     "Pratiquant occasionnel"] <- "Pratiquant"
R> d$relig.4cl[d$relig == "NSP ou NVPR"] <- NA
R> table(d$relig.4cl, d$relig, exclude = NULL)
```

```
                              Pratiquant regulier Pratiquant occasionnel
  Appartenance sans pratique                    0                      0
  Ni croyance ni appartenance                   0                      0
  Pratiquant                                  266                    442
  Rejet                                         0                      0
  <NA>                                          0                      0

                              Appartenance sans pratique Ni croyance ni appartenance
  Appartenance sans pratique                         760                           0
  Ni croyance ni appartenance                          0                         399
  Pratiquant                                           0                           0
  Rejet                                                0                           0
  <NA>                                                 0                           0

                              Rejet NSP ou NVPR <NA>
  Appartenance sans pratique      0           0    0
  Ni croyance ni appartenance     0           0    0
  Pratiquant                      0           0    0
  Rejet                          93           0    0
  <NA>                            0          40    0
```

Exercice 5.19, page 75

Attention, l'ordre des opérations a toute son importance !

```
R> d$var <- "Autre"
R> d$var[d$sexe == "Femme" & d$bricol == "Oui"] <- "Femme faisant du bricolage"
R> d$var[d$sexe == "Homme" & d$age > 30] <- "Homme de plus de 30 ans"
R> d$var[d$sexe == "Homme" & d$age > 40 & d$lecture.bd ==
+     "Oui"] <- "Homme de plus de 40 ans lecteur de BD"
R> table(d$var)


                              Autre            Femme faisant du bricolage
                                925                                   338
             Homme de plus de 30 ans Homme de plus de 40 ans lecteur de BD
                                728                                     9

R> table(d$var, d$sexe)


                                       Homme Femme
  Autre                                  162   763
  Femme faisant du bricolage               0   338
  Homme de plus de 30 ans                728     0
  Homme de plus de 40 ans lecteur de BD    9     0

R> table(d$var, d$bricol)


                                       Non Oui
```

```
  Autre                                847  78
  Femme faisant du bricolage             0 338
  Homme de plus de 30 ans              298 430
  Homme de plus de 40 ans lecteur de BD  2   7

R> table(d$var, d$lecture.bd)


                                     Non Oui
  Autre                              905  20
  Femme faisant du bricolage         324  14
  Homme de plus de 30 ans            724   4
  Homme de plus de 40 ans lecteur de BD  0   9

R> table(d$var, d$age > 30)


                                     FALSE TRUE
  Autre                                283  642
  Femme faisant du bricolage            68  270
  Homme de plus de 30 ans                0  728
  Homme de plus de 40 ans lecteur de BD  0    9

R> table(d$var, d$age > 40)


                                     FALSE TRUE
  Autre                                417  508
  Femme faisant du bricolage           152  186
  Homme de plus de 30 ans              163  565
  Homme de plus de 40 ans lecteur de BD  0    9
```

Exercice 5.20, page 75

```
R> d.ord <- d[order(d$freres.soeurs), ]
R> d.ord <- d[order(d$heures.tv, decreasing = TRUE), c("sexe",
+     "heures.tv")]
R> head(d.ord, 10)

      sexe heures.tv
288  Femme        12
391  Femme        12
1324 Homme        11
1761 Femme        11
100  Femme        10
236  Femme        10
421  Homme        10
426  Femme        10
841  Femme        10
1075 Homme        10
```

Table des figures

Index des fonctions