Ewen Gallic [1]
http://egallic.fr
2015

Logiciel R et programmation

1. ewen.gallic[at]gmail.com

Avant-propos

Ces notes de cours ont été réalisées dans le cadre d'un enseignement d'introduction à R adressé à des étudiants du Master de Statistique et Économétrie de la Faculté des Sciences Économiques de l'Université de Rennes 1.

Objectifs

Cet ouvrage a pour but l'initiation au logiciel statistique et au langage informatique portant le même nom, R, afin d'être capable de s'en servir de manière efficace et autonome.
Le lecteur peut exécuter tous les exemples fournis (et est vivement encouragé à le faire).
Des exercices viennent clore certains chapitres. Les corrections sont disponibles en ligne à l'adresse suivante : http://egallic.fr/enseignement/.

À qui s'adressent ces notes ?

Dans un premier temps, cet ouvrage s'adresse aux débutants qui souhaientent apprendre les bases du langage et du logiciel R. Le lecteur initié peut également s'appuyer sur ces notes afin d'y retrouver des rappels sur les notions basiques.

Remerciements

Ce manuel est le produit de plusieurs années de pratique, qui ont débutées en 2011 grâce à Christophe Cariou, que je tiens à remercier pour m'avoir soufflé l'idée d'utiliser R pour réaliser des projets amusants sur lesquels nous avons travaillés. Un énorme merci à Arthur Charpentier pour la quantité de savoir et d'astuces fournis sur l'utilisation de R, et qui est à l'origine de nombreux exemples présents dans ces notes de cours. Merci aussi à François Briatte de m'avoir introduit à de nouveaux horizons dans le langage, et aussi à Gauthier Vermandel pour ses nombreux conseils. Merci également à Julien-Yacine Chaqra pour ses suggestions et relectures. Enfin, merci à toute la communauté R d'exister et de proposer ce partage de savoir librement et gratuitement.

Table des matières

1 Introduction **1**
- 1.1 Historique . . . 1
- 1.2 Présentation de l'espace de travail . . . 1
  - 1.2.1 La console et les fenêtres de script . . . 1
  - 1.2.2 Les variables . . . 2
    - 1.2.2.1 Assignation . . . 2
    - 1.2.2.2 Conventions de nommage . . . 3
    - 1.2.2.3 Les objets en mémoire . . . 3
    - 1.2.2.4 Modification, suppression . . . 4
  - 1.2.3 Les *packages* . . . 4
  - 1.2.4 L'aide . . . 5
- 1.3 Un environnement de développement : RStudio . . . 6

2 Données **7**
- 2.1 Types de données . . . 7
  - 2.1.1 Mode . . . 7
  - 2.1.2 Longueur . . . 9
  - 2.1.3 Données manquantes . . . 9
  - 2.1.4 L'objet vide . . . 9
- 2.2 Structure des données . . . 10
  - 2.2.1 Structures de base . . . 10
    - 2.2.1.1 Vecteurs . . . 10
    - 2.2.1.2 Facteurs . . . 11
    - 2.2.1.3 Dates . . . 12
    - 2.2.1.4 Matrices . . . 15
    - 2.2.1.5 Listes . . . 17
  - 2.2.2 Bases de données . . . 18
    - 2.2.2.1 Data frames . . . 18
    - 2.2.2.2 Data table . . . 19
- 2.3 Importation, exportation et création de données . . . 20
  - 2.3.1 Importation . . . 20
    - 2.3.1.1 La fonction `read.table` . . . 20
    - 2.3.1.2 La fonction `scan` . . . 22
    - 2.3.1.3 La fonction `read.fwf` . . . 23

2.3.1.4 Importation depuis Excel© . . . 24
2.3.1.5 Importation depuis d'autres formats . . . 25
2.3.2 Exportation . . . 25
2.3.3 Génération . . . 26
2.3.3.1 Séquences régulières . . . 26
2.3.3.2 Séquences pseudo-aléatoires . . . 28
2.4 Manipulation des données . . . 29
2.4.1 Opérateurs . . . 29
2.4.1.1 Opérateurs arithmétiques . . . 30
2.4.1.2 Opérateurs de comparaison . . . 30
2.4.1.3 Opérateurs logiques . . . 31
2.4.1.4 Attention au recyclage . . . 32
2.4.2 Accès aux valeurs, modifications . . . 33
2.4.2.1 Accès par indices . . . 33
2.4.2.2 Accès par noms . . . 38
2.4.3 Chaînes de caractères . . . 41
2.4.3.1 Concaténation . . . 41
2.4.3.2 Conversion en majuscules ou minuscules . . . 43
2.4.3.3 Compter le nombre de caractères d'une chaîne . . . 43
2.4.3.4 Extraction de sous-chaînes . . . 44
2.4.3.5 Recherche de chaînes de caractères . . . 46
2.4.3.6 Nettoyage, complétion . . . 48
2.4.4 Dates . . . 49
2.4.4.1 Extraction . . . 49
2.4.4.2 Opérations . . . 50
2.4.4.3 Intervalles de dates . . . 53
2.4.4.4 Séquence de dates . . . 56
2.4.4.5 Fuseaux horaires . . . 57
2.4.4.6 Paramètres locaux . . . 57
2.4.5 Calculs matriciels . . . 59
2.4.5.1 Addition, soustraction . . . 59
2.4.5.2 Multiplication, division . . . 60
2.4.5.3 Déterminant, trace . . . 62
2.4.6 Data frames . . . 62
2.4.6.1 Sélection . . . 62
2.4.6.2 Filtrage . . . 64
2.4.6.3 Retirer les valeurs dupliquées . . . 65
2.4.6.4 Modification des colonnes . . . 65
2.4.6.5 Tri . . . 70
2.4.6.6 Jointures . . . 72
2.4.6.7 Agrégation . . . 76
2.4.6.8 Stacking et unstacking . . . 79
2.4.7 Data tables . . . 83
2.4.7.1 Création, conversion . . . 83
2.4.7.2 Sélection . . . 84
2.4.7.3 Filtrage . . . 85
2.4.7.4 Retirer les valeurs dupliquées . . . 85
2.4.7.5 Clés . . . 85
2.4.7.6 Modification des colonnes . . . 88
2.4.7.7 Tri . . . 88
2.4.7.8 Copie de `data.table` . . . 89

2.4.7.9 Jointures . . . 90
2.4.7.10 Agrégation . . . 91
2.4.8 Quelques fonctions utiles . . . 93
2.5 Exercices . . . 96

3 Fonctions **102**
3.1 Définition . . . 102
3.2 La structure d'une fonction . . . 103
3.2.1 Le corps d'une fonction . . . 103
3.2.2 Les paramètres d'une fonction . . . 105
3.2.2.1 Appel sans noms . . . 105
3.2.2.2 Paramètres effectifs . . . 105
3.2.2.3 Appel avec des noms partiels . . . 106
3.2.2.4 Fonctions sans paramètres . . . 106
3.2.2.5 La paramètre spécial `...` . . . 107
3.2.3 Portée des fonctions . . . 107
3.3 Exercices . . . 109

4 Boucles et calculs vectoriels **112**
4.1 Les boucles . . . 112
4.1.1 Les boucles avec `while()` . . . 112
4.1.2 Les boucles avec `for()` . . . 113
4.1.3 Les conditions . . . 113
4.1.3.1 Les instructions `if ... else` . . . 113
4.1.3.2 La fonction `switch` . . . 114
4.1.4 L'instruction `repeat ...`, `break` . . . 115
4.1.5 L'instruction `next ...`, `break` . . . 115
4.1.6 Barre de progression . . . 116
4.2 La vectorisation . . . 116
4.2.1 Les fonctions du package `plyr` . . . 117
4.2.1.1 *Array* en *input* : `a*ply()` . . . 118
4.2.1.2 *Data frame* en *input* : `d*ply()` . . . 120
4.2.1.3 *List* en *input* : `l*ply()` . . . 122
4.2.1.4 Calcul parallèle . . . 123
4.2.2 Les fonctions de la famille `apply` du package `base` . . . 124
4.2.2.1 La fonction `lapply` . . . 124
4.2.2.2 La fonction `sapply` . . . 126
4.2.2.3 La fonction `vapply` . . . 128
4.2.2.4 La fonction `apply` . . . 128
4.2.2.5 La fonction `tapply` . . . 129
4.2.2.6 La fonction `mapply` . . . 130
4.2.3 La fonction `Vectorize` . . . 131
4.3 Exercices . . . 131

5 Graphiques **134**
5.1 Structure . . . 134
5.2 Des graphiques élaborés avec `ggplot()` . . . 135
5.2.1 Paramètres esthétiques . . . 136
5.2.2 Paramètres géométriques (fonctions `geom_*()`) . . . 139
5.2.2.1 Points . . . 140
5.2.2.2 Lignes . . . 141
5.2.2.3 Polygones . . . 142

5.2.2.4 Boxplot . . . 142
5.2.2.5 Gigue . . . 143
5.2.2.6 Courbe de tendance . . . 144
5.2.2.7 Histogramme . . . 145
5.2.2.8 Densité . . . 146
5.2.3 Paramètres statistiques (fonctions `stat_*()`) . . . 147
5.2.4 Échelles (fonctions `scale_*()`) . . . 150
5.2.5 Groupes . . . 156
5.2.6 Annotations . . . 157
5.2.6.1 Texte . . . 158
5.2.6.2 Lignes . . . 159
5.2.6.3 Rectangles . . . 161
5.2.7 Positions . . . 161
5.2.8 Facettes . . . 162
5.2.8.1 La fonction `facet_grid()` . . . 163
5.2.8.2 La fonction `facet_wrap()` . . . 165
5.2.8.3 Échelles et facettes . . . 166
5.2.9 Coordonnées . . . 167
5.2.10 Titres, axes, légendes . . . 169
5.3 Enregistrement des graphiques . . . 172
5.4 Cartes . . . 173
5.4.1 Récupérer des cartes toutes faites . . . 173
5.4.1.1 *Package* `rworldmap` . . . 173
5.4.1.2 *Package* `maps` . . . 174
5.4.1.3 Fichier `shapefile` . . . 177
5.4.2 Carte choroplèthe . . . 178
5.5 Graphiques en 3D . . . 179
5.6 Exercices . . . 181

6 Régressions linéaires avec `R` 186
6.1 Rappels . . . 186
6.2 Données de l'exemple . . . 187
6.3 Estimation des paramètres . . . 190
6.4 Lecture des sorties . . . 191
6.5 Extractions . . . 192
6.6 Variables catégorielles . . . 194
6.7 Tests de nullité des coefficients et intervalles de confiance . . . 195
6.8 Prévisions . . . 197
6.9 Exportation des résultats . . . 199
6.10 Exercices . . . 200

7 Expressions régulières 202
7.1 Les recherches basiques . . . 202
7.2 Les classes de caractères . . . 205
7.3 Les quantificateurs . . . 209
7.4 Les métacaractères . . . 210
7.5 Fonctions d'extraction . . . 210
7.6 Quelques fonctions du *package* `stringr` . . . 210
7.6.1 La fonction `str_detect()` . . . 211
7.6.2 La fonction `str_extract()` . . . 211
7.6.3 La fonction `str_extract_all()` . . . 211
7.6.4 La fonction `str_match()` . . . 212

7.6.5 La fonction `str_match_all()` . . . 212
7.6.6 La fonction `str_locate()` . . . 213
7.6.7 La fonction `str_locate_all()` . . . 214
7.6.8 La fonction `str_replace()` . . . 215
7.6.9 La fonction `str_replace_all()` . . . 215
7.6.10 La fonction `str_split()` . . . 215
7.6.11 La fonction `str_string_fixed()` . . . 216

Références **217**

Index des expressions `R` **219**

1

Introduction

Ce document est construit principalement à l'aide des références suivantes : Lafaye de Micheaux et al. (2011), Farnsworth (2008), Charpentier (2014), Zuur et al. (2009), Paradis (2002) et Goulet (2014).

1.1 Historique

Dans le milieu des années 1970, une équipe de chercheurs de AT&T Bell Laboratories, composée de John Chambers, Douglas Bates, Rick Becker, Bill Cleveland, Trevor Hastie, Daryl Pregibon et Allan Wilks, développe un langage de programmation appelé `S` (la lettre `S` faisant référence à *statistics*). Il s'agit d'un langage permettant de manipuler les données et d'effectuer des analyses statistiques et graphiques. Dans le milieu des années 1990, Ross Ihaka et Robert Gentleman créént le `R` au département de Statistiques de l'Université d'Auckland. Ce langage et logiciel, inspiré du `S` et de `Scheme` est distribué sous les termes de la GNU *General Public Licence*. La *`R` Development Core Team* se charge de distribuer et de développer le logiciel `R`. De nombreux contributeurs à travers le monde participent au développement de `R`.

1.2 Présentation de l'espace de travail

1.2.1 La console et les fenêtres de script

`R` est un langage interprété, c'est-à-dire qu'il nécessite un interprète pour exécuter les commandes, et n'a pas de phase de compilation. Il existe plusieurs façons de travailler avec `R` : soit directement dans la console (qui est un interprète), soit dans une fenêtre de script (Fig 1.1). La seconde méthode est fortement conseillée, puisqu'elle permet de sauvegarder le code et d'y accéder facilement pour une utilisation utlérieure. De plus, il existe souvent des raccourcis claviers, en fonction de l'éditeur utilisé, qui permettent de soumettre le code de la fenêtre de script à l'interprète. Les fichiers de code que l'on sauvegardent possèdent l'extension "`.R`".
Dans la console, à la fin de l'affichage d'ouverture de session `R`, se trouve le caractère `>` (*prompt*), invitant l'utilisateur à inscrire une commande. Les expressions sont évaluées une fois qu'elle sont soumises (à l'aide de la touche `ENTREE`) et le résultat est donné, lorsqu'il n'y a pas d'erreur dans le code, juste en dessous :

```
> 2+1

## [1] 3
```

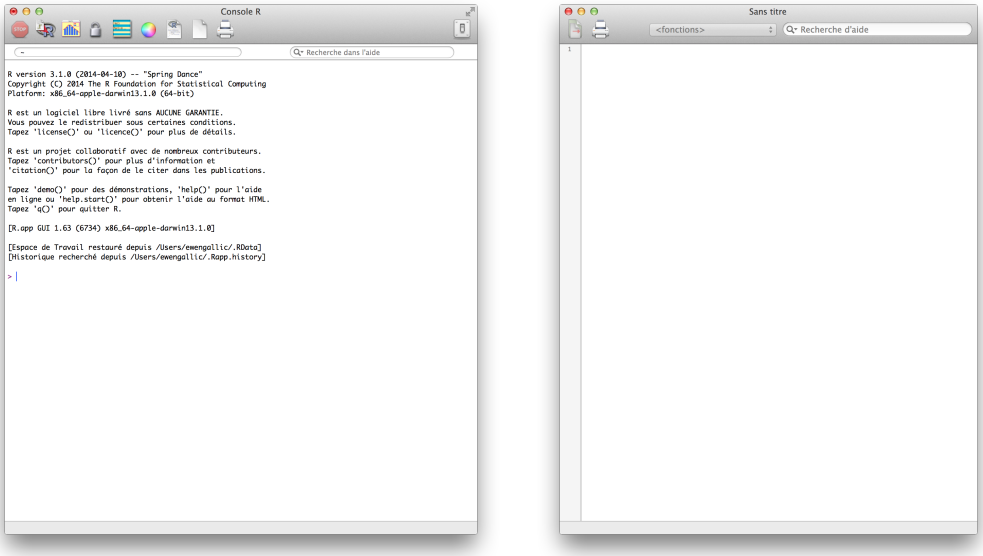


FIGURE 1.1 – Console et fenêtre de script sous Mac OS X

Par la suite, le symbole d'invite de commande ne sera plus affichée dans les cadres de code, afin de permettre au lecteur de reproduire facilement les résultats obtenus en copiant et collant les instructions. Par ailleurs, dans ce document, contrairement à ce que l'on peut observer dans la console `R`, le résultat de l'évalutation de l'expression "`2+1`" est précédé de deux symboles dièses. Le symbole `#` indique à `R` de ne pas evaluer le code qui suit, jusqu'à la fin de la ligne. Il permet donc de placer des lignes en commentaires.

```
# Des commentaires
2+1 # +1

## [1] 3
```

Lorsque l'on travaille dans une fenêtre de script, la combinaison de touches pour soumettre le code dépend du système d'exploitation et de l'éditeur de texte utilisé (`CTRL+R` sous Windows, `CMD+ENTER` sous Mac OS).

1.2.2 Les variables

1.2.2.1 Assignation

De base, le résultat d'une évaluation n'est pas enregistré, et est de fait perdu une fois affiché. Cependant, dans la plupart des cas, il est utile de conserver la sortie dans une variable. Il suffit alors d'attribuer un nom et un contenu à un objet. La flèche d'affectation `<-` permet cela. L'affichage du contenu de l'objet `x` se fait en inscrivant son nom et en l'évaluant.

```
x <- 2+1
x

## [1] 3
```

Remarque 1.2.1

Il est possible d'afficher le contenu de l'objet en entourant l'expression de son affectation à une valeur par des parenthèses :

```
(x <- 2+1)

## [1] 3
```

Remarque 1.2.2

Comme de nombreux langages de programmation utilisent le symbole `=` pour assigner une valeur à un objet, cette pratique est permise dans beaucoup de cas en `R`, mais il arrive qu'elle ne produise pas l'effet escompté dans d'autres cas de figures.

1.2.2.2 Conventions de nommage

Le nom d'une variable doit être composé de caractères alphanumériques ou du point ou du trait de soulignement uniquement. Il ne doit pas commencer par un chiffre ou contenir d'espace, sauf s'il est entouré par des guillemets. Il est important de noter que ces noms sont sensibles à la casse, c'est à dire qu'une distinction entre les majuscules et les minuscules a lieu. Il existe plusieurs pratiques pour nommer des variables, et il n'est pas rare de voir des combinaisons de ces pratiques. Un billet de Robin Lovelace explique les différences entre les méthodes et propose d'en adopter une seule (Lovelace, 2014) :

- tout en minuscule : nomvariable ;
- séparation par un point : nom.variable ;
- séparation par un trait de soulignement : nom_variable ;
- lowerCamelCase, avec la première lettre en minuscule : nomVariable ;
- PascalCase, avec la première lettre en majuscule : NomVariable.

Robin Lovelace propose d'avoir recours aux traits de soulignement, pour la facilité de lecture qu'ils permettent, et par leur emploi par quelques grands noms de la communauté `R` (comme Hadley Wickham [1] ou Yihui Xie [2]).

1.2.2.3 Les objets en mémoire

Pour lister les éléments de la session `R` stockés en mémoire de l'ordinateur, il faut appeler la fonction `ls()` :

```
ls()

## [1] "x"
```

1. http://had.co.nz/
2. http://yihui.name/

1.2.2.4 Modification, suppression

Il est important de noter que l'assignation du contenu d'un objet à un autre objet permet d'obtenir deux objets distincts. Ainsi, si un des deux objets est modifié, cela n'affecte pas l'autre.

```
premier_objet <- 2+1
second_objet <- premier_objet # On assigne la valeur de premier_objet
# à second_objet
premier_objet <- 2+2 # On change la valeur de premier_objet
# Puis on affiche le contenu des deux objets :
premier_objet # Le résultat doit être 4

## [1] 4

second_objet # Le résultat doit être 3

## [1] 3
```

Pour supprimer un objet, il faut utiliser la fonction `rm()` :

```
premier_objet # L'objet existe

## [1] 4

rm(premier_objet) # On le supprime
premier_objet # Son affichage provoque un message d'erreur

## Error in eval(expr, envir, enclos): objet 'premier_objet' introuvable
```

1.2.3 Les *packages*

Les fonctions de base de `R` sont contenues dans un *package* nommé `base`. Celui-ci est chargé automatiquement à l'ouverture de `R`, en même temps qu'une poignée d'autres. Les *packages* sont des jeux de fonctions, accompagnés de fichiers d'aides, parfois de jeux de données, qui sont mis à disposition des utilisateurs. La liste des *packages* chargés en mémoire par défaut s'obtient par l'instruction suivante :

```
getOption("defaultPackages")

## [1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"
```

Les fonctions qui ne sont pas dans les *packages* en mémoire nécessitent d'être chargées. Mais avant de pouvoir le faire, encore faut-il que le *package* soit installé. Si ce n'est pas le cas, il suffit de faire appel à l'instruction `install.packages()`, puis de choisir, selon l'éditeur utilisé, un site miroir. Par exemple, pour installer le *package* `ggplot2`, qui permet de réaliser des graphiques plus ou moins sophistiqués, l'instruction est la suivante :

```
install.packages("ggplot2")
```

Pour charger le *package* en mémoire et ainsi avoir accès aux fonctions que celui-ci contient, l'instruction est `library()`.

```
library("ggplot2")
```

L'instruction `require()`, qui tente de charger le *package* et retourne une valeur logique indiquant le succès ou l'échec de l'opération, est utilisée par certains, mais Yihui Xie indique sur un billet publié sur le site R-bloggers (Xie, 2014) que lors d'un échec du chargement du *package* en début de fichier, l'emploi des fonctions contenues dans ce *package* échouera, tout simplement.

1.2.4 L'aide

Pour conclure cette introduction, il semble important de mentionner la présence de l'aide en ligne. L'accès aux fichiers d'aide pour une fonction dont on connaît le nom, par exemple la fonction logarithme, peut se faire de plusieurs manières :

```
?log # Première manière
help(log) # Seconde manière
help("log") # Troisième manière
```

Pour trouver toutes les fonctions dont le nom contient une chaîne de cractètres donnée, on peut utiliser la fonction `apropos()` :

```
apropos("log")
```

```
##  [1] ".__C__logical"         ".__C__logLik"
##  [3] ".__T__Logic:base"      "annotation_logticks"
##  [5] "as.data.frame.logical" "as.logical"
##  [7] "as.logical.factor"     "dlogis"
##  [9] "GeomLogticks"          "is.logical"
## [11] "log"                   "log10"
## [13] "log1p"                 "log2"
## [15] "logb"                  "Logic"
## [17] "logical"               "logLik"
## [19] "loglin"                "plogis"
## [21] "qlogis"                "rlogis"
## [23] "scale_x_log10"         "scale_y_log10"
## [25] "SSlogis"
```

Les fichiers sont souvent riches d'informations, très bien décrits, et proposent des exemples d'utilisation.

Par ailleurs, la communauté `R` est importante, et rares sont les questions sans réponses sur les *mailing-lists* ou les pages de *Stack Overflow*[3].

3. http://stackoverflow.com/questions/tagged/r

1.3 Un environnement de développement : RStudio

Bien qu'il soit possible d'utiliser `R` *via* le terminal, ou *via* l'éditeur installé par dfaut, il est également possible d'utiliser un environnement de développement, comme RStudio. La version *desktop*, gratuite et en *open source*, est téléchargeable à l'adresse suivante : https://www.rstudio.com/.
Pour utiliser RStudio, il est nécessaire d'avoir au préalable installé `R`.
La figure 1.2 montre l'interface graphique proposée par RStudio. Cette interface est composée de quatre panneaux : un éditeur, une console, une fenêtre historique et environnement, et une dernière avec divers onglets (fichiers, graphiques, *packages*, aide et un navigateur interne).

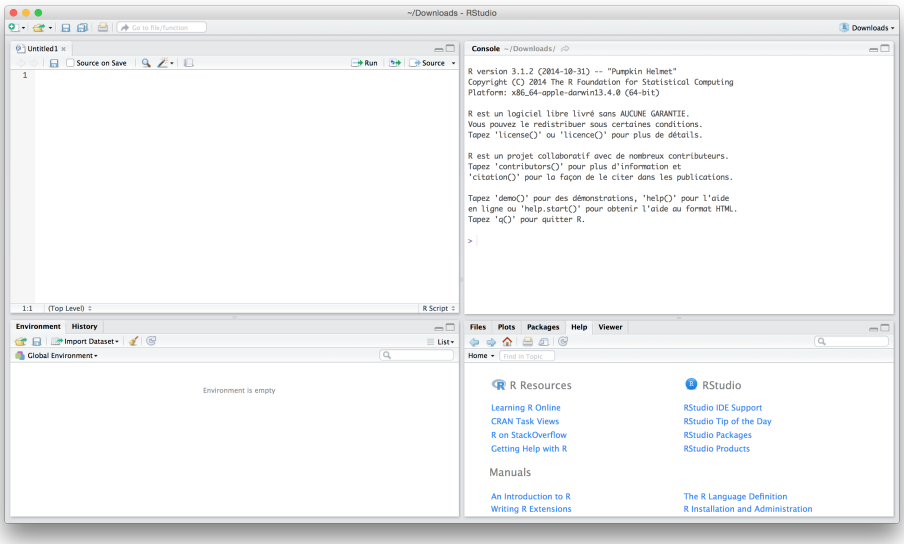


FIGURE 1.2 – Interface graphique de RStudio

On peut noter que RStudio dispose d'un colorateur syntaxique et propose l'auto-complétion (à l'aide de la touche `TAB`).

2

Données

L'objectif de R étant d'analyser des données, il est tout aussi important de connaître les différents types qu'elles peuvent prendre que de savoir les manipuler. Ce chapitre présente dans un premier temps les objets en R, puis s'intéresse à l'importation et à l'exportation de données, et finit par la manière de créer et de manipuler les différents types de données.

2.1 Types de données

R gère des objets, qui sont caractérisés par un **nom**, un **mode**, une **longueur** et un **contenu**.

2.1.1 Mode

Le mode d'un objet est la nature des éléments qui le composent, leur type. On y accède par la fonction `mode()` ou encore `typeof()`.

```
a <- 2
mode(a)

## [1] "numeric"
```

Il existe quatre principaux types :

- *numeric* (numérique).
  On distingue deux types numériques, à savoir les *integers* (entiers) et les *double* ou *real* (réels).

  ```
  a <- 2.0
  typeof(a)
  ## [1] "double"
  is.integer(a) # a est un réel, pas un entier.
  ## [1] FALSE
  b <- 2
  typeof(b)
  ## [1] "double"
  c <- as.integer(b)
  typeof(c)
  ```

```
## [1] "integer"
is.numeric(c) # c est bien un numérique.
## [1] TRUE
```

La variable `c` a la même valeur que la variable `b`, mais elle nécessite un stockage en mémoire moins important (Lafaye de Micheaux et al., 2011). La fonction `is.integer()` retourne `TRUE` lorsque l'objet qui est fourni en paramètre est un entier, `FALSE` sinon. De manière plus générale, les instructions commençant par `is.` et suivies du nom d'un mode permettent de tester si l'objet indiqué en paramètre est de ce mode.

- *character* (caractère).
  Les chaînes de caractères sont placées entre guillemets simples ' ou doubles ".

```
a <- "Hello world!"
a
## [1] "Hello world!"
typeof(a)
## [1] "character"
```

- *logical* (logique, booléen) ;
  Les données de type logique peuvent prendre deux valeurs : `TRUE` ou `FALSE`. Elles répondent à une condition logique.

```
a <- 1 ; b <- 2
a < b
## [1] TRUE
a == 1 # Test d'égalité
## [1] TRUE
a != 1 # Test d'inégalité
## [1] FALSE
is.character(a)
## [1] FALSE
(a <- TRUE)
## [1] TRUE
(a <- T)
## [1] TRUE
```

Comme le montre l'exemple ci-dessus, il est possible d'abréger `TRUE` en `T` ; il en est de même pour `FALSE`, qui peut s'abéger en `F`.

Remarque 2.1.1

Il peut parfois être pratique d'utiliser le fait que `TRUE` peut être automatiquement converti en 1 et `FALSE` en 0.

```
TRUE + TRUE + FALSE + TRUE*TRUE
## [1] 3
```

- *complex* (complexe).
  Les nombres complexes sont caractérisés par leur partie réelle, que l'on peut obtenir à l'aide de la fonction `Re()` ; et par leur partie imaginaire, que l'on obtient grâce à la fonction `Im()`. On créé un nombre complexe à l'aide de la lettre `i`.

```
1i
## [1] 0+1i
z <- 2+3i
Re(z) # Partie réelle de z
## [1] 2
```

```r
Im(z) # Partie imaginaire de z
## [1] 3
Mod(z) # Module de z
## [1] 3.605551
Arg(z) # Argument de z
## [1] 0.9827937
```

2.1.2 Longueur

La longueur d'un objet correspond au nombre d'éléments qu'il contient. On utilise la fonction `length()` pour connaître la longueur d'un objet.

```r
a <- 1
length(a)

## [1] 1
```

2.1.3 Données manquantes

Les données manquantes `NA` (*not available*) sont considérées comme étant de type *logical* par `R`.

```r
x <- NA
typeof(x)

## [1] "logical"

is.na(x)

## [1] TRUE
```

2.1.4 L'objet vide

L'objet `NULL` représente le vide en `R`. Il est de mode `NULL` et de longueur 0. Il ne faut cependant pas le confondre avec un objet vide.

```r
x <- NULL
typeof(x)

## [1] "NULL"

length(x)

## [1] 0

is.null(x)

## [1] TRUE

is.null(list()) # La liste vide n'est pas NULL

## [1] FALSE
```

2.2 Structure des données

Il existe de nombreuses structures servant à organiser les données dans `R`. Cette introduction à `R` n'a pas pour vocation de lister toutes celles qui existantent, aussi, seules quelques unes (les principales) sont présentées dans cette section.
Pour connaître la structure d'un objet, il est possible d'utiliser la fonction `class()`.

2.2.1 Structures de base

2.2.1.1 Vecteurs

Les vecteurs sont extrêmement employés en `R`, et sont composés de données de mêmes types. La création d'un vecteur peut se faire à l'aide de la fonction de concaténation `c()`.

```
c(1,2,3)

## [1] 1 2 3
```

Remarque 2.2.1

Dans l'affichage de la console, `[1]` indique le rang de l'élément qui le suit dans le vecteur dont le contenu est affiché. La section 2.4.2 permettra de mieux comprendre cet affichage.

Il est possible d'attribuer un nom aux éléments d'un vecteur, soit lors de la création, soit *a posteriori*, en utilisant la fonction `names()`.

```
a <- c(nom = "Piketty", prenom = "Thomas", annee_naissance = "1971")
a

##             nom          prenom annee_naissance
##       "Piketty"        "Thomas"          "1971"

b <- c("Piketty", "Thomas", "1971")
b

## [1] "Piketty" "Thomas"  "1971"

names(b) <- c("nom", "prenom", "annee_naissance")
b

##             nom          prenom annee_naissance
##       "Piketty"        "Thomas"          "1971"
```

Lorsque plusieurs types de données sont mélangés lors de la création du vecteur, `R` convertit les données dans le type le plus général.

```
c("deux", 1, TRUE)

## [1] "deux" "1"    "TRUE"
```

La fonction `is.vector()` retourne `TRUE` si l'objet passé en paramètre est un vecteur.

```
a <- c(2,1)
is.vector(a)

## [1] TRUE
```

La fonction `length()` donne le nombre d'éléments qui composent le vecteur.

```
a <- c(2,1)
length(a)

## [1] 2
```

2.2.1.2 Facteurs

Les travaux d'économétrie requièrent souvent l'emploi de variables qualitatives. `R` possède une structure qui gère ces variables : *factor*. On utilise la fonction `factor()` pour encoder un vecteur en facteur.

```
pays <- factor(c("France", "France", "Chine", "Espagne", "Chine"))
pays

## [1] France  France  Chine   Espagne Chine
## Levels: Chine Espagne France

class(pays)

## [1] "factor"
```

La fonction `levels()` retourne les niveaux du facteur.

```
levels(pays)

## [1] "Chine"   "Espagne" "France"
```

Lors des analyses, il arrive de vouloir choisir la modalité de réféfence plutôt que de laisser `R` le faire automatiquement. Il suffit alors d'utiliser la fonction `relevel()`. Il faut toutefois ne pas oublier de réassigner le résultat à l'objet !

```
pays <- relevel(pays, ref = "Espagne")
pays

## [1] France  France  Chine   Espagne Chine
## Levels: Espagne Chine France
```

Si les variables catégorielles sont ordonnées, `R` propose la fonction `ordered()`.

```
revenus <- ordered(c("<1500", ">2000", ">2000", "1500-2000", ">2000", "<1500"),
    levels = c("<1500", "1500-2000", ">2000"))
revenus

## [1] <1500     >2000     >2000     1500-2000 >2000     <1500
## Levels: <1500 < 1500-2000 < >2000
```

2.2.1.3 Dates

On note trois principales classes pour représenter des dates en `R` :

- **Date** : adapté pour les dates, uniquement celles pour lesquelles on ne possède pas d'informations à propos de l'heure (e.g. `"1977-05-27"`) ;
- **POSIXct** et **POSIXlt** : pour les dates avec des heures (e.g. `"1977-05-27 20:00:00"`).

Dates

Les dates sont stoquées comme le nombre de jours depuis `1970-01-01`, avec des valeurs négatives pour des dates antérieures. Le format d'affichage est celui du calendrier grégorien.
La fonction `as.Date()` convertit une chaîne de caratcères en date. Si le format est de la forme `%Y-%m-%d` ou `%Y/%m/%d` (avec `%Y` l'année avec le siècle, `%m` le mois sur deux chiffres et `%d` le jour sur deux chiffres), il n'est pas nécessaire de renseigner le paramètre `format` de la fonction. Le paramètre `origin` permet quant à lui de préciser une origine différente de `1970-01-01`.

```
(d <- as.Date("2015-10-21"))

## [1] "2015-10-21"

class(d)

## [1] "Date"

unclass(d)

## [1] 16729

# La date du jour
as.Date(Sys.time())

## [1] "2015-11-26"

# Avec un format non standard
(d <- as.Date("2015 21 10", format = ("%Y %d %m")))

## [1] "2015-10-21"

(d <- as.Date("21 Octobre, 2015", format = ("%d %B, %Y")))

## [1] "2015-10-21"
```

La fonction `unclass()` utilisée dans le précédent exemple permet de voir la valeur stockée en interne.

Comme on peut le voir dans le précédentexemple, le paramètre `format` a été utilisé. Ce paramètre permet de préciser comment la date est écrite dans la chaîne de caractères, pour que la conversion en objet de classe `date` soit possible. Cette conversion est réalisée, en interne, par la fonction `strptime()`. La page d'aide de cette fonction (`?strptime`) donne les différents codes qui spécifient les formats de date. Attention, certains de ces formats dépendent de paramètres locaux à la machine (des exemples sont donnés plus loin).
Les codes spécifiant les formats de date sont consignés dans le tableau suivant.

Code	Description	Exemple
%a	Abréviation du jour de la semaine (dépend du lieu)	Mer
%A	Jour de la semaine complet (dépend du lieu)	Mercredi
%b	Abréviation du mois (dépend du lieu)	oct
%B	Nom du mois complet (dépend du lieu)	octobre
%c	Date et heure (dépend du lieu) au format %a %e %b %H:%M:%S %Y	Mer 21 oct 13:55:44 2015
%C	Siècle (00-99) -1 (partie entière de la division de l'année par 100)	20
%d	Jour du mois (01–31)	21
%D	Date au format %m/%d/%y	10/21/15
%e	Jour du mois en nombre décimal (1–31)	21
%F	Date au format %Y-%m-%d	2015-10-21
%h	Même chose que %b	oct
%H	Heure (00–24)	13
%I	Heure (01–12)	01
%j	Jour de l'année (001–366)	294
%m	Mois (01–12)	10
%M	Minute (00-59)	55
%n	Retour à la ligne en *output*, caractère blanc en *input*	
%p	AM/PM	PM
%r	Heure au format 12 AM/PM	01:55:44 PM
%R	Même chose que %H:%M	13:55
%S	Seconde (00-61)	44
%t	Tabulation en *output*, caractère blanc en *input*	
%T	Même chose que %H:%M:%S	13:55:44
%u	Jour de la semaine (1–7), commence le lundi	3
%U	Semaine de l'anné (00–53), dimanche comme début de semaine, et le premier dimanche de l'année définit la semaine 1	42
%V	Semaine de l'année (00-53). Si la semaine (qui commence un lundi) qui contient le 1er janvier a quatre jours ou plus dans la nouvelle année, alors elle est considérée comme la semaine 1. Sinon, elle est considérée comme la dernière de l'année précédente, et la semaine suivante est considérée comme semaine 1 (norme ISO 8601)	43
%w	Jour de la semaine (0–6), dimanche étant 0	3
%W	Semaine de l'année (00–53), le lundi étant le premier jour de la semaine, et typiquement, le premier lundi de l'année définit la semaine 1 (convientionn G.B.)	42
%x	Date (dépend du lieu)	21.10.2015
%X	Heure (dépend du lieu)	13:55:44
%y	Année sans le "siècle" (00–99)	15
%Y	Année (en *input*, uniquement de 0 à 9999)	2015
%z	*offset* en heures et minutes par rapport au temps UTC	+0200
%Z	Abréviation du fuseau horaire (en *output* seulement)	CEST

POSIXct et **POSIXlt**

Si les dates à représenter contiennent des heures, il faut préférer le format **POSIXct** ou **POSIXlt**. Avec **POSIXct**, les dates sont stockées en secondes depuis les débuts d'Unix (`1970-01-01 01:00:00`). Avec **POSIXlt**, les dates sont stockées sous forme de liste dont les éléments correspondent à différentes composantes de la date (voir le tableau ci-après). Les fonctions `as.POSIXct()` et `as.POSIXlt()` permettent de convertir une chaîne de caractère représentant une date en objet `POSIXct` ou `POSIXlt`. À nouveau, la fonction `strptime()` est utilisée en interne pour réaliser la conversion des chaînes de caractères en dates.

```
# POSIXct
d <- as.POSIXct("2015-10-21 13:55:44")
unclass(d)

## [1] 1445428544
## attr(,"tzone")
## [1] ""

unclass(as.POSIXct("1970-01-01 01:00:00"))

## [1] 0
## attr(,"tzone")
## [1] ""

# POSIXlt
d <- as.POSIXlt("2015-10-21 13:55:44")
unlist(unclass(d))

##    sec    min   hour   mday    mon   year   wday   yday  isdst   zone
##   "44"   "55"   "13"   "21"    "9"  "115"    "3"  "293"    "1" "CEST"
## gmtoff
##     NA
```

Les éléments de la liste d'un objet de classe `POSIXlt` sont répertoriés dans le tableau suivant (voir page d'aide `?DateTimeClasses`).

Code	Description
`sec`	secondes (0–61)
`min`	minutes (0–59)
`hour`	heures (0–23)
`mday`	jour du mois (1–31)
`mon`	mois après le premier de l'année (0–11)
`year`	années depuis 1900
`wday`	jour de la semaine (0–6), début de semaine le dimanche
`yday`	jour de l'année (0–365)
`isdst`	indicateur d'heure d'été (positif si appliquable, zéro sinon ; négatif si inconnu)
`zone`	Abréviation du fuseau horaire local (`""` si inconnu, ou UTC)
`gmtoff`	décalage en minutes par rapport au temps GMT (`NA` ou `0` si inconnu)

Le *package* `lubridate` propose également des fonctions pour gérer les dates. Il est nécessaire de charger le *package* en mémoire.
Pour convertir une chaîne de caractères représentant une date, il suffit d'utiliser une des trois fonctions suivantes : `ymd()`, `mdy()` et `dmy()`. Les trois lettres de ces fonctions font référence à l'année (`y`), le mois (`m`) et le jour (`d`). La position de ces lettres dans le nom de la fonction doit correspondre à celui des éléments correspondants dans la chaîne de caractères. Ainsi, `ymd()` s'attend à trouver d'abord l'année, puis le mois et enfin le jour dans la chaîne de caractères. Le gros avantage de ces fonctions est qu'il n'est pas nécessaire de préciser les séparateurs, une analyse est faite automatiquement pour le deviner.

```
library(lubridate)
ymd("2015-10-21")

## [1] "2015-10-21 UTC"

mdy("10.21-15")

## [1] "2015-10-21 UTC"

dmy("21 oct 2015")

## [1] "2015-10-21 UTC"
```

Il est possible de préciser le fuseau horaire (à condition que le nom soit reconnu par le système d'exploitation), en renseignant le paramètre `tz`.

```
ymd("2015-10-21", tz = "Pacific/Auckland")

## [1] "2015-10-21 NZDT"
```

Remarque 2.2.2

La fonction `OlsonNames()` retourne un vecteur contenant tous les fuseaux horaires disponibles.

S'il y a des informations d'heures dans les chaînes de caractères repésentant les dates, il suffit d'ajouter `h` (heures), ou `hm (heures et minutes)` ou `hms` (heures, minutes et secondes) au nom de la fonction, en séparant par un trait de soulignement.

```
ymd_hms("2015-10-21 10:00:50")

## [1] "2015-10-21 10:00:50 UTC"
```

2.2.1.4 Matrices

Les matrices sont des vecteurs auxquels un attribut *dim* de dimension est ajouté. Une matrice est un vecteur pour lequel l'attribut *dim* est de longueur 2. La fonction `matrix()` permet de créer une matrice en `R`. Le code ci-après stoque dans `X` une matrice de 3 lignes (`nrow = 3`) et de 2 colonnes (`ncol = 2`). Cette matrice est remplie par colonnes successives par les éléments du vecteur `1:6`.

```r
X <- matrix(1:6, ncol = 2, nrow = 3)
X
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Pour que la matrice soit remplie par lignes successives plutôt que par colonne, il suffit d'ajouter le paramètre `byrow` à la fonction `matrix()` et de lui donner la valeur `TRUE`.

```r
Y <- matrix(1:6, ncol = 2, nrow = 3, byrow = TRUE)
Y
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

Lorsque la dimension dépasse 2, on parle de tableau (*array*). Par exemple, le code ci-après permet de créer un tableau de dimension $2 \times 4 \times 3$.

```r
Z <- array(1:24, dim = c(2, 4, 3))
Z
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   19   21   23
## [2,]   18   20   22   24
```

Pour les matrices comme pour les tableaux, les données doivent toutes être du même type.
Les fonctions `nrow()` et `ncol()` donnent le nombre de lignes (*rows*) et de colonnes (*columns*) d'une matrice. La fonction `dim()` donne un vecteur contenant les dimensions de la matrice ou du tableau donné en paramètre.

```r
X <- matrix(1:6, ncol = 2, nrow = 3)
dim(X)
```

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

```
nrow(X)

## [1] 3

ncol(X)

## [1] 2
```

2.2.1.5 Listes

Les listes dans R sont les structures les moins rigides. Elles peuvent contenir des éléments de modes différents, y compris des listes. Le mode des éléments de la liste n'est pas altéré, ce qui peut s'avérer utile dans certains cas.

```
(personne <- list("Piketty", "Thomas", "1971"))

## [[1]]
## [1] "Piketty"
##
## [[2]]
## [1] "Thomas"
##
## [[3]]
## [1] "1971"

class(personne)

## [1] "list"
```

Comme pour les vecteurs, il est possible de nommer les éléments de la liste, ce qui permet une meilleure lecture et un accès aux données parfois plus pratique. La section 2.4 apportera plus de détails à ce propos.

```
personne <- list(`nom de famille` = "Piketty", prenom = "Thomas", annee = 1971)
personne

## $`nom de famille`
## [1] "Piketty"
##
## $prenom
## [1] "Thomas"
##
## $annee
## [1] 1971
```

Pour aplatir une liste, R propose une fonction : `unlist()`. Cette fonction retourne un vecteur contenant tous les composants de la liste. Il faut toutefois garder à l'esprit que R se charge alors de convertir tous les éléments dans le mode le plus général. De fait, si cette fonction peut s'avérer pratique dans le cas de scalaires par exemple, elle peut également être destructrice lorsque mal employée.

```r
unlist(personne)

## nom de famille         prenom          annee
##      "Piketty"       "Thomas"         "1971"
```

2.2.2 Bases de données

2.2.2.1 Data frames

Les tableaux de données (ou *data.frames*) sont au cœur de nombreuses procédures dans `R` notemment lorsqu'on effectue une régression linéaire. Un *data frame* est une liste composée d'un ou plusieurs vecteurs de même longueur. Le mode des vecteurs n'est pas nécessairement identique à celui des autres. Une manière simple de se représenter les tableaux de données est de penser à une matrice dans laquelle les lignes représentent des individus et les colonnes des caractéristiques propres à ces individus. La création d'un tableau de données se fait *via* la fonction `data.frame()`. La fonction `as.data.frame()` permet quant à elle de convertir un objet d'un autre type en *data frame*.
L'exemple ci-après utilise les données du *dataset* "*women*", présent dans le *package* "*datasets*". Il s'agit de la taille (en pouces) et du poids (en livres) moyen des femmes américaines âgées entre 30 et 39 ans.

```r
femmes <- data.frame(height = c(58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
    69, 70, 71, 72), weight = c(115, 117, 120, 123, 126, 129, 132, 135, 139,
    142, 146, 150, 154, 159, 164))
femmes

##    height weight
## 1      58    115
## 2      59    117
## 3      60    120
## 4      61    123
## 5      62    126
## 6      63    129
## 7      64    132
## 8      65    135
## 9      66    139
## 10     67    142
## 11     68    146
## 12     69    150
## 13     70    154
## 14     71    159
## 15     72    164

is.data.frame(femmes)

## [1] TRUE

class(femmes)

## [1] "data.frame"
```

Comme pour les matrices, on peut obtenir les dimensions d'un *data frame* avec la fonction `dim()`, le nombre de lignes avec `nrow()` et le nombre de colonnes avec `ncol()`.

2.2.2.2 Data table

Le *package* `data.table` propose une structure de données appelée `data.table` [1], qui se veut être une alternative aux objets *data.frame*. Il faut toutefois noter qu'un objet de classe `data.table` est également un `data.frame`. L'idée derrière les `data.table` est de réduire la longueur du code, de proposer une syntatxe plus facile à écrire et lire, mais aussi de réduire les temps de calculs. La syntaxe se rapproche un peu du `SQL`, mais évite pas mal de lourdeurs de ce langage.

Il est nécessaire de charger le *package*.

```
library(data.table)

##
## Attaching package: 'data.table'
##
## The following objects are masked from 'package:lubridate':
##
##     hour, mday, month, quarter, wday, week, yday, year
```

```
femmes <- data.table(height = c(58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
    69, 70, 71, 72), weight = c(115, 117, 120, 123, 126, 129, 132, 135, 139,
    142, 146, 150, 154, 159, 164))
femmes
```

```
##     height weight
##  1:     58    115
##  2:     59    117
##  3:     60    120
##  4:     61    123
##  5:     62    126
##  6:     63    129
##  7:     64    132
##  8:     65    135
##  9:     66    139
## 10:     67    142
## 11:     68    146
## 12:     69    150
## 13:     70    154
## 14:     71    159
## 15:     72    164
```

```
is.data.frame(femmes)
```

```
## [1] TRUE
```

```
is.data.table(femmes)
```

```
## [1] TRUE
```

```
class(femmes)
```

```
## [1] "data.table" "data.frame"
```

1. La vignette qui accompagne le *package* est très bien faite : http://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.pdf.

2.3 Importation, exportation et création de données

Pour pouvoir analyser des données, il faut pourvoir les charger en mémoire. Heureusement, il n'est pas nécessaire de les rentrer à la main, il existe des fonctions qui permettent de les importer, depuis divers formats. Il est également possible d'enregistrer les données dans plusieurs formats. Puisqu'il est souvent utile de créer des données à l'intérieur de la session (par exemple des années, des variables indicatrices, des noms de mois, etc.), cette section présente quelques fonctions utiles à la création de données.
Avant toute chose, il convient de préciser qu'un environnement de travail (*working directory*) est assigné à `R`. C'est dans ce répertoire, que l'on peut connaître grâce à l'instruction `getwd()` et modifier à l'aide de `setwd()`, que `R` ira chercher les fichiers à importer et enregistrer les données lors de l'exportation. Si on souhaite lire des fichiers qui sont hors de ce répertoire, il est nécessaire d'indiquer à `R` leur chemin (relatif ou absolu).

```
getwd()

## [1] "/Users/ewengallic/Documents"
```

Attention, l'utilisation de la fonction `setwd()` est considérée comme une **mauvaise pratique**. Deux exemples permettent de comprendre que l'utilisation de cette fonction agit comme une bombe à retardement :

- lors d'un travail en collaboration, il est fort probable que la ou les personnes avec qui l'on travaille ne s'appellent pas comme nous ; de fait, le chemin indiquant le répertoire courant sera faux ;
- lors de l'acquisition d'un nouvel ordinateur, il est fort possible que l'arborescence ne soit pas exactement la même ; à nouveau, le chemin indiquant le répertoire de travail ne sera plus le bon.

Quelle pratique adopter, dans ce cas ? La réponse est simple : travailler à l'aide de projets. RStudio propose cette alternative [2], et dans ce cas, le chemin vers le répertoire de travail est défini à l'extérieur. Il suffit alors d'utilier des chemins relatifs lors de l'importation ou l'exportation de données, graphiques, etc.

Remarque 2.3.1

La fonction `list.files()` est très pratique : elle permet de lister tous les documents et répertoires contenus dans le répertoire de travail, ou dans n'importe quel répertoire si le paramètre `path` renseigne le chemin.

2.3.1 Importation

Lorsque les données sont présentes dans un fichier texte (ASCII), `R` propose deux fonctions : `read.table()` et `scan()`. Pour les données stockées dans des fichiers aux formats différents, il est nécessaire de s'appuyer sur d'autres fonctions, non contenues dans le *package* `base`. C'est le cas pour les fichiers Excel©, SAS© ou STATA©.

2.3.1.1 La fonction `read.table`

Cette fonction est très pratique lorsque les données sont déjà organisées dans un tableau dans le fichier à importer. Elle crée alors un *data frame* contenant les données. Dans l'exemple fictif ci-dessous, les données du fichier `fichier.txt` sont stockées dans le *data frame* `df`. Les variables seront nommées par défaut `V1`, `V2`, ...

2. Menu *File*, *New Project...*

```
df <- read.table("fichier.txt")
```

Il existe de nombreux paramètres pour la fonction `read.table()`. Seuls les principaux seront abordés ici. Le lecteur pourra consulter l'aide `?read.table()` pour de plus amples détails.

Paramètre	Description
file	Le nom du fichier, doit être une chaîne de caractères. Il peut être précédé du chemin relatif ou absolu. Attention (utile pour les utilisateurs de Windows) le caractère "\" est proscrit, et doit être remplacé par "/" ou bien "\\". À noter qu'il est possible de saisir une adresse web (URL) en guise de chaîne de caractère.
header	Valeur logique (`header=FALSE` par défaut) indiquant si la première ligne contient les noms de variables.
sep	Le séparateur de champ dans le fichier (chaîne vide par défaut, ce qui est au final traduit par une espace comme séparation). Par exmple, `sep=";"` si les champs sont séparés par un point-virgule, ou encore `sep="\t"` s'ils sont séparés par une tabulation.
dec	Le caractère employé pour les décimales (par défaut, `dec="."`).
row.names	Un vecteur contenant le nom des lignes (de type caractère), ou bien le numéro ou le nom d'une variable du fichier. En omettant ce paramètre, les lignes sont numérotées.
na.strings	Une chaîne de caractère (ou un vecteur de chaînes de caractères) indiquant la valeur des données manquantes (par défaut, `na.strings="NA"`). Ces données manquantes seront converties en `NA`.
colClasses	Un vecteur de caractères indiquant les modes des colonnes.

Remarque 2.3.2

Il est possible d'employer l'instruction `file.choose()` pour ouvrir une boîte de dialogue au lieu d'écrire le chemin complet vers le fichier de données. Bien que cette interface puisse paraître agréable à l'utilisateur novice, elle risque de vite devenir contraignante si le fichier `R` est amené à être exécuté plusieurs fois, puisqu'à chaque appel, la boîte de dialogue s'ouvrira.

```
df <- read.table(file.choose())
```

Remarque 2.3.3

Il existe des variantes de `read.table()` qui s'appuient sur cette fonction pour proposer à l'utilisateur des fonctions directement capables de lire leur fichier de données, sans avoir à changer les paramètres `sep` et `decim`.

Fonction	Séparateur de champs	Séparateur décimal
`read.csv()`	`","`	`"."`
`read.csv2()`	`";"`	`","`
`read.delim()`	`"\t"`	`"."`
`read.delim2()`	`"\t"`	`","`

Remarque 2.3.4

Le *package* `readr` propose des alternatives extrêmement rapides aux fonctions `read.table()` et dérivées. L'équivalent de `read.table()` est `read_delim()`. Il suffit de lui préciser le séparateur de champ (`delim`), le séparateur des miliers étant automatiquement détecté. Attention, une ligne d'en-tête doit être présente dans le fichier de données.

```
df <- read_delim("data.csv", ";")
```

Fonction	Séparateur de champs	Séparateur décimal
`read_csv()`	`","`	`"."`
`read_csv2()`	`";"`	`","`
`read_tsv()`	`"\t"`	`"."`

2.3.1.2 La fonction `scan`

La fonction `scan()` est beaucoup plus souple que `read.table()`. Son emploi est requis dès que les données ne sont pas organisées comme un tableau. La nature des variables peut être spécifiée en renseignant le paramètre `what`. On retrouve la plupart des paramètres de la fonction `read.table()`. Le tableau ci-après présente les principaux ; le lecteur est invité à se reporter au fichier d'aide (`?scan`) pour de plus amples détails.

Paramètre	Description
file	Le nom du fichier, doit être une chaîne de caractères. Il peut être précédé du chemin relatif ou absolu. Attention (utile pour les utilisateurs de Windows) le caractère "\" est proscrit, et doit être remplacé par "/" ou bien "\\". À noter qu'il est possible de saisir une adresse web (URL) en guise de chaîne de caractère.

continue sur la page suivante

suite du tableau

Paramètre	Description
what	Permet de précider le type des données lues.
nmax	Si présent, indique le nombre de données à lire, ou, si `what` est une liste, le nombre maximum de lignes à lire.
n	Le nombre de données à lire (pas de limite par défaut).
sep	Le séparateur de champ dans le fichier (chaîne vide par défaut, ce qui est au final traduit par une espace comme séparation). Par exmple, `sep=";"` si les champs sont séparés par un point-virgule, ou encore `sep="\t"` s'ils sont séparés par une tabulation.
dec	Le caractère employé pour les décimales (par défaut, `dec="."`).
skip	Le nombre de ligne à omettre avant de procéder à l'importation.
na.strings	Une chaîne de caractère (ou un vecteur de chaînes de caractères) indiquant la valeur des données manquantes (par défaut, `na.strings="NA"`). Ces données manquantes seront converties en `NA`.
flush	Valeur logique (par défaut, `FALSE`). Si `TRUE` scan va à la ligne suivante après avoir lu le dernier champ requis, ce qui permet d'ajouter des commentaires après le dernier champ, dans le fichier de données, mais empêche de mettre plus d'une observations par ligne.

Remarque 2.3.5

Si dans l'appel de `scan()`, le paramètre `file` est comme suit : `file=""`, l'utilisateur est invité à entrer les données au clavier. Pour clore la saisie, il suffit de terminer par une ligne blanche.

2.3.1.3 La fonction `read.fwf`

Parfois, les données que l'on récupère sont dans un format à largeur fixée (*fixed width format*). C'est le cas par exemple pour les données météorologiques issues du site de la *National Oceanic and Atmospheric Administration* (NOAA). `R` propose une fonction pour importer rapidement ce type de données : `read.fwf()`. Un paramètre fait son apparition par rapport à la fonction `read.table()`, il s'agit de `widths`, qui permet d'indiquer la largeur de chaque champ.
Prenons un exemple. Si le fichier de données est nommé "`data_fwf.txt`" et est présent dans le répertoire de travail, et qu'il contient les données suivantes :

```
FRANCEFR14.01
FRANCEFR23.02
FRANCEFR32.96
ITALIEIT15.90
ITALIEIT25.48
ITALIEIT34.32
```

alors, l'importation de ces données dans R pourra se faire de la manière suivante :

```
(df_fwf <- read.fwf("data_fwf.txt", widths = c(6,2,4)))

##       V1 V2   V3
## 1 FRANCE FR 14.0
## 2 FRANCE FR 23.0
## 3 FRANCE FR 32.9
## 4 ITALIE IT 15.9
## 5 ITALIE IT 25.4
## 6 ITALIE IT 34.3
```

Remarque 2.3.6

La fonction `read_fwf()` du *package* `readr` est une alternative à la fonction `read.fwf()`.

2.3.1.4 Importation depuis Excel©

Nous allons voir comment importer des données directement depuis un fichier à l'extension `.xls` ou `.xlsx`. Il existe plusieurs méthodes, qui sont bien expliquées dans les documents de référence de ce document. Nous allons uniquement aborder la manière à l'aide des fonctions `read.xls()` et `read_excel()` contenues dans les *packages* `gdata` et `readxl` respectivement. Par souci de reproductibilité, nous allons nous appuyer sur le fichier `iris.xls`, contenu dans le répertoire du *package* `gdata`.

```
library("gdata")
# Récupérer le lien vers le fichier iris.xls
xlsfile <- file.path(path.package("gdata"), "xls", "iris.xls")
iris <- read.xls(xlsfile)  # Créé un fichier csv temporaire
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Par défaut, la fonction `read.xls` importe les données de la première feuille. Le paramètre `sheet` permet d'indiquer la feuille souhaitée, en la référençant soit par son numéro, soit par son nom.

Remarque 2.3.7

En pratique, si le fichier contient des formules, il est préférable d'utiliser la fonction "collage spécial" d'Excel© pour coller les valeurs dans un nouveau fichier, ou bien d'enregistrer le fichier sous un format `.txt` dans Excel©.

Remarque 2.3.8

Pour les utilisateurs sous Windows, il est peut-être nécessaire d'indiquer le chemin vers l'interprète perl (le fichier se nomme `perl.exe`)

```
perl <- "C:\\Program Files\\FusionInventory-Agent\\perl\\bin\\perl.exe"
iris <- read.xls(xlsfile, perl = perl)  # Créé un fichier csv temporaire
```

Avec `read_xl()`, la procédure est identique :

```
library("readxl")
# Récupérer le lien vers le fichier iris.xls
xlsfile <- file.path(path.package("gdata"), "xls", "iris.xls")
iris <- read_excel(xlsfile)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

2.3.1.5 Importation depuis d'autres formats

Le *package* `foreign`, installé de base sous Windows, mais non chargé automatiquement en mémoire, permet de lire des données stockées dans de nombreux formats (par exemple DBF©, STATA©, SPSS©, etc.). Pour les fichiers `sas7bdat`, produits par SAS©, on peut utiliser la fonction `read.sas7bdat()` du *package* `sas7bdat`.

2.3.2 Exportation

Pour enregistrer des données depuis un *data.frame*, un vecteur ou une matrice, la fonction `write.table()` peut être utilisée. Par exemple, si le *data.frame* se nomme `donnees`, l'instruction ressemble à :

```
write.table(donnees, file = "nom_fichier.txt", sep = ";")
```

Il existe d'autres paramètres, que le lecteur pourra consulter dans l'aide de la fonction `write.table()`.

Remarque 2.3.9

La fonction `write_csv()` du *package* `readr` est une alternative à la fonction `write.csv()`.

La fonction `save()` permet elle aussi de sauvegarder des données en dehors de la session `R`, et cette fois, n'importe quelle classe d'objet peut être sauvegardée. Par exemple, si l'utilisateur souhaite sauvegarder les objets `d1`, `d2` et `d3`, il lui suffit de rentrer l'instruction suivante :

```
save(d1, d2, d3, file = "nom_fichier.RData")
```

Pour charger les fichiers d'extention `.RData`, il faut utiliser la fonction `load()` :

```
load("nom_fichier.RData")
```

Remarque 2.3.10

Pour sauvegarder tous les éléments de la session `R`, la fonction `save.image()` peut être appelée, ce qui revient au même que d'évaluer l'instruction `save(list = ls(all=TRUE), file=".RData")`.

2.3.3 Génération

2.3.3.1 Séquences régulières

La fonction de concaténation, abordée précédemment est un moyen simple de créer un vecteur de données.

```
c(1,2,3)

## [1] 1 2 3
```

La fonction `seq()` est extrêmement utile. Elle crée un vecteur de séquences de nombres. Le premier paramètre définit la valeur du début de séquence, tandis que le second paramètre définit la valeur finale. Le paramètre `by` définit l'incrément, et vaut 1 par défaut. Le paramètre `length.out` permet quant à lui de définir la longueur souhaitée de la séquence, et définit donc automatiquement la valeur de l'incrément. Il faut noter que `by` et `length.out` ne peuvent évidemment pas être utilisés en même temps.

```
(a <- 2:4)

## [1] 2 3 4

seq(1,3)

## [1] 1 2 3

seq(1,3, by = 0.5)

## [1] 1.0 1.5 2.0 2.5 3.0

seq(1,3, length.out = 10)

##  [1] 1.000000 1.222222 1.444444 1.666667 1.888889 2.111111 2.333333
##  [8] 2.555556 2.777778 3.000000
```

La fonction `:()` génère une séquence de données espacées d'une unité. Attention, cette fonction, que l'on peut voir comme un opérateur, est prioritaire sur les opérations arithmétiques.

```
1:10

##  [1]  1  2  3  4  5  6  7  8  9 10

1.5:10

## [1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

Comme mentionné en section 2.1.1, le stockage des données de mode *integer* prend moins de place que celles de type *numeric*. Avec les fonctions `seq()` ou `:()`, les données sont de type *integer*.

```
class(c(2,3,4))

## [1] "numeric"

class(seq(2,4))

## [1] "integer"
```

Pour créer une séquence de valeurs avec duplication du premier paramètre, on peut utiliser la fonction `rep()`. En jouant avec les paramètres, on peut créer des réplications différentes. Voici quelques exemples.

```
rep(1, 3) # Répète trois fois la valeur 1

## [1] 1 1 1

rep(1:2, 3) # Répète trois fois la séquence 1:2

## [1] 1 2 1 2 1 2

rep(1:2, each = 3) # Répète chaque élément de la séquence 1:2 trois fois

## [1] 1 1 1 2 2 2

# Répète deux fois la séquence dans laquelle
# les éléments de la séquence 1:2 sont répétés trois fois
rep(1:2, 2, each = 3)

##  [1] 1 1 1 2 2 2 1 1 1 2 2 2

# Répète la séquence 1:2 jusqu'à ce que
# la longueur du résultat soit égale à 3 (le résultat peut être tronqué)
rep(1:2, length.out = 3)

## [1] 1 2 1
```

La fonction `seq_len()` crée une suite de nombres entiers allant de 1 au nombre passé à l'unique paramètre `length.out`.

```
seq_len(4)

## [1] 1 2 3 4
```

La fonction `sequence()` génère une suite de nombres. Pour chaque élément passé au paramètre `nvec`, une séquence de nombre allant de 1 à cet élément est créée. Les séquences d'entiers ainsi créées sont concaténées.

```
sequence(2:4)

## [1] 1 2 1 2 3 1 2 3 4
```

```r
sequence(c(3,5))

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

La fonction `gl()` permet de créer des séries de facteurs. Elle requière deux paramètres : `n`, pour indiquer le nombre de niveaux souhaité, et `k` pour indiquer le nombre de réplications voulu. Il est possible de définir les étiquettes pour chacun des niveaux, en renseignant le paramètre `labels` ou encore de préciser si les niveaux des facteurs doivent être ordonnés, avec le paramètre logique `ordered`. Le paramètre `length` permet quant à lui de définir la longueur souhaitée du résultat.

```r
gl(2, 4)

## [1] 1 1 1 1 2 2 2 2
## Levels: 1 2

gl(2, 4, length = 10)

##  [1] 1 1 1 1 2 2 2 2 1 1
## Levels: 1 2

gl(2, 4, labels = c("Oui", "Non"))

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

Enfin, `expand.grid()` est une fonction très utile pour générer toutes les combinaisons possibles des vecteurs donnés en paramètre.

```r
expand.grid(age = seq(18, 20), sexe = c("Femme", "Homme"), fumeur = c("Oui",
    "Non"))

##    age  sexe fumeur
## 1   18 Femme    Oui
## 2   19 Femme    Oui
## 3   20 Femme    Oui
## 4   18 Homme    Oui
## 5   19 Homme    Oui
## 6   20 Homme    Oui
## 7   18 Femme    Non
## 8   19 Femme    Non
## 9   20 Femme    Non
## 10  18 Homme    Non
## 11  19 Homme    Non
## 12  20 Homme    Non
```

2.3.3.2 Séquences pseudo-aléatoires

`R` propose des fonctions pour de nombreuses distributions de probabilité. Pour chaque distribution de probabilité, il y a en général quatre fonctions, dont le nom commence par une des quatre lettres suivantes : '`r`' (*random*), '`d`' (*density*), '`p`' (*probability*) et '`q`' (*quantile*). Les fonctions commençant par la lettre '`r`' permettent de générer des données issues d'une loi de

probabilité donnée ; celles commençant par la lettre 'd' correspondent aux densités de probabilité (ou fonctions de masse dans le cas d'une variable aléatoire discrète) ; celles dont la première lettre est un 'p' donnent la densité de probabilité cumulée (fonctions de répartition) ; et enfin celles commençant par la lettre 'q' donnent les valeur de quantiles (fonctions quantiles).
Prenons un exemple avec la loi $\mathcal{N}(0,1)$:

```
rnorm(n = 1, mean = 0, sd = 1)

## [1] -1.650897

dnorm(1) == 1/sqrt(2*pi) * exp(-1/2)

## [1] TRUE

pnorm(1.96)

## [1] 0.9750021

qnorm(0.025) ; qnorm(0.975)

## [1] -1.959964
## [1] 1.959964
```

R propose des fonctions pour les lois de probabilité discrètes suivantes (liste non exhaustive) :
- Binomiale : `rbinom(n, size, prob)` ;
- Poisson : `rpois(n, lambda)` ;
- Géométrique : `rgeom(n, prob)` ;
- Hyper-géométrique : `rhyper(nn, m, n, k)` ;
- Binomiale négative : `rnbinom(n, size, prob, mu)`.

R propose des fonctions pour les lois de probabilité continues suivantes (liste non exhaustive) :
- Normale : `rnorm(n, mean = 0, sd = 1)` ;
- Student : `rt(n, df, ncp)` ;
- Khi-deux : `rchisq(n, df, ncp = 0)` ;
- Fisher : `rf(n, df1, df2, ncp)` ;
- Exponentielle : `rexp(n, rate = 1)` ;
- Uniforme : `runif(n, min = 0, max = 1)` ;
- Beta : `rbeta(n, shape1, shape2, ncp = 0)` ;
- Logistique : `rlogis(n, location = 0, scale = 1)` ;
- Log-Normale : `rlnorm(n, meanlog = 0, sdlog = 1)` ;
- Gamma : `rgamma(n, shape, rate = 1, scale = 1/rate)` ;
- Weibull : `rweibull(n, shape, scale = 1)` ;

2.4 Manipulation des données

Dans la section 2.2, nous avons vu comment créer différents objets en R. Cette section présente d'abord les opérateurs, puis les façons d'accéder aux données en fonction du type d'objet, les calculs sur les listes et vecteurs, et enfin les calculs matriciels.

2.4.1 Opérateurs

Il existe trois sortes d'opérateurs en R, à savoir arithmétiques, de comparaison et logiques.

2.4.1.1 Opérateurs arithmétiques

Les opérateurs arithmétiques agissent sur des vecteurs ou des matrices. Ils opèrent sur des objets de type numérique, complèxe ou logique. En voici la liste :

```
x <- c(1, 2, 3, 4, 5)
y <- c(2, 5, 2, 8, 1)
x + y  # Addition

## [1]  3  7  5 12  6

x - y  # Soustraction

## [1] -1 -3  1 -4  4

x * y   # Multiplication

## [1]  2 10  6 32  5

x / y   # Division

## [1] 0.5 0.4 1.5 0.5 5.0

x^y  # Puissance

## [1]     1    32     9 65536     5

x %% y  # Modulo

## [1] 1 2 1 4 0

x %/% y # Division entière

## [1] 0 0 1 0 5
```

2.4.1.2 Opérateurs de comparaison

Les opérateurs de comparaison agissent sur des vecteurs, des matrices, des tableaux de données et des listes. Le type de données n'est pas restreint comme pour les opérateurs arithmétiques.

```
x <- seq_len(5)
x < 2  # Inférieur à

## [1]  TRUE FALSE FALSE FALSE FALSE

x <= 2  # Inférieur ou égal à

## [1]  TRUE  TRUE FALSE FALSE FALSE

x > 2   # Supérieur à

## [1] FALSE FALSE  TRUE  TRUE  TRUE

x >= 2  # Supérieur ou égal à
```

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

x == 2  # Égal à

## [1] FALSE  TRUE FALSE FALSE FALSE

x != 2  # Différent de

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

Attention, il faut être prudent avec la fonction d'égalité. En effet, un exemple simple (voir Paradis (2002)) montre que deux objets qui nous semblent identiques ne le sont pas aux yeux de la machine, à cause des approximations effectuées lors des calculs. Il convient alors dans certains cas d'utiliser la fonction `all.equal()` plutôt que l'opérateur logique `==` ou la fonction `identical()`.

```
0.9 == (1 - 0.1)

## [1] TRUE

all.equal(0.9, 1-0.1)

## [1] TRUE

identical(0.9, 1 - 0.1)

## [1] TRUE

0.9 == (1.1 - 0.2)

## [1] FALSE

all.equal(0.9, 1.1-0.2)

## [1] TRUE

identical(0.9, 1.1-0.2)

## [1] FALSE
```

En fait, la fonction `all.equal()` donne une égalité approximative, à l'aide d'un seuil de tolérance !

```
all.equal(0.9, 1.1-0.2, tolerance = 1e-16)

## [1] "Mean relative difference: 1.233581e-16"
```

2.4.1.3 Opérateurs logiques

Les opérateurs logiques opèrent sur un ou deux objets de type logique. Les opérateurs 'Et' et 'Ou' existent en `R` sous deux formes : la forme simple (`&` et `|`) et la forme double (`&&` et `||`). La forme simple opère sur chaque élément des objets, et retourne le résultat de l'opération pour chaque élément. La forme double, quant à elle, opère uniquement sur le premier élément des objets.

```
x <- c(TRUE, TRUE, FALSE, FALSE)
y <- c(TRUE, FALSE, TRUE, FALSE)
!x  # 'Non' logique

## [1] FALSE FALSE  TRUE  TRUE

x&y     # 'Et' logique

## [1]  TRUE FALSE FALSE FALSE

x&&y    # 'Et' logique (revient à faire x[1] & y[1])

## [1] TRUE

x|y     # 'Ou' logique

## [1]  TRUE  TRUE  TRUE FALSE

x||y    # 'Ou' logique (revient à faire x[1] | y[1])

## [1] TRUE

xor(x, y)       # 'Ou' exlusif

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

Remarque 2.4.1

Il existe deux fonctions importantes pour les vecteurs de type `logical` : `any()` et `all()` qui indiquent respectivement, si au moins un des éléments du vecteur vaut `TRUE` et si tous les éléments du vecteur valent `TRUE`.

```
x <- c(TRUE, FALSE)
any(x)

## [1] TRUE

y <- c(FALSE, FALSE)
any(y)

## [1] FALSE

all(!y)

## [1] TRUE
```

2.4.1.4 Attention au recyclage

Jusqu'ici, les exemples fournis au sujet des opérateurs ne concernaient que des objets de même longueur. Dans le cas des **vecteurs**, si un des deux objets concernés par l'opération est de taille inférieure à celle de l'autre, `R` effectue un recyclage, c'est à dire qu'il va compléter le vecteur le

plus court à l'aide des valeurs de ce même vecteur, afin d'obtenir deux objets de même taille. On peut voir que R affiche un message d'avertissement lors d'une telle opération.

```
x <- c(1, 2, 3)
y <- c(1, 2, 3, 4, 5, 6, 7)
x + y

## Warning in x + y: la taille d'un objet plus long n'est pas multiple de la taille
d'un objet plus court

## [1] 2 4 6 5 7 9 8

# R a ajouté le vecteur c(1, 2, 3, 1, 2, 3, 1) à y
```

2.4.2 Accès aux valeurs, modifications

Pour accéder aux éléments d'un objet, R propose plusieurs méthodes, qui peuvent varier selon le type d'objet.

2.4.2.1 Accès par indices

Un moyen simple d'accéder aux éléments d'un objet est d'utiliser le système d'indexation. Cette indexation peut être de type numérique ou logique. Dans le cas d'un vecteur, on extrait une composante en utilisant la fonction `"["()`, dont les paramètres en dehors du vecteur pour lequel on souhaite réaliser une extraction peuvent être un vecteur d'indices d'éléments à extraire ou ne pas extraire, un vecteur d'éléments de type logique indiquant si l'élément doit être extrait ou non.

```
x <- c(4, 7, 3, 5, 0)
"["(x, 2)  # Extraire le second élément de x

## [1] 7

x[2]    # Une écriture plus commode pour extraire le second élément de x

## [1] 7

x[-2]   # Tous les éléments de x sauf le second

## [1] 4 3 5 0

x[3:4]  # Les troisième et quatrième éléments de x

## [1] 3 5

i <- 3:4
x[i] # On peut utiliser une variable contenant un vecteur d'indices

## [1] 3 5

x[c(F, T, F, F, F)]     # Le second élément de x

## [1] 7
```

```
x[x<1]  # Les éléments de x inférieurs à 1

## [1] 0

x<1     # Il s'agit bien d'un vecteur de logiques

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

Remarque 2.4.2

La fonction `which()` retourne les positions des éléments d'un vecteur logique pour lesquels la valeur vaut `TRUE`. Les fonctions `which.min()` et `which.max()` retournent respectivement la position du (premier) minimum et du premier maximum d'un vecteur numérique ou logique.

```
x <- c(2, 4, 5, 1, 7, 6)
which(x < 7 & x > 2)

## [1] 2 3 6

which.min(x)

## [1] 4

which.max(x)

## [1] 5

x[which.max(x)]

## [1] 7
```

Le remplacement d'un élément s'effectue simplement en utilisant la flèche d'assignation. On peut modifir plusieurs éléments à la fois.

```
x <- seq_len(5)
x[2] <- 3
x

## [1] 1 3 3 4 5

x[2] <- x[3] <- 0
x

## [1] 1 0 0 4 5

x[which(x == 0)] <- 10
x

## [1]  1 10 10  4  5
```

Pour les **matrices** ou les **tableaux de données**, l'extraction par indice se fait dans le même esprit, mais il faut indiquer un vecteur d'indices (`i`) pour les lignes et un pour les colonnes (`j`),

de la manière suivante : `x[i, j]`, avec `x` la matrice ou le tableau de donnée. En omettant le vecteur d'indices pour les lignes ou les colonnes, `R` retourne toutes les lignes ou les colonnes respectivement. Enfin, en ajoutant le symbole "moins" (`-`) devant le vecteur d'indices de lignes ou de colonnes, on demande à `R` de ne pas retourner les éléments dont les indices sont mentionnés.

```
(x <- matrix(1:9, ncol = 3, nrow = 3))

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

x[1, 2]  # Élément de la ligne 1 et de la colonne 2

## [1] 4

i <- c(1,3) ; j <- 3
x[i,j]  # Éléments des lignes 1 et 3 de la troisième colonne

## [1] 7 9

x[, 2]  # Éléments de la seconde colonne

## [1] 4 5 6

x[1 ,]  # Éléments de la première ligne

## [1] 1 4 7

x[, -c(1,3)]    # x sans les colonnes 1 et 3

## [1] 4 5 6
```

Avec le dernier exemple, on voit que `R` retourne un vecteur et non une matrice. Si on souhaite obtenir une matrice à une seule colonne, il suffit d'ajouter l'argument `drop` auquel on donne la valeur `FALSE`.

```
x[, -c(1,3), drop = FALSE]

##      [,1]
## [1,]    4
## [2,]    5
## [3,]    6
```

Pour modifier une valeur, c'est à nouveau avec la flèche d'assignation.

```
x <- matrix(1:9, ncol = 3, nrow = 3)
x[1,2] <- 0
x

##      [,1] [,2] [,3]
## [1,]    1    0    7
```

```
## [2,]    2    5    8
## [3,]    3    6    9
```

Remarque 2.4.3

En utilisant `which()` sur une matrice, on peut demander à `R` de retourner les indices sous forme de couples :

```
which(x > 5, arr.ind = TRUE)

##      row col
## [1,]   3   2
## [2,]   1   3
## [3,]   2   3
## [4,]   3   3
```

Il est également possible d'utiliser des vecteurs logiques.

```
(x_logique <- matrix(c(TRUE, FALSE), ncol = 3, nrow = 3))

## Warning in matrix(c(TRUE, FALSE), ncol = 3, nrow = 3): la longueur des données
[2] n'est pas un diviseur ni un multiple du nombre de lignes [3]

##       [,1]  [,2]  [,3]
## [1,]  TRUE FALSE  TRUE
## [2,] FALSE  TRUE FALSE
## [3,]  TRUE FALSE  TRUE

x[x_logique]

## [1] 1 3 5 7 9
```

Comme indiqué dans la section 2.2.1.4, les matrices et les tableaux sont des vecteurs, qui correspondent à un empilement des vecteurs colonnes. Ainsi, on peut extraire les éléments de la matrice à l'aide d'un vecteur d'indices ou de valeurs logiques.

```
x[c(1,3,7)]

## [1] 1 3 7
```

Pour les **tableaux de dimension supérieure à 2**, il suffit juste d'ajouter un paramètre par dimension supplémentaire. Voici un exemple avec un tableau de dimension 3 :

```
z <-array(1:24, dim =c(2, 4, 3))
z[2,4,3]  # Élément de la ligne 2, colonne 4 de la dimension 3

## [1] 24
```

Enfin, pour les **listes**, l'emploi de la fonction `"["()` retourne une liste, puisque les listes sont constituées de listes.

```
personne <- list("Piketty", "Thomas", "1971")
personne[1]

## [[1]]
## [1] "Piketty"

class(personne[1])

## [1] "list"

personne[c(1,3)]

## [[1]]
## [1] "Piketty"
##
## [[2]]
## [1] "1971"
```

Pour obtenir le contenu d'un ou plusieurs éléments de la liste, on fait donc appel à une autre fonction : `"[["()`. Comme pour la fonction avec les crochets simples, on peut l'employer d'une manière plus commode en l'accolant au nom de la liste contenant les éléments que l'on souhaite extraire.

```
"[["(personne, 1)  # Premier élément de la liste

## [1] "Piketty"

personne[[1]]   # idem

## [1] "Piketty"

personne[[1,2]] # Ne fonctionne pas

## Error in personne[[1, 2]]: nombre d'indices incorrect

personne[[c(1,2)]]      # Ne fonctionne pas non plus ici

## Error in personne[[c(1, 2)]]: indice hors limites

personne[1,2] # Ne fonctionne pas

## Error in personne[1, 2]: nombre de dimensions incorrect

personne[c(1,2)]        # Retourne une liste de dim 2 contenant :

## [[1]]
## [1] "Piketty"
##
## [[2]]
## [1] "Thomas"

        # - le premier élément de la liste
        # - le second élément de la liste
```

Grâce à la propriété d'indexation récursive, il est possible d'accéder aux éléments des objets de la liste. Par exemple, si le troisième élément de la liste `l` est un vecteur, et que l'on souhaite récupérer son premier élément, on peut écrire procéder de deux manières :

```
(l <- list("foo_1", "foo_2", c("foo_3", "foo_4")))

## [[1]]
## [1] "foo_1"
##
## [[2]]
## [1] "foo_2"
##
## [[3]]
## [1] "foo_3" "foo_4"

l[[3]]  # Troisième élément de la liste

## [1] "foo_3" "foo_4"

l[[c(3,1)]]     # Premier élément du troisième élément

## [1] "foo_3"

l[[3]][1]       # Premier élément du troisième élément

## [1] "foo_3"
```

Pour modifier l'élément d'une liste ou son contenu, on utilise encore la flèche d'assignation. Attention, ceci peut être destructif. En effet, si on indique à `R` que l'on souhaite modifier le contenu d'un élément, en utilisant la fonction `"[["()`, le résultat sera le même qu'en utilisant la fonction `"["()`.

```
l <- list(1, TRUE, "foo", list(matrix(1:4, ncol = 2), "foo_2"))
l[[4]] <- 2
l  # La liste en position 4 a été remplacée par un vecteur de longueur 1

## [[1]]
## [1] 1
##
## [[2]]
## [1] TRUE
##
## [[3]]
## [1] "foo"
##
## [[4]]
## [1] 2
```

2.4.2.2 Accès par noms

Comme précisé dans la section 2.2, les éléments d'un **vecteur** peuvent posséder un nom. Il est alors possible d'accéder à un élément par son nom.

```
personne <- c(nom = "Piketty", prenom = "Thomas", `annee de naissance` = "1971")
personne["nom"]

##       nom
## "Piketty"

names(personne)  # Accès aux noms des éléments du vecteur

## [1] "nom"                "prenom"             "annee de naissance"

names(personne) <- c("nom", "prenom", "naissance")  # Modification des noms
personne

##       nom    prenom naissance
## "Piketty"  "Thomas"    "1971"

names(personne) <- NULL  # Suppression des noms
personne

## [1] "Piketty" "Thomas"  "1971"
```

Si l'objet est une **liste** ou un **tableau de données**, on peut aussi utiliser le symbole dollar (`$`).

```
personne_liste <- list('nom de famille' = "Piketty", prenom = "Thomas",
    annee = 1971)
personne_liste[["nom de famille"]]

## [1] "Piketty"

personne_liste$"nom de famille"  # Le nom contenant au moins une espace,

## [1] "Piketty"

        # Il est nécessaire d'employer des guillemets
personne_liste$prenom

## [1] "Thomas"

names(personne_liste)  # Accès aux noms des éléments de la liste

## [1] "nom de famille" "prenom"         "annee"
```

Dans le cas des **matrices** et des **tableaux de données**, il est possible de donner un nom aux lignes et aux colonnes, avec les fonctions `rownames()` et `colnames()` respectivement. La fonction `dimnames()` retourne une liste dont le premier élément est une liste contenant le vecteur des noms de lignes, et le second élément une liste contenant le vecteur des noms de colonnes.

```
femmes <- data.frame(height = c(58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
    69, 70, 71, 72), weight = c(115, 117, 120, 123, 126, 129, 132, 135, 139,
    142, 146, 150, 154, 159, 164))
colnames(femmes)
```

```
## [1] "height" "weight"

rownames(femmes)

##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15"

dimnames(femmes)

## [[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15"
##
## [[2]]
## [1] "height" "weight"
```

Remarque 2.4.4

Lorsqu'on extrait des éléments d'un objet à l'aide des noms, les attributs de l'objet d'origine ne sont pas altérés.

Pour remplacer des valeurs, on utilise encore la flèche d'assignation. `R` produit un message d'erreur si l'élément de remplacement n'est pas de la bonne longueur pour la *data frame*.

```
df <- data.frame(x = seq_len(3), y = rep(2,3), z = c(1,4,3))
df$x <- rep(NA, 3)
df

##    x y z
## 1 NA 2 1
## 2 NA 2 4
## 3 NA 2 3

df$x <- rep(1, 4)

## Error in `$<-.data.frame`(`*tmp*`, "x", value = c(1, 1, 1, 1)): le tableau de
remplacement a 4 lignes, le tableau remplacé en a 3

df  # La modification n'a pas été faite

##    x y z
## 1 NA 2 1
## 2 NA 2 4
## 3 NA 2 3

# Attention
is.list(df$x)

## [1] FALSE

df$x <- list(4,3,2)
df$x    # La colonne "x" a changé de structure !
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 2
```

2.4.3 Chaînes de caractères

Savoir manipuler les chaînes de caractères permet de gagner énormément de temps dans le traitement de données texte, mais permet également de produire des sorties, graphiques ou non, plus lisibles et compréhensibles que les affichages par défaut proposés par `R`. Le lecteur intéressé par la manipulation avancée des chaînes de caractères en `R` est invité à consulter l'*ebook* de Gaston Sanchez (Sanchez, 2013).

2.4.3.1 Concaténation

La fonction `cat()` concatène et affiche à l'écran des objets. Les paramètres qui sont fournis sont convertis en vecteurs de chaînes de caractères, qui sont ensuite concaténés en un seul vecteur de caractères. Les éléments de ce vecteur sont ensuite joints entre eux, et éventuellement séparés par un caractère différent de l'espace, si le paramètre `sep` est modifié.

```
cat("Hello", "World", "!")

## Hello World !

# En ajoutant un passage à la ligne entre chaque élément
cat("Hello", "World", "!", sep = "\n")

## Hello
## World
## !

cat(matrix(1:6))

## 1 2 3 4 5 6
```

Remarque 2.4.5

Le caractère `\\` (barre oblique inversée, ou *backslash*) est le caractère d'échappement en `R`. Il permet d'afficher certains caractères, comme les guillemets dans une chaîne elle-même définie à l'aide de guillemets, ou bien les caractères de contrôle, comme la tabulation, le saut de ligne, etc. Le fichier d'aide `?Quotes` de `R` fournir de plus amples détails. Le tableau ci-arpès fournit quelques exemple courants.

`\n`	Nouvelle ligne	`\r`	Retour à la ligne
`\t`	Tabulation	`\b`	Retour arrière
`\\`	Barre olbique inversée	`\'`	Apostrophe
`\"`	Apostrophe double	`\‘`	Accent grave

La fonction `str_c()` du *package* `stringr` convertit en chaîne de caractères les éléments passés en paramètre puis les concatène. Il est tout à fait possible de stocker le résultat dans un objet, dans la mesure où le résultat de l'évaluation de `str_c()`, contrairement à celui de l'évaluation de `cat()` est une chaîne de caractères. Le paramètre `sep` permet de définir la chaîne de caractères séparant les termes (rien par défaut).
Il est nécessaire de charger le *package* avant d'appeler les fonctions qu'il contient (et de l'installer lors de la première utilisation).

```
library(stringr)
x <- str_c("Hello", "World", "!", sep = " ")
x

## [1] "Hello World !"
```

La fonction `str_c()` dispose aussi du paramètre `collapse`, qui est utile pour joindre les éléments d'un vecteur dans une même chaîne de caractères.

```
str_c(c("Rennes", "Bretagne"))

## [1] "Rennes"   "Bretagne"

# Il n'y a qu'un paramètre, donc le séparateur est inutile !
str_c(c("Rennes", "Bretagne"), sep = ", ")

## [1] "Rennes"   "Bretagne"

# En ajoutant le paramètre collapse
str_c(c("Rennes", "Bretagne"), collapse = ", ")

## [1] "Rennes, Bretagne"
```

Remarque 2.4.6

La fonction `str_c()` s'appuie en fait sur la fonction `paste()` du *package* `base`. Par défaut, la chaîne de séparation utilisée dans `paste()` est une espace. La fonction `paste0()` permet quant à elle de concaténer les chaînes sans séparateur. Il existe cependant de légères différences entre les deux fonctions, comme la festion des éléments `NULL` et `character(0)` :

```
paste("You", "shall", character(0), "not", NULL, "pass")

## [1] "You shall  not  pass"

str_c("You", "shall", character(0), "not", NULL, "pass", sep = " ")

## [1] "You shall not pass"
```

Si les objets donnés en paramètre à la fonction `paste()` sont de tailles différentes, un recyclage est effectué par `R`.

```r
str_c("Hello", c("Julien-Yacine", "Sonia", "Victor"))

## [1] "HelloJulien-Yacine" "HelloSonia"         "HelloVictor"
```

Attention, `R` convertit les valeurs manquantes (`NA`) en chaînes de caractères `"NA"`.

```r
str_c("Hello", NA)

## [1] NA
```

Remarque 2.4.7

Quand on souhaite accéder à une variable en renseignant son nom en utilisant la fonction `str_c()`, il faut faire appel à la fonction `get()`.

```r
variable_1 <- 5
# Affiche la chaîne "variable_1"
str_c("variable_", 1)

## [1] "variable_1"

# Affiche le contenu de la variable nommée "variable_1"
get(str_c("variable_", 1))

## [1] 5
```

2.4.3.2 Conversion en majuscules ou minuscules

Afin de passer une chaîne de caractères en majuscules ou en minuscules, `R` propose les fonctions `toupper()` et `tolower()`. La fonction `casefold()` permet de faire l'un ou l'autre, en précisant la valeur du paramètre `upper`.

```r
x <- "Bonjour !"
toupper(x)

## [1] "BONJOUR !"

tolower(x)

## [1] "bonjour !"

casefold(x)

## [1] "bonjour !"

casefold(x, upper = TRUE)

## [1] "BONJOUR !"
```

2.4.3.3 Compter le nombre de caractères d'une chaîne

La fonction `str_length()` du *package* `stringr` indique le nombre de caractères contenus dans une chaîne.

```r
str_length("Bonjour")

## [1] 7
```

Remarque 2.4.8

La fonction `str_length()` s'appuie sur la fonction `nchar()` du *package* `base`.
On peut noter que la longueur de `NA` vaut `NA` avec `str_length()` et `2` avec `nchar()` :

```r
texte <- c("Mais enfin", "c'est quoi", "un gigowatt ?", NA)
str_length(string = texte)

## [1] 10 10 13 NA

nchar(x = texte)

## [1] 10 10 13  2
```

2.4.3.4 Extraction de sous-chaînes

Pour extraire une sous-chaîne, on peut utiliser la fonction `str_sub()` du *package* `stringr`, qui prend en paramètres une chaîne de caractères, la position du début et celle de la fin de l'élément à extraire. Attention, l'indice du premier élément, tout comme pour les vecteurs, est 1 en `R`.

```r
x <- "Debt is one person's liability, but another person's asset."
str_sub(x, 1, 4)

## [1] "Debt"
```

Il est possible d'utiliser la flèche d'assignation après la fonction `str_sub()` pour remplacer la sous-chaîne extraite par une autre. Si la chaîne de remplacement est trop longue, elle n'est pas tronquée pour avoir la même longueur que celle qui est extraite.

```r
str_sub(x, 1, 4) <- "Remplacement"
x

## [1] "Remplacement is one person's liability, but another person's asset."
```

Remarque 2.4.9

Le paramètre **string** de la fonction **str_sub()** peut être un vecteur. Dans ce cas, la fonction est appliquée à chaque élément du vecteur.

```
str_sub(c("Rouge", "Vert", "Bleu"), 2, 3)

## [1] "ou" "er" "le"
```

Lors du remplacement, R peut avoir recours au recyclage.

```
x <- c("Rouge", "Vert", "Bleu")
str_sub(x, 2, 3) <- c("!!", "@@")

## Warning in `stri_sub<-`(`*tmp*`, from = start, to = end, value =
c("!!", : longer object length is not a multiple of shorter object length

x

## [1] "R!!ge" "V@@t"  "B!!u"
```

En fournissant aux paramètres **start** et **end** des valeurs négatives, on indique à Rde lire la chaîne à l'envers :

```
texte <- "le train de tes injures roule sur le rail de mon indifférence"
str_sub(string = texte, start = 4, end = 8)

## [1] "train"

# Compter à partir du dernier caractère
str_sub(string = texte, start = -12, end = -1)

## [1] "indifférence"

str_sub(string = texte, start = -16)

## [1] "mon indifférence"
```

Remarque 2.4.10

La fonction `str_sub()` s'appuie sur la fonction `substr()` du *package* `base`, mais propose quelques améliorations.
Dans le cas du remplacement d'une chaîne extraitre par une autre, on distingue trois cas :

- la chaîne de remplacement est de même longueur que celle extraite : `str_sub()` et `substr()` se comportent de la même manière ;
- la chaîne de remplacement est plus courte que celle extraite : avec `substr()`, la chaîne de remplacement est complétée par la fin de celle extraite, tandis qu'avec `str_sub()`, la chaîne extraite est retirée et remplacée par celle de remplacement ;
- la chaîne de remplacement est plus longue que celle extraite : avec `substr()`, la chaîne de remplacement est tronquée, tandis qu'elle ne l'est pas avec `str_sub()`.

```
texte <- "le train de tes injures roule sur le rail de mon indifférence"
# On copie le contenu de texte dans une nouvelle variable
texte_2 <- texte

# Remplacement plus court que la chaîne extraite
str_sub(string = texte, start = 17, end = 23)

## [1] "injures"

str_sub(string = texte, start = 17, end = 23) <- "jurons"
substr(x = texte_2, start = 17, stop = 23) <- "jurons"
texte ; texte_2

## [1] "le train de tes jurons roule sur le rail de mon indifférence"
## [1] "le train de tes juronss roule sur le rail de mon indifférence"

# Remplacement plus long que la chaîne extraite
str_sub(string = texte, start = 1, end = 8)

## [1] "le train"

str_sub(string = texte, start = 1, end = 8) <- "la locomotive"
substr(x = texte_2, start = 1, stop = 8) <- "la locomotive"
texte ; texte_2

## [1] "la locomotive de tes jurons roule sur le rail de mon indifférence"
## [1] "la locom de tes juronss roule sur le rail de mon indifférence"
```

2.4.3.5 Recherche de chaînes de caractères

Quand on souhaite trouver un motif (*pattern*) dans un vecteur de chaînes de caractères, on peut utiliser la fonction `str_detect()` du *package* `stringr`. Elle retourne les indices des éléments du vecteur dans lesquels le motif a été trouvé.

```
str_detect(string = c("Pomme", "Poire", "Ananas"), pattern = "o")

## [1]  TRUE  TRUE FALSE
```

Pour remplacer la première occurrence du motif trouvé par une autre chaîne (qui n'est pas tenue d'être de la même longueur de caractères), on peut utiliser la fonction `str_replace()`.

```
str_replace(string = c("Pomme", "Poire", "Ananas"), pattern = "a", replacement = "@@")

## [1] "Pomme"   "Poire"   "An@@nas"
```

Si on désire remplacer toutes les occurrences trouvées, il faut alors utiliser la fonction `str_replace_all()`.

```
str_replace_all(string = c("Pomme", "Poire", "Ananas"), pattern = "a", replacement = "@@")

## [1] "Pomme"    "Poire"    "An@@n@@s"
```

Si on souhaite ignorer la casse, il suffit de faire appel à la fonction `ignore.case()` comme suit :

```
str_detect(string = c("Obi-Wan Kenobi", "Darth Vader"), pattern = "w")

## [1] FALSE FALSE

str_detect(string = c("Obi-Wan Kenobi", "Darth Vader"), pattern = ignore.case("w"))

## Please use (fixed|coll|regexp)(x, ignore_case = TRUE) instead of ignore.case(x)

## [1]  TRUE FALSE
```

Pour briser une chaîne en fonction d'un motif, le *package* `stringr` propose la fonction `str_split()`.

```
x = "Criquette ! Vous, ici ? Dans votre propre salle de bain ? Quelle surprise !"
str_split(string = x, pattern = " ")

## [[1]]
##  [1] "Criquette" "!"         "Vous,"     "ici"       "?"
##  [6] "Dans"      "votre"     "propre"    "salle"     "de"
## [11] "bain"      "?"         "Quelle"    "surprise"  "!"
```

Une des rares fonctions du *package* `stringr` ne commençant pas par le préfixe `str` est la fonction `word()`. Comme son nom le laisse présager, elle permet d'extraire des mots dans une phrase. Les mots sont repérés par défaut par une espace, mais il est possible de préciser un séparateur différent avec le paramètre `sep`. Le paramètre `start` attend une valeur entière donnant la position du premier mot à extraire (celle du premier mot de la phrase, par défaut). Si la valeur est négative, le comptage s'effectue de la droite vers la gauche. Le paramètre `end` attent l'entier donnant la position du dernier mot à extraire (vaut la valeur de `start` s'il est omis). Encore une fois, si la valeur est négative, le comptage s'effectue de la droite vers la gauche.

```
phrase <- c("Mademoiselle Deray, il est interdit de manger de la choucroute ici.",
            "Oh si si, prenez un chewing-gum, Émile.")
word(phrase) # Extraction du premier mot

## [1] "Mademoiselle" "Oh"

word(phrase, 2) # Extraction du second mot
```

```
## [1] "Deray," "si"

word(phrase, -1) # Extraction du dernier mot

## [1] "ici."   "Émile."

word(phrase, 2, -1) # Extraction du second au dernier mot

## [1] "Deray, il est interdit de manger de la choucroute ici."
## [2] "si si, prenez un chewing-gum, Émile."

# Du premier au dernier mot, du second au dernier, et du troisième au dernier
# pour le premier élément de phrase
word(phrase[1], 1:3, -1)

## [1] "Mademoiselle Deray, il est interdit de manger de la choucroute ici."
## [2] "Deray, il est interdit de manger de la choucroute ici."
## [3] "il est interdit de manger de la choucroute ici."

# Premier mot, Premier et second mot, Premier et troisième mot
# pour le second élément de phrase
word(phrase[2], 1, 1:3)

## [1] "Oh"        "Oh si"     "Oh si si,"
```

Remarque 2.4.11

La fonction `str_detect()` s'appuie sur la fonction `grepl()` du *package* `base` ; les fonctions `str_replace()` et `str_replace_all()` sur la fonction `sub()` du *package* `base`. La fonction `str_split()` correspond à la fonction `strsplit()` du *package* `base`.

Des recherches bien plus complèxes et puissantes peuvent être effectuées avec `R`, et sont présentées dans la section 7.

2.4.3.6 Nettoyage, complétion

La fonction `str_dup()`

La fonction `str_dup()` du *package* `stringr` permet de dupliquer du texte. Elle est vectorisée à la fois pour le vecteur de texte à fournir au paramètre `string` et au vecteur d'entiers à fournir au paramètre `times`

```
texte <- c("bla", "ah", "eh")
str_dup(texte, 2)

## [1] "blabla" "ahah"   "eheh"

str_dup(texte, 1:3)

## [1] "bla"    "ahah"   "eheheh"
```

La fonction `str_pad()`

Certains fichiers de données ont des largeurs fixes pour chaque colonne. Si on désire créer facilement ce genre de fichiers, la fonction `str_pad()` du *package* `stringr` peut se révéler très pratique. En effet, elle permet de définir un caractère à insérer avant ou après une chaîne, ou bien avant et après, et de le répéter un nombre donné de fois. On précise la longueur des chaînes retournées avec le paramètre `width`, le caractère à insérer pour éventuellement compléter la chaîne avec le paramètre `pad` et le côté sur lequel insérer ce caractère avec le paramètre `side` pouvant prendre les valeurs `"left"`, `"right"` ou `"both"`.

```
coords <- c(lat = "48.11", long = "-1.6794")
str_pad(string = coords, width = 7, side = "left", pad = " ")

## [1] "  48.11" "-1.6794"

# Les chaînes plus longues que la valeur fournie à width sont inchangées
str_pad(c("Gauthier", "Pascaline"), 3)

## [1] "Gauthier"  "Pascaline"
```

La fonction `str_trim()`

Pour retirer des caractères blancs (e.g. espaces, sauts de ligne, retours à la ligne, quadratins, etc.), on peut utiliser des méthodes abordées dans la section 2.4.3.5, ou bien la fonction `str_trim()` du *package* `stringr`. Elle retire tous les caractères blancs à gauche et à droite d'une chaîne de caractères. Le paramètre `side` permet de choisir le côté pour lequel on désire retirer les caractères blancs. Il peut prendre les valeurs `both` (par défaut), `left` ou `right`.

```
texte <- c("\n\nPardon, du sucre ?", "Oui, seize   \n ", "...\t\t...\t")
str_trim(texte, side = "both")

## [1] "Pardon, du sucre ?" "Oui, seize"         "...\t\t..."
```

2.4.4 Dates

2.4.4.1 Extraction

Grâce à la structure en liste pour stocker les éléments d'une date, il est aisé d'extraire ces derniers ! Cependant, récupérer le nombre d'années depuis 1900 n'est pas forcément très pratique...

```
d <- as.POSIXlt("2015-10-21 13:55:44")

# Récupérer l'heure
d$hour

## [1] 13

# Le nombre d'années depuis 1900
d$year

## [1] 115
```

Le *package* `lubridate` propose quelques fonctions pour accéder aux éléments d'une date.

- `second()` : secondes (0–59) ;
- `minute()` : minutes (0–59) ;
- `hour()` : heures (0–23) ;
- `day()`, `mday()` : jour du mois (1–31) ;
- `wday()` : jour de la semaine (1–7), le dimanche étant le 1 ;
- `yday()` : jour de l'année (1–366) ;
- `week()` : numéro de la semaine dans l'année. Les semaines sont définies comme les périodes complètes de 7 jours s'étant déroulées depuis le premier janvier, plus 1 ;
- `isoweek()` : Semaine de l'année (00-53). Si la semaine (qui commence un lundi) qui contient le 1er janvier a quatre jours ou plus dans la nouvelle année, alors elle est considérée comme la semaine 1. Sinon, elle est considérée comme la dernière de l'année précédente, et la semaine suivante est considérée comme semaine 1 (norme ISO 8601) ;
- `month()` : mois (1–12) ;
- `year()` : année, uniquement celles après l'an 1 ;
- `tz()` : fuseau horaire ;

Les fonctions `wday()` et `month()` permettent aussi d'afficher le jour de la semaine et le mois, respectivement, en chaîne de caractères, en donnant leur nom anglais. Pour cela, il faut attribuer la valeur `TRUE` au paramètre `label`. Si on souhaite les abréviations, il faut donner la valeur `TRUE` au paramètre `abbr` en sus.

```
library(lubridate)
(d <- ymd_hms("2015-10-30 23:59:59"))

## [1] "2015-10-30 23:59:59 UTC"

lubridate::wday(d, label = TRUE)

## [1] Fri
## Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat

lubridate::wday(d, label = TRUE, abbr = TRUE)

## [1] Fri
## Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat

lubridate::month(d, label = TRUE)

## [1] Oct
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec

lubridate::month(d, label = TRUE, abbr = TRUE)

## [1] Oct
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

2.4.4.2 Opérations

Avant de présenter quelques exemples, créons quelques variables de dates.

```
# Quelques dates
d_date_1 <- as.Date("2014-09-01")
d_date_2 <- as.Date("2015-10-21")
```

```
d_posix_ct_1 <- as.POSIXct("2014-09-01 18:32:28")
d_posix_ct_2 <- as.POSIXct("2015-10-21 13:55:44")

d_posix_lt_1 <- as.POSIXlt("2014-09-01 18:32:28")
d_posix_lt_2 <- as.POSIXlt("2015-10-21 13:55:44")
```

Pour ajouter ou retirer des durées de temps, il faut se rappeler comment sont stockées les dates. Avec les objets de mode `date`, il s'agit d'un nombre de jours. Aussi, additioner un nombre n à un objet de type `date` retourne la date n jours plus tard. Pour les objets de classe `POSIXct` ou `POSIXlt`, comme `R` stock la date en secondes, l'ajout d'un nombre n retourne la date augmentée de n secondes.

```
d_date_2

## [1] "2015-10-21"

d_date_2 + 10

## [1] "2015-10-31"

d_date_2 - 10

## [1] "2015-10-11"

d_posix_ct_2

## [1] "2015-10-21 13:55:44 CEST"

d_posix_ct_2 + 10

## [1] "2015-10-21 13:55:54 CEST"

d_posix_ct_2 - 10

## [1] "2015-10-21 13:55:34 CEST"
```

Toujours dû à ce stockage interne en jours ou secondes, il est possible de comparer facilement deux dates entre-elles, avec les opérateurs classiques.

```
d_date_1 > d_date_2

## [1] FALSE

d_posix_ct_2 > d_posix_ct_1

## [1] TRUE

d_posix_lt_2 == d_posix_lt_1

## [1] FALSE
```

Pour obtenir la différence entre deux dates, on peut utiliser l'opérateur `-`. Dans les cas des `POSIXlt` et `POSIXlt`, il est également possible d'utiliser la fonction `difftime()`, en indiquant les

deux dates en paramètres, et éventuellement en renseignant le paramètre `units` pour indiquer l'unité désirée pour la différence. Le tableau ci-après répertorie les valeurs possibles pour le paramètre `units`.

Code	Description
`auto`	le choix de l'unité pour que celle-ci soit la plus large possible (à l'exception de `weeks`)
`secs`	secondes
`mins`	minutes
`hours`	heures
`days`	jours (périodes de 24 heures)
`weeks`	semaines

```
# Avec des dates
d_date_2 - d_date_1

## Time difference of 415 days

d_posix_ct_2 - d_posix_ct_1

## Time difference of 414.8078 days

# Avec des POSIXct
difftime(d_posix_ct_2, d_posix_ct_1, units = "hours")

## Time difference of 9955.388 hours

round(difftime(d_posix_ct_2, d_posix_ct_1, units = "hours"))

## Time difference of 9955 hours

# Avec des POSIXlt
d_posix_lt_2 - d_posix_lt_1

## Time difference of 414.8078 days

difftime(d_posix_lt_2, d_posix_lt_1, units = "week")

## Time difference of 59.25826 weeks
```

On peut noter qu'il est possible d'utiliser les méthodes du groupe `Math` (`round()`, `signif()`, `floor()`, `ceiling()`, `trunc()`, `abs()` et `sign()`, etc.)

```
as.numeric(round(difftime(d_posix_ct_2, d_posix_ct_1, units = "hours")))

## [1] 9955
```

2.4.4.3 Intervalles de dates

La fonction `new_interval()` du *package* `lubridate` crée un intervalle de temps entre deux dates. On peut également utiliser l'opérateur `\%-\%`.

```
debut <- ymd_hms("2014-10-01 07:00:00", tz = "Australia/Perth")
fin <- ymd_hms("2014-10-01 23:00:00", tz = "Australia/Perth")

(intervalle_1 <- interval(debut, fin))

## [1] 2014-10-01 07:00:00 AWST--2014-10-01 23:00:00 AWST

debut %--% fin

## [1] 2014-10-01 07:00:00 AWST--2014-10-01 23:00:00 AWST
```

La durée d'un intervalle s'obtient avec la fonction `int_length()`. Le résultat est exprimé en secondes.

```
int_length(intervalle_1)

## [1] 57600
```

On peut chercher si deux intervalles se chevauchent, avec la fonction `int_overlaps()`.

```
debut_2 <- ymd_hms("2014-10-01 08:00:00", tz = "Australia/Perth")
fin_2 <- ymd_hms("2014-10-01 10:00:00", tz = "Australia/Perth")

debut_3 <- ymd_hms("2014-10-02 09:00:00", tz = "Australia/Perth")
fin_3 <- ymd_hms("2014-10-02 10:00:00", tz = "Australia/Perth")

(intervalle_2 <- new_interval(debut_2, fin_2))

## [1] 2014-10-01 08:00:00 AWST--2014-10-01 10:00:00 AWST

(intervalle_3 <- new_interval(debut_3, fin_3))

## [1] 2014-10-02 09:00:00 AWST--2014-10-02 10:00:00 AWST

int_overlaps(intervalle_1, intervalle_2)

## [1] TRUE

int_overlaps(intervalle_1, intervalle_3)

## [1] FALSE
```

On peut utiliser les fonctions ensemblistes sur des intervalles (`union()` (union), `intersect()` (intersection), `setdiff()` (différence), `setequal()` (égalité) et `\%within\%` (tester si une date est dans un intervalle)).

```
union(intervalle_1, intervalle_2)

## [1] 2014-10-01 07:00:00 AWST--2014-10-01 23:00:00 AWST
```

```
intersect(intervalle_1, intervalle_3)

## [1] NA--NA

setdiff(intervalle_1, intervalle_2)

## [1] 2014-10-01 10:00:00 AWST--2014-10-01 08:00:00 AWST

setequal(intervalle_1, intervalle_1)

## [1] TRUE

debut_2 %within% intervalle_1

## [1] TRUE
```

Après avoir utilisé ces fonctions ensemblistes, on peut être amené à vouloir extraire les bornes de l'intervalle. Il faut alors utiliser les fonctions `int_start()` et `int_end()` pour les bornes inférieures et supérieures respectivement.

```
(intervalle <- setdiff(intervalle_1, intervalle_2))

## [1] 2014-10-01 10:00:00 AWST--2014-10-01 08:00:00 AWST

int_start(intervalle)

## [1] "2014-10-01 10:00:00 AWST"

int_end(intervalle)

## [1] "2014-10-01 08:00:00 AWST"
```

La fonction `int_flip()` échange les bornes de l'intervalle. Le nouvel intervalle possède la même longueur, mais une direction opposée.

```
int_flip(intervalle_1)

## [1] 2014-10-01 23:00:00 AWST--2014-10-01 07:00:00 AWST
```

La fonction `int_shift()` décale le début et la fin de l'intervalle, en fonction de la valeur fournie au paramètre `by`, qui doit être un objet de durée, que l'on peut obtenir avec la fonction `new_duration()`.

```
int_shift(intervalle_1, new_duration(days = 5))

## [1] 2014-10-06 07:00:00 AWST--2014-10-06 23:00:00 AWST
```

La fonction `int_aligns()` permet de tester si deux intervalles ont une borne commune, sans prendre en compte la direction de chaque intervalle. Ainsi, la fonction teste si les moments les plus anciens ou plus récents de chaque intervallent arrivent au même moment.

```
int_1 <- new_interval(ymd("2014-10-01"), ymd("2014-10-05"))
int_2 <- new_interval(ymd("2014-10-04"), ymd("2014-10-01"))
int_3 <- new_interval(ymd("2015-10-01"), ymd("2015-10-05"))

int_aligns(int_1, int_2)

## [1] TRUE

int_aligns(int_1, int_3)

## [1] FALSE
```

Les intervalles sont des laps de temps spécifiques. Le *package* `lubridate` fournit également deux classes de laps de temps : `duration` (durées) et `period` (époques). Les fonctions qui créèent les durées possèdent le nom au pluriel de ces durées précédé du préfixe `d`; les fonctions qui créent les époques laissent quant à elle tomber le préxixe.

```
minutes(10)

## [1] "10M 0S"

dminutes(10)

## [1] "600s (~10 minutes)"
```

Les `duration` ne tiennent pas compte des fluctuations de la ligne du temps, comme les années bissextiles, alors que les `period`, oui. L'exemple qui suit utilise la fonction `leap_year()`, qui retourne un logique indiquant si l'année passée en paramètre est bissextile.

```
leap_year(2000)

## [1] TRUE

# Ajout d'une durée d'un an, soit 365 jours
ymd("2000-01-01") + dyears(1)

## [1] "2000-12-31 UTC"

# Ajout d'une période d'un an, soit 366 jours dans ce cas
ymd("2000-01-01") + years(1)

## [1] "2001-01-01 UTC"
```

On peut obtenir la durée d'un intervalle exprimée en jours, en mois, en semaines, etc. en le divisant par une durée.

```
intervalle <- new_interval(ymd_hms("2014-10-01 10:00:00"),
                           ymd_hms("2014-10-06 08:00:00"))

# Nombre de jours
intervalle / ddays(1)

## [1] 4.916667
```

```
# Nombre de durées de 2 jours
intervalle / ddays(2)

## [1] 2.458333

# Nombre d'heures
intervalle / dhours(1)

## [1] 118
```

Enfin, on peut effectuer une division entière ou extraire le reste de cette division (modulo).

```
intervalle %% days(1)

## [1] 2014-10-05 10:00:00 UTC--2014-10-06 08:00:00 UTC

as.period(intervalle %% days(1))

## [1] "22H 0M 0S"

intervalle %/% days(1)

## [1] 4

as.period(intervalle %/% days(1))

## [1] "4S"
```

2.4.4.4 Séquence de dates

Il est possible d'utiliser la fonction `seq()` avec des dates, qu'elles soient stockées au format `date`, `POSIXct` ou `POSIXlt`. Le lecteur s'asurera de bien comprendre les exemples suivants.

```
# Avec des objets de class date
seq(as.Date("2014-09-01"), length = 4, by = "day")

## [1] "2014-09-01" "2014-09-02" "2014-09-03" "2014-09-04"

# Tous les deux jours
seq(as.Date("2014-09-01"), length = 4, by = "2 day")

## [1] "2014-09-01" "2014-09-03" "2014-09-05" "2014-09-07"

# En spécifiant le début et la date maximum
seq(as.Date("2014-09-01"), as.Date("2014-09-08") , by = "2 day")

## [1] "2014-09-01" "2014-09-03" "2014-09-05" "2014-09-07"

# Avec des objets de classe POSIXct et POSIXlt
seq(d_posix_ct_1, by = "9 month", length = 2)

## [1] "2014-09-01 18:32:28 CEST" "2015-06-01 18:32:28 CEST"

seq(d_posix_lt_1, by = "9 month", length = 2)

## [1] "2014-09-01 18:32:28 CEST" "2015-06-01 18:32:28 CEST"
```

2.4.4.5 Fuseaux horaires

Il peut être utile de faire des conversions d'heures, pour obtenir la valeur dans un autre fuseau horaire. La fonction `with_tz()` le permet.

```
# Une heure à Paris
(d <- ymd_hms("2014-10-30 20:00:00", tz = "Europe/Paris"))

## [1] "2014-10-30 20:00:00 CET"

# L'heure équivalente à New York City
with_tz(d, "America/New_York")

## [1] "2014-10-30 15:00:00 EDT"
```

La fonction `force_tz()` permet de remplacer le fuseau horaire d'une date.

```
(d <- ymd_hms("2014-10-30 20:00:00", tz = "Europe/Paris"))

## [1] "2014-10-30 20:00:00 CET"

force_tz(d, "America/New_York")

## [1] "2014-10-30 20:00:00 EDT"
```

2.4.4.6 Paramètres locaux

Il est fréquent d'avoir des dates où le jour de la semaine, ou le mois de l'année est indiqué en anglais. Si on travaille sur une machine avec un système d'exploitation en français, cela pose problème. Heureusement, à chaque problème sa solution.

```
d_char_fr <- "Mer 04 Fév 2015"
d_char <- "Wed 04 Feb 2015"

as.Date(d_char_fr, format = c("%a %d %b %Y"))

## [1] "2015-02-04"

as.Date(d_char, format = c("%a %d %b %Y"))

## [1] NA
```

Comme on peut le voir, `R` cherche à réaliser un *matching* entre ce qu'on lui indique être le jour de la semaine abrégé (`%a`) et les jours retournés par l'exécution de l'instruction suivante : `weekdays(Sys.Date()+0:6)`. Il en va de même avec les abréviations des mois (telles que contenues dans la variable réservée `month.abb`[3]. Or, que ce soit pour les jours de la semaine ou pour les mois de l'année, leur valeur dépend des paramètres locaux de la machine. Il faut alors indiquer à `R` un changement des paramètres locaux pour que la conversion réussisse.

```
# Les paramètres locaux pour les unités de temps
(old <- Sys.getlocale("LC_TIME"))
```

3. La variable `month.name` contient quant à elle le nom complet des mois

```
## [1] "fr_FR.UTF-8"

# Indiquer que les dates doivent être au format américain
# Sous Windows
# Sys.setlocale('LC_TIME','english_us')
# Sous Unix
Sys.setlocale('LC_TIME','en_us')

## [1] "en_us"

as.Date(d_char, format = c("%a %d %b %Y"))

## [1] "2015-02-04"

as.Date(d_char_fr, format = c("%a %d %b %Y"))

## [1] NA

# Retourner aux paramètres précédents
Sys.setlocale('LC_TIME', old)

## [1] "fr_FR.UTF-8"

as.Date(d_char, format = c("%a %d %b %Y"))

## [1] NA

as.Date(d_char_fr, format = c("%a %d %b %Y"))

## [1] "2015-02-04"
```

Le *package* `lubridate` propose une manière plus simple de gérer ce type de soucis ! Lorsque les dates contiennent des noms de jours ou de mois, il suffit, avec les fonctions de conversion du *package* `lubridate`, de préciser le lieu au paramètre `locale` :

```
dmy("Wed 04 Feb 2015")

## Warning: All formats failed to parse. No formats found.

## [1] NA

# Sous Windows
# dmy("Wed 04 Feb 2015", locale = "english_us")
# Sous Unix
dmy("Wed 04 Feb 2015", locale = "en_us")

## [1] "2015-02-04 UTC"
```

Remarque 2.4.12

En fait, dans l'exemple précédent, le jour `Wed` a été ignoré. On peut s'en convaincre en tentant de remplacer `Wed` par autre chose :

```
# dmy("Wed 04 Feb 2015", locale = "english_us") # Windows
dmy("Toto 04 Feb 2015", locale = "en_us") # Unix

## [1] "2015-02-04 UTC"
```

2.4.5 Calculs matriciels

Cette section aborde les opérations effectuées sur des matrices.

2.4.5.1 Addition, soustraction

L'addition et la soustraction de deux matrices A et B de même type ou d'une matrice et d'un scalaire a se font avec les opérateurs `+` et `-` respectivement.

```
(A <- matrix(c(1, 3, 2, 2, 2, 1, 3, 1, 3), ncol = 3))

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    2    1
## [3,]    2    1    3

(B <- matrix(c(4, 6, 4, 5, 5, 6, 6, 4, 5), ncol = 3))

##      [,1] [,2] [,3]
## [1,]    4    5    6
## [2,]    6    5    4
## [3,]    4    6    5

(C <- matrix(c(0, 3, 1), ncol = 1))

##      [,1]
## [1,]    0
## [2,]    3
## [3,]    1

a <- 2
A+a

##      [,1] [,2] [,3]
## [1,]    3    4    5
## [2,]    5    4    3
## [3,]    4    3    5

A+B

##      [,1] [,2] [,3]
## [1,]    5    7    9
## [2,]    9    7    5
## [3,]    6    7    8
```

```
A-a
```

```
##      [,1] [,2] [,3]
## [1,]   -1    0    1
## [2,]    1    0   -1
## [3,]    0   -1    1
```

```
A-B
```

```
##      [,1] [,2] [,3]
## [1,]   -3   -3   -3
## [2,]   -3   -3   -3
## [3,]   -2   -5   -2
```

2.4.5.2 Multiplication, division

La multiplication et la division par un scalaire (aA et A/a) se font en utilisant les opérateurs `*` et `/` respectivement.

```
a*A
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    6    4    2
## [3,]    4    2    6
```

```
A/a
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  1.0  1.5
## [2,]  1.5  1.0  0.5
## [3,]  1.0  0.5  1.5
```

La transposée d'une matrice ($A^\top$) s'obtient grâce à la fonction `t()`.

```
t(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    2    2    1
## [3,]    3    1    3
```

La conjuguée d'une matrice ($\overline{A}$) s'obtient avec `Conj()`.

```
Conj(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    2    1
## [3,]    2    1    3
```

Pour multiplier deux matrices (AB), l'opérateur en `R` est `%*%`.

```
A%*%C
```

```
##      [,1]
## [1,]    9
## [2,]    7
## [3,]    6
```

Il ne faut pas confondre avec l'opérateur `*`, qui, lorsqu'il est employé entre deux matrices, effectue la multiplication terme à terme.

```
A*B
```

```
##      [,1] [,2] [,3]
## [1,]    4   10   18
## [2,]   18   10    4
## [3,]    8    6   15
```

L'inversion d'une matrice (A^{-1}) s'effectue avec la fonction `solve()`.

```
solve(A)
```

```
##             [,1]  [,2]       [,3]
## [1,] -0.41666667  0.25  0.3333333
## [2,]  0.58333333  0.25 -0.6666667
## [3,]  0.08333333 -0.25  0.3333333
```

Ce qui permet d'effectuer une division matricielle (BA^1).

```
B %/% solve(A)
```

```
##      [,1] [,2] [,3]
## [1,]  -10   19   18
## [2,]   10   20   -7
## [3,]   48  -25   15
```

La fonction `crossprod()` effectue le produit avec transposition ($A^\top B$).

```
crossprod(A,B)
```

```
##      [,1] [,2] [,3]
## [1,]   30   32   28
## [2,]   24   26   25
## [3,]   30   38   37
```

Elle réalise l'opération plus rapidement que l'instruction suivante :

```
t(A) %*% B
```

```
##      [,1] [,2] [,3]
## [1,]   30   32   28
## [2,]   24   26   25
## [3,]   30   38   37
```

2.4.5.3 Déterminant, trace

Le calcul du déterminant d'une matrice ($\det(A)$) s'obtient avec la fonction `det()`.

```
det(A)

## [1] -12
```

En utilisant la fonction `diag()`, qui donne les éléments de la diagonale de la matrice passée en paramètre, on peut facilement obtenir la trace de la matrice ($tr(A)$).

```
diag(A)

## [1] 1 2 3

sum(diag(A))

## [1] 6
```

2.4.6 Data frames

Cette section présente des méthodes pour manipuler les tableaux de données, les *data.frames*. Les descriptions des fonctions du *package* `dplyr` sont fortement inspirées de la vignette du *package*[4].

2.4.6.1 Sélection

Dans la section 2.4.2, il a été vu que l'accès aux données d'un *data.frame* peut se faire à l'aide du système d'indexation ou bien par celui de nommage, en utilisant la fonction `[[()`. Le *package* `dplyr` propose une autre manière de procéder pour sélectionner des colonnes, ou pour filtrer les données. Ces méthodes ont le mérite d'être faciles à lire. Reprenons le *data.frame* donné en exemple dans la section 2.4.2.

```
femmes <- data.frame(height = c(58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68,
    69, 70, 71, 72), weight = c(115, 117, 120, 123, 126, 129, 132, 135, 139,
    142, 146, 150, 154, 159, 164))
```

La sélection des colonnes se fait en utilisant la fonction `select()`. On peut noter que le nom des colonnes ne doit pas être entouré de guillemets. Le premier paramètre correspond au *data.frame*, le second et les suivants à la ou les colonnes désirées. L'odre d'apparition des colonnes correspond à celui fourni à la fonction `select()`.

```
library(dplyr)
select(femmes, height)

##    height
## 1      58
## 2      59
## 3      60
## 4      61
## 5      62
```

4. https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html

```
## 6      63
## 7      64
## 8      65
## 9      66
## 10     67
## 11     68
## 12     69
## 13     70
## 14     71
## 15     72
```

Dans les exemples qui suivent, l'opérateur `\%>\%` (pipe) introduit dans le *package* `magrittr`[5] va être utilisé. Cet opérateur permet de transmettre une valeur, ou le résultat d'une expression à une expression ou un appel suivant. Il permet une lisibilité plus grande.

```
femmes %>%
  select(weight, height) %>%
  head()

##   weight height
## 1    115     58
## 2    117     59
## 3    120     60
## 4    123     61
## 5    126     62
## 6    129     63
```

Le résultat obtenu est un *data.frame* :

```
femmes %>%
  select(weight, height) %>%
  class()

## [1] "data.frame"
```

Pour extraire une colonne en particulier, si on ne désire pas un *data.frame* en sortie, mais un vecteur, il faut utiliser le symbole dollar :

```
femmes %>%
  select(weight, height) %>%
  .$weight

##  [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
```

Pour exclure une colonne, il faut faire précéder son nom du symbole `-` :

```
femmes %>%
  select(-weight) %>%
  head()
```

5. L'auteur du *package* indique que le nom de son *package* doit être prononcé avec un accent français raffiné.

```
##   height
## 1     58
## 2     59
## 3     60
## 4     61
## 5     62
## 6     63
```

Parfois, le nom des variables à extraire est fourni sous forme de chaîne de caractères (ça peut être le cas lors de la construction d'une fonction que l'on définit). La fonction `select()` ne permet pas de passer le nom des colonnes sous forme de chaînes de caractères. Heureusement, il existe un *package*, `modify`, qui s'en charge. Il est nécessaire d'installer le *package* lors de la première utilisation :

```
library(devtools)
install_github(repo = "modify", username = "skranz")
```

```
library(modify)
noms <- c("weight", "height")
femmes %>%
  s_select(noms) %>%
  head()

##   weight height
## 1    115     58
## 2    117     59
## 3    120     60
## 4    123     61
## 5    126     62
## 6    129     63
```

La sélection de lignes peut se faire par le système d'indexation avec la fonction `slice()` du *package* `dplyr` :

```
# Les lignes 4 et 5 du data.frame femmes
femmes %>%
  slice(4:5)

##   height weight
## 1     61    123
## 2     62    126
```

2.4.6.2 Filtrage

Le filtrage par masque a été abordé dans la section 2.4.2. Encore une fois, le *package* `dplyr` propose une fonction très simple d'emploi pour sélectionner des lignes : `filter()`. Le premier paramètre est le nom du *data.frame*, le second et les suivants correspondent aux expressions à évaluer à l'intérieur du *data.frame*.

```
# Les femmes dont la taille vaut exactement 60 pouces
femmes %>%
  filter(height == 60)

##   height weight
## 1     60    120

# Les femmes dont la masse vaut plus de 120 libres
# et dont la taille est inférieure ou égale à 62 pouces
femmes %>%
  filter(weight > 120, height <= 62)

##   height weight
## 1     61    123
## 2     62    126

# Les femmes dont la masse vaut 150 ou 159 livres
# ou dont la taille est égale à 62 pouces
femmes %>%
  filter(weight %in% c(150,159) | height == 62)

##   height weight
## 1     62    126
## 2     69    150
## 3     71    159
```

2.4.6.3 Retirer les valeurs dupliquées

Pour se séparer des observations contenues plusieurs fois dans un *data.frame*, on utilise la fonction `unique()`.

```
# Les observations 1 et 7, ainsi que 3 et 6 sont dupliquées
df <- data.frame(x = letters[c(1,2,3,1,4,3,1)],
                 y = letters[c(1,5,2,4,3,2,1)])

unique(df)

##   x y
## 1 a a
## 2 b e
## 3 c b
## 4 a d
## 5 d c
```

2.4.6.4 Modification des colonnes

La section 2.4.2 a posé les bases au sujet de la modification d'objets. Cette section apporte quelques informations supplémentaires.
Une des premières modifications que l'on peut apporter à un *data.frame* est de renommer les colonnes. On peut s'en sortir sans fournir trop d'efforts à l'aide de la fonction `colnames()` et de la flèche d'assignation :

```
colnames(femmes)

## [1] "height" "weight"

colnames(femmes)[which(colnames(femmes) == "height")] <- "taille"
colnames(femmes)

## [1] "taille" "weight"
```

Cependant, cette syntaxe est assez lourde. La fonction `rename()` du *package* `dplyr` propose de réaliser la même tâche de manière plus lisible :

```
femmes <-
  femmes %>%
  rename(masse = weight)
colnames(femmes)

## [1] "taille" "masse"
```

Remarque 2.4.13

Avec la fonction `select()`, il est possible de sélectionner une variable et de la renommer directement :

```
femmes %>%
  select(poids = masse, hauteur = taille) %>%
  head(2)

##   poids hauteur
## 1   115      58
## 2   117      59
```

Pour ajouter une colonne, ou pour appliquer des changements à tous les éléments d'une colonne, on peut utiliser la flèche d'assignation :

```
# Nombre de chomeurs et d'actifs en millions de personnes
chomeurs <- data.frame(annee = 2012:2008,
nb_chomeurs = c(2.811, 2.604, 2.635, 2.573, 2.064),
pop_active = c(28.328, 28.147, 28.157, 28.074, 27.813))

chomeurs$taux_chomage_0 <- chomeurs$nb_chomeurs / chomeurs$pop_active * 100
```

Comme on peut le voir, il est assez laborieux de devoir réécrire le nom du *data frame* pour accéder à ses colonnes afin de faire des modifications ou des créations de variables (bien qu'avec uniquement deux variables à utiliser, cela reste encore faisable sans trop de pénibilité). Le emphpackage `dplyr` propose la fonction `mutate()` pour créer ou modifier une ou plusieurs colonnes.

```
chomeurs <-
  chomeurs %>%
  mutate(tx_chomage_1 = nb_chomeurs/pop_active*100,
```

```
         log_nb_chomeurs = log(nb_chomeurs))
chomeurs

##   annee nb_chomeurs pop_active taux_chomage_0 tx_chomage_1 log_nb_chomeurs
## 1  2012       2.811     28.328       9.923044     9.923044       1.0335403
## 2  2011       2.604     28.147       9.251430     9.251430       0.9570487
## 3  2010       2.635     28.157       9.358241     9.358241       0.9688832
## 4  2009       2.573     28.074       9.165064     9.165064       0.9450725
## 5  2008       2.064     27.813       7.420990     7.420990       0.7246458
```

Avec la fonction `mutate()`, il est possible de faire référence directement à la variable créé :

```
chomeurs <-
  chomeurs %>%
  mutate(tx_chomage_2 = nb_chomeurs/pop_active*100,
         log_tx_chomage = log(tx_chomage_2))
chomeurs

##   annee nb_chomeurs pop_active taux_chomage_0 tx_chomage_1 log_nb_chomeurs
## 1  2012       2.811     28.328       9.923044     9.923044       1.0335403
## 2  2011       2.604     28.147       9.251430     9.251430       0.9570487
## 3  2010       2.635     28.157       9.358241     9.358241       0.9688832
## 4  2009       2.573     28.074       9.165064     9.165064       0.9450725
## 5  2008       2.064     27.813       7.420990     7.420990       0.7246458
##   tx_chomage_2 log_tx_chomage
## 1     9.923044       2.294860
## 2     9.251430       2.224778
## 3     9.358241       2.236257
## 4     9.165064       2.215399
## 5     7.420990       2.004312
```

Pour modifier une colonne, il suffit de créer une colonne portant le même nom :

```
chomeurs %>%
  mutate(annee = annee / 1000)

##   annee nb_chomeurs pop_active taux_chomage_0 tx_chomage_1 log_nb_chomeurs
## 1 2.012       2.811     28.328       9.923044     9.923044       1.0335403
## 2 2.011       2.604     28.147       9.251430     9.251430       0.9570487
## 3 2.010       2.635     28.157       9.358241     9.358241       0.9688832
## 4 2.009       2.573     28.074       9.165064     9.165064       0.9450725
## 5 2.008       2.064     27.813       7.420990     7.420990       0.7246458
##   tx_chomage_2 log_tx_chomage
## 1     9.923044       2.294860
## 2     9.251430       2.224778
## 3     9.358241       2.236257
## 4     9.165064       2.215399
## 5     7.420990       2.004312
```

Pour conserver uniquement les variables nouvellement créées, on peut utiliser la fonction `transmute()` du *package* `dplyr` :

```
chomeurs %>%
  transmute(annee = annee / 1000,
            tx_chomage = nb_chomeurs/pop_active*100)

##   annee tx_chomage
## 1 2.012   9.923044
## 2 2.011   9.251430
## 3 2.010   9.358241
## 4 2.009   9.165064
## 5 2.008   7.420990
```

Remarque 2.4.14

En dehors du *package* `dplyr`, il existe trois fonctions, qui font gagner du temps dans la rédaction : `with()`, `within()` et `transform()`. Leur emploi permet d'aboutir au même point.

```
# Changements simples
chomeurs <- data.frame(annee = 2012:2008,
nb_chomeurs = c(2.811, 2.604, 2.635, 2.573, 2.064),
pop_active = c(28.328, 28.147, 28.157, 28.074, 27.813))

chomeurs$tx_chomage_1 <- with(chomeurs, nb_chomeurs/pop_active*100)
chomeurs <- within(chomeurs, tx_chomage_2 <- nb_chomeurs/pop_active*100)
chomeurs <- transform(chomeurs, tx_chomage_3 = nb_chomeurs/pop_active*100)
chomeurs
```

```
##   annee nb_chomeurs pop_active tx_chomage_1 tx_chomage_2 tx_chomage_3
## 1  2012       2.811     28.328     9.923044     9.923044     9.923044
## 2  2011       2.604     28.147     9.251430     9.251430     9.251430
## 3  2010       2.635     28.157     9.358241     9.358241     9.358241
## 4  2009       2.573     28.074     9.165064     9.165064     9.165064
## 5  2008       2.064     27.813     7.420990     7.420990     7.420990
```

Les fonctions `within()` et `transform()` proposent toutefois une syntaxe plus lisible que `with()` lors de changements multiples.

```
# Changements multiples
chomeurs[c("log_nb_chomeurs", "log_pop_active")] <- with(chomeurs, list(
  log(nb_chomeurs), log(pop_active)
  ))
chomeurs <- within(chomeurs,{
  log_nb_chomeurs_2 <- log(nb_chomeurs)
  log_pop_active_2 <- log(pop_active)
  })
chomeurs <- transform(chomeurs,
                      log_nb_chomeurs_3 = log(nb_chomeurs),
                      log_pop_active_3 = log(pop_active)
                      )
chomeurs %>% head(2)
```

```
##   annee nb_chomeurs pop_active tx_chomage_1 tx_chomage_2 tx_chomage_3
## 1  2012       2.811     28.328     9.923044     9.923044     9.923044
## 2  2011       2.604     28.147     9.251430     9.251430     9.251430
##   log_nb_chomeurs log_pop_active log_pop_active_2 log_nb_chomeurs_2
## 1       1.0335403       3.343851         3.343851         1.0335403
## 2       0.9570487       3.337441         3.337441         0.9570487
##   log_nb_chomeurs_3 log_pop_active_3
## 1         1.0335403         3.343851
## 2         0.9570487         3.337441
```

Remarque 2.4.15

Il existe une méthode qui à première vue semble agréable à utiliser : la fonction `attach()`, qui comme son nom l'indique, détache le *data frame*, c'est-à-dire rend les colonnes qui le composent visibles dans l'espace de travail. La fonction `detach()` produit l'effet inverse, et masque alors de l'espace de travail les colonnes du tableau de données indiqué en paramètre.

```
data(quakes)
quakes <- quakes[1:4,]  # On ne prend que quelques observations pour l'exemple
quakes
```

```
##      lat   long depth mag stations
## 1 -20.42 181.62   562 4.8       41
## 2 -20.62 181.03   650 4.2       15
## 3 -26.00 184.10    42 5.4       43
## 4 -17.97 181.66   626 4.1       19
```

```
lat  # lat n'est pas dans l'espace de travail
```

```
## Error in eval(expr, envir, enclos): objet 'lat' introuvable
```

```
attach(quakes)
lat     # maintenant il l'est
```

```
## [1] -20.42 -20.62 -26.00 -17.97
```

```
detach(quakes)
lat     # il est à nouveau masqué
```

```
## Error in eval(expr, envir, enclos): objet 'lat' introuvable
```

Cependant, l'utilisation de la fonction `attach` cause bien souvent des confusions et problèmes lors de l'emploi d'autres fonctions.

2.4.6.5 Tri

Il est aisé de trier un *data.frame* par ordre croissant ou décroissant d'une ou plusieurs de ses colonnes. Pour ce faire, on peut utiliser la fonction `order()`, qui retourne les rangs de classement des éléments du ou des paramètres. En cas d'*ex æquo*, les rangs sont classés par ordre d'apparition dans le *data frame*.

```
df <- data.frame(nom = c("Durand", "Martin", "Martin", "Martin", "Durand"),
                 prenom = c("Sonia", "Serge", "Julien-Yacine", "Victor", "Emma"),
                 note = c(23, 18, 17, 17, 19))
# Ordonner par notes décroissantes
order(df$note, decreasing = TRUE)
```

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

```
df[order(df$note), ]
```

```
##      nom        prenom note
```

```
## 3 Martin Julien-Yacine   17
## 4 Martin        Victor   17
## 2 Martin         Serge   18
## 5 Durand          Emma   19
## 1 Durand         Sonia   23

# Ordonner par ordre alphabétique des noms puis des prénoms
df[with(df, order(nom, prenom)), ]

##      nom        prenom note
## 5 Durand          Emma   19
## 1 Durand         Sonia   23
## 3 Martin Julien-Yacine   17
## 2 Martin         Serge   18
## 4 Martin        Victor   17
```

Remarque 2.4.16

La fonction `sort()` retourne un vecteur trié par valeurs croissantes ou décroissantes. La fonction `order()` retourne les rangs du classement.

La fonction `arrange()` du *package* `dplyr` permet aussi de trier un tableau de données. Son utilisation semble plus aisée pour certains, surtout lorsque le tri s'effectue en fonction de plusieurs colonnes. Comme il n'y a pas de paramètre `decreasing` deux méthodes sont possibles :

1. faire appel à la fonction `desc()`, elle aussi contenue dans le *package* `dplyr`, qui permet de trier le vecteur donné en paramètre par ordre de valeurs décroissantes ;
2. faire préceder le nom de la colonne par le signe `-` si les valeurs de la colonne sont numériques.

```
# Ordonner par notes décroissantes
df %>% arrange(-note)

##      nom        prenom note
## 1 Durand         Sonia   23
## 2 Durand          Emma   19
## 3 Martin         Serge   18
## 4 Martin Julien-Yacine   17
## 5 Martin        Victor   17

# Ordonner par ordre alphabétique des noms puis inverse des prénoms
df %>% arrange(nom, desc(prenom))

##      nom        prenom note
## 1 Durand         Sonia   23
## 2 Durand          Emma   19
## 3 Martin        Victor   17
## 4 Martin         Serge   18
## 5 Martin Julien-Yacine   17
```

2.4.6.6 Jointures

Deux fonctions permettent de juxtaposer deux *data.frames* ou plus entre-elles (cela fonctionne aussi avec des matrices) : `cbind()` et `rbind()`. La première fusionne les colonnes tandis que la seconde fusionne les lignes. Il faut toutefois veiller à ce que le nombre de lignes des éléments passés en paramètre de `cbind()` correspondent, et que le nombre de colonnes de ceux passés en paramètres de `rbind()` correspondent aussi. De plus, lors de l'utilisation de `rbind()`, le nom des colonnes doit correspondre.

```
(A <- data.frame(x1 = c(1, -1), x2 = c(0, 3)))

##   x1 x2
## 1  1  0
## 2 -1  3

(B <- data.frame(x1 = c(3,2), x2 = c(1, 1)))

##   x1 x2
## 1  3  1
## 2  2  1

(C <- data.frame(x3 = c(0, 3)))

##   x3
## 1  0
## 2  3

rbind(A,B)

##   x1 x2
## 1  1  0
## 2 -1  3
## 3  3  1
## 4  2  1

cbind(A,B)

##   x1 x2 x1 x2
## 1  1  0  3  1
## 2 -1  3  2  1

cbind(A,B,C)

##   x1 x2 x1 x2 x3
## 1  1  0  3  1  0
## 2 -1  3  2  1  3
```

Toutefois, lors de la fusion de *data.frames*, l'emploi de la fonction `cbind()` n'est pas très pratique, puisque s'il existe une colonne identique aux tableaux de données à fusionner, elle sera dupliquée dans le résultat de la fusion.

```
(pib <- data.frame(annee = 2010:2013, pib = c(1998.5, 2059.3, 2091.1, 2113.7)))

##   annee    pib
```

```
## 1  2010 1998.5
## 2  2011 2059.3
## 3  2012 2091.1
## 4  2013 2113.7
```

```
(importations <- data.frame(annee = 2010:2013, importations = c(558.1, 625.3,
    628.5, 629.1)))
```

```
##   annee importations
## 1  2010        558.1
## 2  2011        625.3
## 3  2012        628.5
## 4  2013        629.1
```

```
cbind(pib, importations)
```

```
##   annee    pib annee importations
## 1  2010 1998.5  2010        558.1
## 2  2011 2059.3  2011        625.3
## 3  2012 2091.1  2012        628.5
## 4  2013 2113.7  2013        629.1
```

Le *package* `dplyr` propose des fonctions pour joindre deux tableaux de données. Ces fonctions ont la même syntaxe : `xx_join(x, y, by = NULL, copy = FALSE, ...)`, où `x` et `y` sont les tableaux à joindre, `by` est un vecteur de chaînes de caractères contenant le nom des variables permettant la jointure (si la valeur est `NULL`, ce qui est le cas par défaut, la jointure se fera à l'aide des variables portant le même nom dans les deux tables).

- `inner_join()` : toutes les lignes de `x` pour lesquelles il y a des valeurs correspondantes dans `y`, et toutes les colonnes de `x` et `y`. S'il y a plusieurs correspondances dans les noms entre `x` et `y`, toutes les combinaisons possibles sont retournées ;
- `left_join()` : toutes les lignes de `x`, et toutes les colonnes de `x` et `y`. Les lignes dans `x` pour lesquelles il n'y a pas de correspondance dans `y` auront des valeurs `NA` dans les nouvelles colonnes. S'il y a plusieurs correspondances dans les noms entre `x` et `y`, toutes les combinaisons sont retournées ;
- `right_join()` : toutes les lignes de `y`, et toutes les colonnes de `x` et `y`. Les lignes dans `y` pour lesquelles il n'y a pas de correspondance dans `x` auront des valeurs `NA` dans les nouvelles colonnes. S'il y a plusieurs correspondances dans les noms entre `x` et `y`, toutes les combinaisons sont retournées ;
- `semi_join()` : toutes les lignes de `x` pour lesquelles il y a des valeurs correspondantes dans `y`, en ne conservant uniquement les colonnes de `x` ;
- `anti_join()` : toutes les lignes de `x` pour lesquelles il n'y a pas de correspondances dans `y`, en ne conservant que les colonnes de `x`.

```
exportations <- data.frame(year = 2011:2013, exportations = c(572.6, 587.3,
    597.8))
importations <- data.frame(annee = 2010:2012, importations = c(558.1, 625.3,
    628.5))
exportations
```

```
##   year exportations
## 1 2011        572.6
## 2 2012        587.3
```

```
## 3 2013        597.8

importations

##   annee importations
## 1  2010        558.1
## 2  2011        625.3
## 3  2012        628.5

exportations %>% inner_join(importations, by = c(year = "annee"))

##   year exportations importations
## 1 2011        572.6        625.3
## 2 2012        587.3        628.5

exportations %>% left_join(importations, by = c(year = "annee"))

##   year exportations importations
## 1 2011        572.6        625.3
## 2 2012        587.3        628.5
## 3 2013        597.8           NA

exportations %>% right_join(importations, by = c(year = "annee"))

##   year exportations importations
## 1 2010           NA        558.1
## 2 2011        572.6        625.3
## 3 2012        587.3        628.5

exportations %>% semi_join(importations, by = c(year = "annee"))

##   year exportations
## 1 2011        572.6
## 2 2012        587.3

exportations %>% anti_join(importations, by = c(year = "annee"))

##   year exportations
## 1 2013        597.8
```

Remarque 2.4.17

La fonction `merge()` permet de fusionner deux *data.frames* par noms de colonnes ou de lignes. Si le ou les noms de colonnes qui permettent de relier les deux tables sont différents d'une table à l'autre, il faut employer les paramètres `by.x` et `by.y` pour indiquer à `R` leurs noms. Le paramètre `all` (`all.x`) [`all.y`] lorsqu'il vaut `TRUE` indique que l'on souhaite faire figurer toutes les valeurs présentes dans chacun des deux tableaux de données (du premièr *data frame*) [du second *data frame*], et laisser `R` ajouter des valeurs manquantes (`NA`) si necessaire.

```
merge(pib, importations)
```

```
##   annee    pib importations
## 1  2010 1998.5        558.1
## 2  2011 2059.3        625.3
## 3  2012 2091.1        628.5
```

```
exportations <- data.frame(year = 2011:2013, exportations = c(572.6, 587.3,
    597.8))
merge(pib, exportations, by.x = c("annee"), by.y = c("year"))
```

```
##   annee    pib exportations
## 1  2011 2059.3        572.6
## 2  2012 2091.1        587.3
## 3  2013 2113.7        597.8
```

```
merge(pib, exportations, by.x = c("annee"), by.y = c("year"), all.x = TRUE)
```

```
##   annee    pib exportations
## 1  2010 1998.5           NA
## 2  2011 2059.3        572.6
## 3  2012 2091.1        587.3
## 4  2013 2113.7        597.8
```

Dans le cas où l'on souhaite rajouter une colonne d'une *data frame* à une autre en se basant sur une autre colonne de référence, on peut aussi utiliser la fonction `match()` et la flèche d'assignation.

```
df <- pib
match(df$annee, exportations$year)
```

```
## [1] NA  1  2  3
```

```
df$exportations <- exportations[match(df$annee,
                                      exportations$year), "exportations"]
df
```

```
##   annee    pib exportations
## 1  2010 1998.5           NA
## 2  2011 2059.3        572.6
## 3  2012 2091.1        587.3
## 4  2013 2113.7        597.8
```

2.4.6.7 Agrégation

Il arrive de vouloir agréger les valeurs d'une variable, pour passer par exemple d'une dimention trimestrielle à annuelle. Avec des observations spatiales, cela peut aussi être le cas, comme par exemple lorsque l'on dispose de données à l'échelle des départements et que l'on souhaite connaître les valeurs agrégées à l'échelle des régions.
Le *package* `dplyr` propose une fonction simple d'utilisation : `summarise()`. Elle permet de réduire une colonne d'un *data.frame* à une seule observation. On lui fournit un *data.frame* en premier paramètre, et une ou plusieurs opérations à réaliser au *data.frame* en paramètres suivants.

```
# Nombre d'ingenieurs et cadres au chômage
chomage <- data.frame(region = rep(c(rep("Bretagne", 4), rep("Corse", 2)), 2),
                      departement = rep(c("Cotes-d'Armor", "Finistere",
                                          "Ille-et-Vilaine", "Morbihan",
                                          "Corse-du-Sud", "Haute-Corse"), 2),
                      annee = rep(c(2011, 2010), each = 6),
                      ouvriers = c(8738, 12701, 11390, 10228, 975, 1297,
                                   8113, 12258, 10897, 9617, 936, 1220),
                      ingenieurs = c(1420, 2530, 3986, 2025, 259, 254,
                                     1334, 2401, 3776, 1979, 253, 241))
chomage
```

```
##      region     departement annee ouvriers ingenieurs
## 1  Bretagne   Cotes-d'Armor  2011     8738       1420
## 2  Bretagne       Finistere  2011    12701       2530
## 3  Bretagne Ille-et-Vilaine  2011    11390       3986
## 4  Bretagne        Morbihan  2011    10228       2025
## 5     Corse    Corse-du-Sud  2011      975        259
## 6     Corse     Haute-Corse  2011     1297        254
## 7  Bretagne   Cotes-d'Armor  2010     8113       1334
## 8  Bretagne       Finistere  2010    12258       2401
## 9  Bretagne Ille-et-Vilaine  2010    10897       3776
## 10 Bretagne        Morbihan  2010     9617       1979
## 11    Corse    Corse-du-Sud  2010      936        253
## 12    Corse     Haute-Corse  2010     1220        241
```

```
# Moyenne de la colonne ouvriers
chomage %>% summarise(moy_ouvriers = mean(ouvriers),
                      sd_ouvriers = sd(ouvriers),
                      moy_ingenieurs = mean(ouvriers),
                      sd_ingenieurs = sd(ingenieurs))
```

```
##   moy_ouvriers sd_ouvriers moy_ingenieurs sd_ingenieurs
## 1     7364.167    4801.029       7364.167      1331.482
```

Couplée à la fonction `group_by()` du même *package*, elle prend toute sa puissance. La fonction `group_by()` permet de regrouper les observations d'un *data.frame* en sous-groupes, qui sont créés en fonction des variables données en paramètres.

```
# Agrégation par année
chomage %>%
  group_by(annee) %>%
```

```
  summarise(ouvriers = sum(ouvriers),
            ingenieurs = sum(ingenieurs))

## Source: local data frame [2 x 3]
##
##   annee ouvriers ingenieurs
##   (dbl)    (dbl)      (dbl)
## 1  2010    43041       9984
## 2  2011    45329      10474

# Agrégation par année et par région
chomage %>%
  group_by(annee, region) %>%
  summarise(ouvriers = sum(ouvriers),
            ingenieurs = sum(ingenieurs))

## Source: local data frame [4 x 4]
## Groups: annee [?]
##
##   annee   region ouvriers ingenieurs
##   (dbl)   (fctr)    (dbl)      (dbl)
## 1  2010 Bretagne    40885       9490
## 2  2010    Corse     2156        494
## 3  2011 Bretagne    43057       9961
## 4  2011    Corse     2272        513
```

Remarque 2.4.18

Hormis *via* `dplyr`, `R` propose une fonction, `aggregate()`, qui découpe un tableau de données en fonction d'un ou plusieurs facteurs. La liste fournie au paramètre `by` précise la ou les colonnes qui serviront de facteurs pour l'agrégation, la fonction fournie au paramètre `FUN` indique la modification à appliquer aux sous-populations qui suivent le découpage.

```
# Nombre d'ingenieurs et cadres au chômage
chomage <- data.frame(region = rep(c(rep("Bretagne", 4), rep("Corse", 2)),2),
                      departement = rep(c("Cotes-d'Armor", "Finistere",
                                          "Ille-et-Vilaine", "Morbihan",
                                          "Corse-du-Sud", "Haute-Corse"),2),
                      annee = rep(c(2011, 2010), each = 6),
                      ouvriers = c(8738, 12701, 11390, 10228, 975, 1297,
                                   8113, 12258, 10897, 9617, 936, 1220),
                      ingenieurs = c(1420, 2530, 3986, 2025, 259, 254,
                                     1334, 2401, 3776, 1979, 253, 241))
chomage
```

```
##      region     departement annee ouvriers ingenieurs
## 1  Bretagne   Cotes-d'Armor  2011     8738       1420
## 2  Bretagne       Finistere  2011    12701       2530
## 3  Bretagne Ille-et-Vilaine  2011    11390       3986
## 4  Bretagne        Morbihan  2011    10228       2025
## 5     Corse    Corse-du-Sud  2011      975        259
## 6     Corse     Haute-Corse  2011     1297        254
## 7  Bretagne   Cotes-d'Armor  2010     8113       1334
## 8  Bretagne       Finistere  2010    12258       2401
## 9  Bretagne Ille-et-Vilaine  2010    10897       3776
## 10 Bretagne        Morbihan  2010     9617       1979
## 11    Corse    Corse-du-Sud  2010      936        253
## 12    Corse     Haute-Corse  2010     1220        241
```

```
# Agrégation par année
aggregate(chomage[, c("ouvriers", "ingenieurs")],
          by = list(annee = chomage$annee), FUN = sum)
```

```
##   annee ouvriers ingenieurs
## 1  2010    43041       9984
## 2  2011    45329      10474
```

```
# Agrégation par année et par région
aggregate(chomage[, c("ouvriers", "ingenieurs")],
          by = list(annee = chomage$region, region = chomage$annee),
          FUN = sum)
```

```
##      annee region ouvriers ingenieurs
## 1 Bretagne   2010    40885       9490
## 2    Corse   2010     2156        494
## 3 Bretagne   2011    43057       9961
## 4    Corse   2011     2272        513
```

2.4.6.8 Stacking et unstacking

Pour empiler (*stack*) les contenus des colonnes d'un tableau de données dans un seul vecteur, `R` propose la fonction `stack()`. Le *data frame* retourné est composé de deux colonnes, dont la première contient les valeurs et la seconde le nom de la colonne dans le *data frame* initial. Pour réaliser l'opération inverse, on peut utiliser la fonction `unstack()`.

```
patients <- data.frame(traitement_1 = c(4, 2, 7),
                       traitement_2 = c(8, 5, 0),
                       traitement_3 = c(0, 5, 6))
patients
```

```
##   traitement_1 traitement_2 traitement_3
## 1            4            8            0
## 2            2            5            5
## 3            7            0            6
```

```
stack(patients)
```

```
##   values          ind
## 1      4 traitement_1
## 2      2 traitement_1
## 3      7 traitement_1
## 4      8 traitement_2
## 5      5 traitement_2
## 6      0 traitement_2
## 7      0 traitement_3
## 8      5 traitement_3
## 9      6 traitement_3
```

```
unstack(stack(patients))
```

```
##   traitement_1 traitement_2 traitement_3
## 1            4            8            0
## 2            2            5            5
## 3            7            0            6
```

Le *package* `tidyr` propose des fonctions intéressantes pour convertir des tableaux en longueur en tableaux en hauteur, et *vice versa*.

Tableau large vers tableau long

Pour passer d'un tableau large à un tableau en longueur, on peut utiliser la fonction `gather()`. Le premier paramètre est le *data.frame*, le paramètre `key` correspond au nom que l'on souhaite donner à la colonne qui va contenir les noms de colonnes du tableau large, le paramètre `value` correspond au nom que l'on souhaite donner à la colonne qui va contenir les valeurs. Les paramètres suivants, s'ils sont fournis, correspondent aux différents niveaux qui seront présents dans la colonne dont le nom est défini par le paramètre `key`.

```
pop <- data.frame(ville = c("Paris", "Paris", "Lyon", "Lyon"),
                  arrondissement = c(1, 2, 1, 2),
                  pop_municipale = c(17443, 22927, 28932, 30575),
                  pop_totale = c(17620, 23102, 29874, 31131))
```

```
pop
```

```
##   ville arrondissement pop_municipale pop_totale
## 1 Paris              1          17443      17620
## 2 Paris              2          22927      23102
## 3  Lyon              1          28932      29874
## 4  Lyon              2          30575      31131
```

```
library(tidyr)
pop_long <-
  pop %>%
  gather(key = type_pop, value = population, pop_municipale,pop_totale)
pop_long
```

```
##   ville arrondissement       type_pop population
## 1 Paris              1 pop_municipale      17443
## 2 Paris              2 pop_municipale      22927
## 3  Lyon              1 pop_municipale      28932
## 4  Lyon              2 pop_municipale      30575
## 5 Paris              1     pop_totale      17620
## 6 Paris              2     pop_totale      23102
## 7  Lyon              1     pop_totale      29874
## 8  Lyon              2     pop_totale      31131
```

De manière alternative, plutôt que de lister les niveaux des variables qui seront dans la colonne dont le nom est défini par le paramètre `key`, on peut exclure les variables qui serviront à définir les différents groupes :

```
pop %>%
  gather(key = type_pop, value = population, -ville, -arrondissement)
```

```
##   ville arrondissement       type_pop population
## 1 Paris              1 pop_municipale      17443
## 2 Paris              2 pop_municipale      22927
## 3  Lyon              1 pop_municipale      28932
## 4  Lyon              2 pop_municipale      30575
## 5 Paris              1     pop_totale      17620
## 6 Paris              2     pop_totale      23102
## 7  Lyon              1     pop_totale      29874
## 8  Lyon              2     pop_totale      31131
```

Tableau long vers tableau large

Pour effectuer l'opération inverse, on peut utiliser la fonction `spread()`. Le premier paramètre est le *data.frame*. Le paramètre `key` est le nom de la colonne qui contient les noms des colonnes que l'on souhaite créer (des niveaux d'une variable). Le paramètre `value` correspond au nom de la colonne contenant les valeurs.

```
pop_long %>%
  spread(type_pop, population)
```

```
##   ville arrondissement pop_municipale pop_totale
## 1  Lyon              1          28932      29874
## 2  Lyon              2          30575      31131
## 3 Paris              1          17443      17620
## 4 Paris              2          22927      23102
```

Remarque 2.4.19

Le *package* `reshape2` propose également des fonctions pour convertir des tableaux en longueur en tableaux en hauteur, et *vice versa*.

Tableau large vers tableau long

Pour passer d'un tableau large à un tableau long, on peut utiliser la fonction `melt()` (qui correspond à `gather()`), en indiquant à travers le paramètre `id.vars` le vecteur contenant le nom des variables à conserver. Le paramètre `value.name` permet de changer le nom de la colonne du *data frame* retourné dans laquelle les valeurs sont empilées. Le paramètre `variable.name` permet de renommer la colonne du *data frame* retourné contenant les modalités.

```
# Un tableau large
library(reshape2)

##
## Attaching package: 'reshape2'
##
## The following objects are masked from 'package:data.table':
##
##    dcast, melt

pop_long <- melt(pop, id.vars=c("ville", "arrondissement"),
                 value.name = "population",
                 variable.name="type_population")
pop_long

##   ville arrondissement type_population population
## 1 Paris              1  pop_municipale      17443
## 2 Paris              2  pop_municipale      22927
## 3  Lyon              1  pop_municipale      28932
## 4  Lyon              2  pop_municipale      30575
## 5 Paris              1      pop_totale      17620
## 6 Paris              2      pop_totale      23102
## 7  Lyon              1      pop_totale      29874
## 8  Lyon              2      pop_totale      31131
```

Tableau long vers tableau large

Pour effectuer l'opération inverse, on peut utiliser la fonction `dcast()` (qui correspond à la fonction `spread()`. Le paramètre `formula` doit contenir un objet de classe `formula` (de type `x ~ y`), qui précise les colonnes que l'on souhaite conserver en fonction de celles qui contiennent les noms des nouvelles colonnes dans lesquelles nous voulons placer les mesures. Le paramètre `value.var` permet justement d'indiquer le nom de la colonne dans laquelle se trouvent les valeurs.

```
dcast(pop_long, formula = ville + arrondissement ~ type_population,
      value.var="population")

##   ville arrondissement pop_municipale pop_totale
## 1  Lyon              1          28932      29874
## 2  Lyon              2          30575      31131
## 3 Paris              1          17443      17620
## 4 Paris              2          22927      23102
```

2.4.7 Data tables

Le *package* `data.table` propose une nouvelle structure de données, appelée `data.table`[6], qui se veut être une alternative aux objets *data.frame*. L'idée est de réduire la longueur du code, de proposer une syntatxe plus facile à écrire et lire, mais aussi de réduire les temps de calculs. La syntaxe se rapproche un peu du `SQL`, mais évite pas mal de lourdeurs de ce langage.

Il est nécessaire de charger le *package*.

```
library(data.table)
```

2.4.7.1 Création, conversion

La création d'un `data.table` se fait de la même manière que pour un `data.frame`, en faisant appel à la fonction `data.table()`.

```
chomage <- data.table(region = rep(c(rep("Bretagne", 4), rep("Corse", 2)), 2),
                      departement = rep(c("Cotes-d'Armor", "Finistere",
                                          "Ille-et-Vilaine", "Morbihan",
                                          "Corse-du-Sud", "Haute-Corse"), 2),
                      annee = rep(c(2011, 2010), each = 6),
                      ouvriers = c(8738, 12701, 11390, 10228, 975, 1297,
                                   8113, 12258, 10897, 9617, 936, 1220),
                      ingenieurs = c(1420, 2530, 3986, 2025, 259, 254,
                                     1334, 2401, 3776, 1979, 253, 241))
chomage
```

```
##       region     departement annee ouvriers ingenieurs
##  1: Bretagne   Cotes-d'Armor  2011     8738       1420
##  2: Bretagne       Finistere  2011    12701       2530
##  3: Bretagne Ille-et-Vilaine  2011    11390       3986
##  4: Bretagne        Morbihan  2011    10228       2025
##  5:    Corse    Corse-du-Sud  2011      975        259
##  6:    Corse     Haute-Corse  2011     1297        254
##  7: Bretagne   Cotes-d'Armor  2010     8113       1334
##  8: Bretagne       Finistere  2010    12258       2401
##  9: Bretagne Ille-et-Vilaine  2010    10897       3776
## 10: Bretagne        Morbihan  2010     9617       1979
## 11:    Corse    Corse-du-Sud  2010      936        253
## 12:    Corse     Haute-Corse  2010     1220        241
```

Il est facile de convertir un `data.frame` en `data.table`, en faiant appel à la même fonction.

```
pop <- data.table(region = rep(c("Bretagne", "Alsace"), each = 2),
                  annee = rep(c(2010, 2011), 2),
                  population = c(3199066, 3217767, 1845687, 1852325))

population <- data.table(pop)
population
```

6. La vignette qui accompagne le *package* est très bien faite : http://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.pdf.

```
##      region annee population
## 1: Bretagne  2010    3199066
## 2: Bretagne  2011    3217767
## 3:   Alsace  2010    1845687
## 4:   Alsace  2011    1852325
```

La fonction `tables()` donne un aperçu des `data.table`s en mémoire.

```
tables()
```

```
##      NAME       NROW NCOL MB COLS
## [1,] chomage      12    5  1 region,departement,annee,ouvriers,ingenieurs
## [2,] pop           4    3  1 region,annee,population
## [3,] population    4    3  1 region,annee,population
##      KEY
## [1,]
## [2,]
## [3,]
## Total: 3MB
```

La fonction `data.frame()` convertit les `data.table()` en `data.frame()`, ce qui est essentiel, dans la mesure où certaines fonctions ne prennent en compte qu'un objet `data.frame` en paramètre, et excluent toute autre structure.

2.4.7.2 Sélection

Comme pour un `data.frame`, on peut accéder aux éléments par leur numéro de ligne. Par contre, l'accès par le numéro de colonne ne fonctionne pas comme avec une matrice ou un *data.frame*.

```
chomage[1,]
```

```
##      region   departement annee ouvriers ingenieurs
## 1: Bretagne Cotes-d'Armor  2011     8738       1420
```

```
# Retourne un résultat pas forcément attendu
chomage[,1]
```

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

```
chomage[1,1]
```

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

Pour accéder aux valeurs dans les colonnes, il faut fournir le nom de la colonne, sans guillemets en second paramètre à la fonction `"[.data.table"()`. Pour choisir plusieurs colonnes, il faut placer le nom des variables, sans guillemets, dans une liste, et le fournir en second paramètre de la fonction `"[.data.table"()`.

```
chomage[1,ouvriers]
```

```
## [1] 8738
```

```
chomage[1, list(ouvriers, ingenieurs)]
```

```
##    ouvriers ingenieurs
## 1:     8738       1420
```

2.4.7.3 Filtrage

On peut extraire des sous parties du `data.table` à l'aide des opérateurs logiques qui scannent la totalité des données. Cependant, la section 2.4.7.5 propose un moyen beaucoup plus rapide, surtout avec des gros volumes de données, pour effectuer des recherches dans un `data.table`.

```
chomage[annee == 2010 & departement %in% c("Finistere", "Morbihan"),
        list(departement, ouvriers, annee)]

##    departement ouvriers annee
## 1:   Finistere    12258  2010
## 2:    Morbihan     9617  2010
```

2.4.7.4 Retirer les valeurs dupliquées

Il arrive parfois, après avoir effectué des opérations sur les tables, d'obtenir des duplications d'enregistrement. Comme pour les *data.frames*, il faut se servir de la fonction `unique()` pour retirer les doublons. Attention, si le `data.table` possède une clé, il est nécessaire de la retirer avant d'appeler la fonction `unique()`.

```
# Les observations 1 et 7, ainsi que 3 et 6 sont dupliquées
dt <- data.table(x = letters[c(1,2,3,1,4,3,1)],
                 y = letters[c(1,5,2,4,3,2,1)])

# setkey(dt, NULL)
unique(dt)

##    x y
## 1: a a
## 2: b e
## 3: c b
## 4: a d
## 5: d c
```

2.4.7.5 Clés

Dans les bases de données relationnelles, la présence d'une clé primaire permet d'identifier chaque observation de la base. Celle-ci peut être composée d'un seul ou de plusieurs champs. Ici, avec les `data.table`, il en est à peu près de même. Une clé (*key*) peut être composée d'une seule ou de plusieurs variables (colonnes) du `data.table`, qui peuvent être de mode `factor`, `numeric`, `integer`, `character`, etc. Les lignes sont ordonnées en fonction de la clé, ce qui implique l'impossibilité d'avoir plusieurs clés. Les clés dupliquées sont autorisées.

Pour définir une clé, on peut utiliser la fonction `setkey()`.

```
setkey(chomage, departement)
tables()

##      NAME       NROW NCOL MB COLS
## [1,] chomage      12    5  1 region,departement,annee,ouvriers,ingenieurs
## [2,] dt            7    2  1 x,y
```

```
## [3,] pop           4    3  1 region,annee,population
## [4,] population    4    3  1 region,annee,population
##      KEY
## [1,] departement
## [2,]
## [3,]
## [4,]
## Total: 4MB
```

Comme précisé plus haut, on peut voir à présent que le `data.table` a été ordonné en fonction de la clé.

```
chomage
```

```
##       region     departement annee ouvriers ingenieurs
##  1:    Corse    Corse-du-Sud  2011      975        259
##  2:    Corse    Corse-du-Sud  2010      936        253
##  3: Bretagne   Cotes-d'Armor  2011     8738       1420
##  4: Bretagne   Cotes-d'Armor  2010     8113       1334
##  5: Bretagne       Finistere  2011    12701       2530
##  6: Bretagne       Finistere  2010    12258       2401
##  7:    Corse     Haute-Corse  2011     1297        254
##  8:    Corse     Haute-Corse  2010     1220        241
##  9: Bretagne Ille-et-Vilaine  2011    11390       3986
## 10: Bretagne Ille-et-Vilaine  2010    10897       3776
## 11: Bretagne        Morbihan  2011    10228       2025
## 12: Bretagne        Morbihan  2010     9617       1979
```

On peut à présent faire référence à une valeur de la clé pour accéder à une information

```
chomage["Finistere",]
```

```
##      region departement annee ouvriers ingenieurs
## 1: Bretagne   Finistere  2011    12701       2530
## 2: Bretagne   Finistere  2010    12258       2401
```

```
# On n'est pas obligé de mettre la virgule
chomage["Finistere"]
```

```
##      region departement annee ouvriers ingenieurs
## 1: Bretagne   Finistere  2011    12701       2530
## 2: Bretagne   Finistere  2010    12258       2401
```

Comme on peut le voir, la clé `"departement"` est dupliqueée pour la valeur `"Finistere"`. De fait, deux observations sont retournées. Si on souhaite obtenir uniquement la première ou la dernière, on peut utiliser le paramètre `mult`, en lui donnant respectivement la valeur `"first"` ou `"last"`.

```
chomage["Finistere", mult = "first"]
```

```
##      region departement annee ouvriers ingenieurs
## 1: Bretagne   Finistere  2011    12701       2530
```

```
chomage["Finistere", mult = "last"]

##      region departement annee ouvriers ingenieurs
## 1: Bretagne   Finistere  2010    12258       2401
```

La clé peut être composée de plusieurs variables, il faut utiliser la fonction `J()`.

```
setkey(chomage, region, departement)
tables()

##      NAME       NROW NCOL MB COLS
## [1,] chomage      12    5  1 region,departement,annee,ouvriers,ingenieurs
## [2,] dt            7    2  1 x,y
## [3,] pop           4    3  1 region,annee,population
## [4,] population    4    3  1 region,annee,population
##      KEY
## [1,] region,departement
## [2,]
## [3,]
## [4,]
## Total: 4MB

chomage["Corse"]

##    region  departement annee ouvriers ingenieurs
## 1:  Corse Corse-du-Sud  2011      975        259
## 2:  Corse Corse-du-Sud  2010      936        253
## 3:  Corse  Haute-Corse  2011     1297        254
## 4:  Corse  Haute-Corse  2010     1220        241

chomage[J("Bretagne", "Finistere")]

##      region departement annee ouvriers ingenieurs
## 1: Bretagne   Finistere  2011    12701       2530
## 2: Bretagne   Finistere  2010    12258       2401
```

À l'heure de la rédaction de ces notes, il n'existe pas encore de moyen de faire une recherche par clé secondaire. Il est cependant possible de contourner le problème en effectuant de nouveau l'opération de définition de clé.

```
# Pour rechercher les observations pour lesquelles le département
# est le Finistere, sans savoir que c'est en Bretagne

# Ne fonctionne pas
chomage["Finistere"]

##       region departement annee ouvriers ingenieurs
## 1: Finistere          NA    NA       NA         NA

setkey(chomage, departement)
chomage["Finistere"]
```

```
##      region departement annee ouvriers ingenieurs
## 1: Bretagne   Finistere  2011    12701       2530
## 2: Bretagne   Finistere  2010    12258       2401

# Retour à la clé précédente
setkey(chomage, region, departement)
```

2.4.7.6 Modification des colonnes

Dans l'application de la fonction `"[.data.table"()` à un `data.table`, le deuxième paramètre peut être une ou plusieurs expressions dont les paramètres sont les noms des colonnes du `data.frame`, sans les guillemets.

Si on souhaite ajouter une nouvelle variable dans le `data.table`, on peut le faire avec le symbole d'assignation, ou bien en utilisant le symbole `:=`.

```
chomage[, a := letters[1:12]]
# Equivalent de
# chomage$temp <- letters[1:12]
```

L'ajout d'une nouvelle variable à partir des valeurs contenues dans le `data.table` est un peu plus subtil. Il faut utiliser le symbole `:=` et il n'est plus possible d'utiliser la flèche `<-`.

```
chomage[, b := ouvriers + ingenieurs]
```

L'ajout de plusieurs variables possède un syntaxe un peu moins intuitive.

```
chomage[, ":=" (c = ouvriers/100, d = paste0(a,b))]
```

Une fonctionnalité pratique est qu'il est possible de définir une variable et de la réutiliser directement après sa définition.

```
chomage[, c("e", "f") := list(tmp <- ouvriers + ingenieurs,
                              round(sqrt(tmp)))]
```

La suppression se fait avec `NULL`, et il faut placer le nom des variables entouré de guillemets dans un vecteur.

```
# Suppression de plusieurs colonnes
chomage[, c("a", "b", "c", "d", "e") := NULL]

# Suppression d'une seule colonne
chomage[, f := NULL]
```

2.4.7.7 Tri

Lorsqu'une clé est définie, les enregistrement du `data.table` sont triés en fonction de cette clé. Si la clé contient plusieurs noms de colonnes, par exemple `var_1, var_2, ...`, le tri s'effectue d'abord par la première variable (`var_1`), puis la seconde (`var_2`), etc.
Il est possible de forcer le tri sur une autre variable, ou plusieurs autres.

```
head(chomage[order(annee)])
```

```
##      region     departement annee ouvriers ingenieurs
## 1: Bretagne   Cotes-d'Armor  2010     8113       1334
## 2: Bretagne       Finistere  2010    12258       2401
## 3: Bretagne Ille-et-Vilaine  2010    10897       3776
## 4: Bretagne        Morbihan  2010     9617       1979
## 5:    Corse    Corse-du-Sud  2010      936        253
## 6:    Corse     Haute-Corse  2010     1220        241
```

2.4.7.8 Copie de `data.table`

Attention, pour copier un `data.table`, il faut utiliser la fonction `copy()` ! En effet, la flèche d'assignation ne réalise pas la création d'une nouvelle variable dont le contenu est désigné par le bout de la flèche, elle réalise une référence.

```
dt <- data.table(x = letters[1:3], y = 4:6)
dt
```

```
##    x y
## 1: a 4
## 2: b 5
## 3: c 6
```

```
# Référence
dt_ref <- dt

# Copie
dt_copie <- copy(dt)

# Ajoutons une variable
dt[, a := rep(1,3)]
dt_ref[, z := paste0(x,y)]
dt_copie[, t := paste0(y,x)]

# Les variables "a" et "z" ont été ajoutées à dt et dt_ref
# sans impacter dt_copie
# La variable "t" n'a été ajoutée qu'à dt_copie
dt
```

```
##    x y a  z
## 1: a 4 1 a4
## 2: b 5 1 b5
## 3: c 6 1 c6
```

```
dt_ref
```

```
##    x y a  z
## 1: a 4 1 a4
## 2: b 5 1 b5
## 3: c 6 1 c6
```

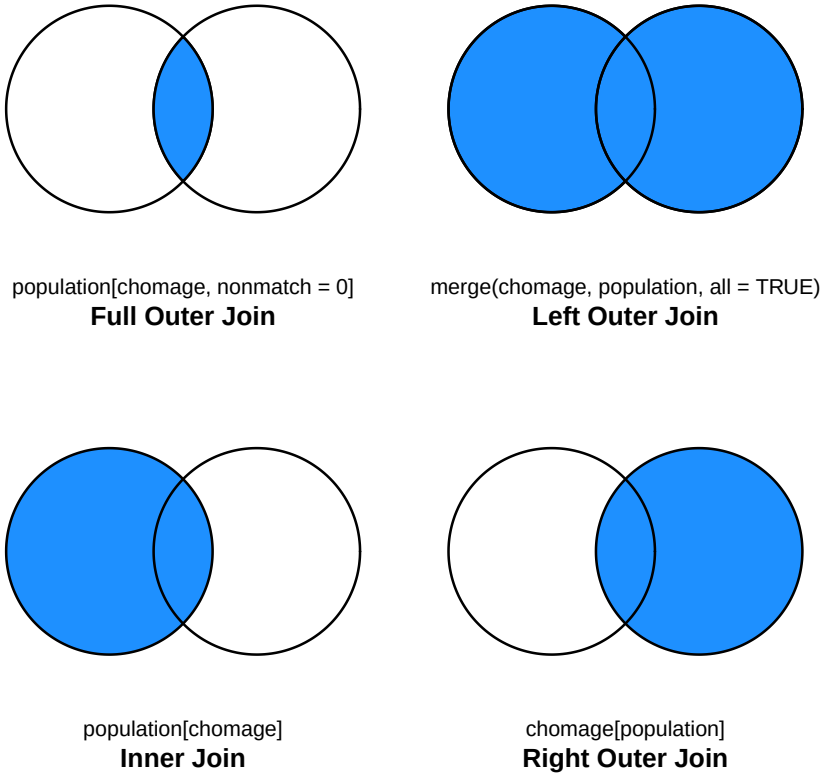
```
dt_copie

##    x y  t
## 1: a 4 4a
## 2: b 5 5b
## 3: c 6 6c

# Par contre, la suppression de dt_ref ne supprime pas dt
rm(dt_ref)
dt

##    x y a  z
## 1: a 4 1 a4
## 2: b 5 1 b5
## 3: c 6 1 c6
```

2.4.7.9 Jointures

Pour réaliser une jointure, on peut utiliser la fonction `merge()`, comme pour les *data.frames* (c.f. section 2.4.6.6). Cependant, le *package* `data.table` fournit d'autres méthodes, plus rapides, en passant, encore une fois, par la fonction `"[.data.frame"()`. La manière la plus simple, pour ne pas faire d'erreur lors de la jointure, est de définir en clé les variables communes aux deux tables.

```
setkey(chomage, region, annee)
setkey(population, region, annee)
chomage[population]

##        region     departement annee ouvriers ingenieurs population
##  1:   Alsace              NA  2010       NA         NA    1845687
##  2:   Alsace              NA  2011       NA         NA    1852325
##  3: Bretagne   Cotes-d'Armor  2010     8113       1334    3199066
##  4: Bretagne       Finistere  2010    12258       2401    3199066
##  5: Bretagne Ille-et-Vilaine  2010    10897       3776    3199066
##  6: Bretagne        Morbihan  2010     9617       1979    3199066
##  7: Bretagne   Cotes-d'Armor  2011     8738       1420    3217767
##  8: Bretagne       Finistere  2011    12701       2530    3217767
##  9: Bretagne Ille-et-Vilaine  2011    11390       3986    3217767
## 10: Bretagne        Morbihan  2011    10228       2025    3217767
```

On peut le voir, la jointure réalisée est de type *Right Join*, c'est à dire que tous les éléments de la seconde table sont présents, mais que seuls ceux de la première à l'intersection avec la seconde sont conservés (on note que les observations concernant la Corse ont disparues). Les diagrammes suivants donnent les autres fusions possibles. On peut remarquer que le *Full Outer Join* n'est pas possible avec la fonction `"[.data.frame"()`.



L'avantage, outre le temps gagné lorsqu'on a de gros volumes de données, d'utiliser `"[.data.frame"()` plutôt que `merge()`, est aussi qu'il est possible de faire une opération juste après la jointure. On gagne à la fois en temps d'exécution, et en lignes de codes, ainsi qu'en clarté de lecture.

```
chomage[population][,round((ouvriers + ingenieurs)/population*100, 2)]

##  [1]   NA   NA 0.30 0.46 0.46 0.36 0.32 0.47 0.48 0.38

chomage[population,
        list(tmp = round((ouvriers + ingenieurs)/population*100, 2), departement)]

##      tmp     departement
##  1:   NA              NA
##  2:   NA              NA
##  3: 0.30   Cotes-d'Armor
##  4: 0.46       Finistere
##  5: 0.46 Ille-et-Vilaine
##  6: 0.36        Morbihan
##  7: 0.32   Cotes-d'Armor
##  8: 0.47       Finistere
##  9: 0.48 Ille-et-Vilaine
## 10: 0.38        Morbihan
```

2.4.7.10 Agrégation

Pour obtenir le résultat d'une fonction appliquée à toute une colonne, il faut travailler avec le second paramètre de la fonction `"[.data.table"()`.

```
chomage[, mean(ouvriers)]

## [1] 7364.167
```

Si on souhaite effectuer des regroupements, il faut rajouter le paramètre `by` et lui atribuer le nom de la colonne (sans guillemets). Si le regroupement doit s'effectuer selon plusieurs colonnes, il faut alors envelopper les noms séparés par des virgules (mais sans espace) par des guillemets. On peut également fournir la liste des noms dans une liste, ce qui est un peu plus souple.

```
chomage[, mean(ouvriers), by = region]

##      region       V1
## 1: Bretagne 10492.75
## 2:    Corse  1107.00

chomage[, mean(ouvriers), by = "region,departement"]

##      region     departement      V1
## 1: Bretagne   Cotes-d'Armor  8425.5
## 2: Bretagne       Finistere 12479.5
## 3: Bretagne Ille-et-Vilaine 11143.5
## 4: Bretagne        Morbihan  9922.5
## 5:    Corse    Corse-du-Sud   955.5
## 6:    Corse     Haute-Corse  1258.5

# Fournit le même résultat
# chomage[, mean(ouvriers), by = list(region, departement)]
```

Il est possible d'effectuer des calculs sur plusieurs des colonnes du `data.table`. Pour ce faire, il faut placer le second paramètre de la fonction `"[.data.table"()` dans une liste. Si on souhaite donner des noms aux variables issues du calcul, c'est possible. Attention, il faut bien être dans une liste pour attribuer le nom !

```
chomage[, list(mean(ouvriers), sd(ouvriers)), by = region]

##      region       V1        V2
## 1: Bretagne 10492.75 1627.4581
## 2:    Corse  1107.00  178.4507

# Ne permet pas d'attribuer le nom "moyenne" au calcul
chomage[, moyenne = mean(ouvriers), by = region]

## Error in `[.data.table`(chomage, , moyenne = mean(ouvriers), by = region): argument
inutilisé (moyenne = mean(ouvriers))

# Il faut être dans une liste
chomage[, list(moyenne = mean(ouvriers)), by = region]

##      region  moyenne
## 1: Bretagne 10492.75
## 2:    Corse  1107.00
```

Les opérations de regroupement sont beaucoup plus rapides qu'avec les *data.frames*.

Par ailleurs, on peut effectuer plusieurs opérations d'un coup, qui dépendent des résultats précédents, en utilisant à nouveau la fonction `"[.data.table"()`.

```
# Créons un data.table immense (presque 7 millions d'obs.)
n <- 10000
dt <- data.table(x = rep(letters, each = 26*n),
                 y = rep(LETTERS, each = n),
                 z = runif(n*26^2))

nrow(dt)

## [1] 6760000

# Après avoir calculé la moyenne de z pour chaque groupe
# défini par les valeurs de x et y,
# on ne conserve que les observations pour lesquelles la moyenne
# fraîchement créée est supérieure à 0.5
dt[, list(moy = mean(z)), by = "x,y"][moy > 0.5]

##      x y       moy
##   1: a F 0.5006888
##   2: a J 0.5042198
##   3: a N 0.5037950
##   4: a O 0.5024583
##   5: a P 0.5021354
##  ---
## 321: z P 0.5005918
## 322: z S 0.5027753
## 323: z V 0.5005497
## 324: z Y 0.5026409
## 325: z Z 0.5018170
```

2.4.8 Quelques fonctions utiles

Le tableau ci-après répertorie quelques fonctions de base qu'il est bon de connaître pour manipuler les données dans `R`.

Fonction	Description
`x%%y`	Modulo
`x%/%y`	Division entière
`ceiling(x)`	Plus petits entier supérieur ou égal à `x`
`floor(x)`	Plus petits entier inférieur ou égal à `x`
`round(x, digits)`	Arrondi de `x` à `digits` décimales près
`signif(x, digits)`	Arrondi de `x` à `digits` chiffres significatifs
`trunc(x)`	Partie entière de `x`
`abs(x)`	Valeur absolue de `x`

continue sur la page suivante

suite du tableau

Fonction	Description
`cor(x)`	Matrice de corrélations, si `x` est une matrice ou une *data frame*
`cor(x, y)`	Corrélation linéaire entre `x` et `y`, ou matrice de corrélations si `x` et `y` sont des matrices ou des *data.frames*
`cummax(x)`	Vecteur dont le i^e élément est le maximum des éléments de `x[1]` à `x[i]`
`cummin(x)`	Vecteur dont le i^e élément est le minimum des éléments de `x[1]` à `x[i]`
`cumprod(x)`	Vecteur dont le i^e élément est le produit des éléments `x[1]` à `x[i]`
`cumsum(x)`	Vecteur dont le i^e élément est la somme des éléments de `x[1]` à `x[i]`
`exp(x)`	Exponentielle de `x`
`log(x, base)`	Logarithme de `x`, avec `base=10` par défaut
`max(..., na.rm)`	Maximum du premier paramètre (peut être de type `numeric`, `logical` ou `character`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`mean(x, na.rm, trim)`	Moyenne de `x`. Si `na.rm = TRUE`, omet les valeurs non disponibles. Le paramètre `trim`, compris entre `0` (par défaut) et `0.5` indique la fraction d'observations à retirer de chaque côté de `x` avant de calculer la moyenne des observations restantes
`median(x, na.rm)`	Médiane de `x`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`min(..., na.rm)`	Minimum du premier paramètre (peut être de type `numeric`, `logical` ou `character`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`prod(..., na.rm)`	Produit des éléments du premier paramètre (peut être de type `numeric`, `logical` ou `complex`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`quantile(x, probs, na.rm)`	Fractiles empiriques de `x` d'ordre `probs`. Si `na.rm = TRUE`, omet les valeurs non disponibles.
`range(..., na.rm, finite)`	Étendue du premier paramètre (peut être de type `numeric`, `logical` ou `character`. Si `na.rm = TRUE`, omet les valeurs non disponibles. Si `finite = TRUE`, les éléments non-finis sont omis.
`sd(x, na.rm)`	Écart-type de `x`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`sign(x)`	Signe de `x`
`sqrt(x)`	Racine carrée de `x`
`sum(..., na.rm)`	Somme du premier paramètre (peut être de type `numeric`, `logical` ou `complex`. Si `na.rm = TRUE`, omet les valeurs non disponibles.

continue sur la page suivante

suite du tableau

Fonction	Description
`var(x, na.rm)` ou `cov(x)`	Variance corrigée de `x` (division par $n-1$ et non n). Si `na.rm = TRUE`, omet les valeurs non disponibles. Si `x` est une matrice ou un *data frame*, le résultat est la matrice de variance-covariance
`var(x, y, na.rm)` ou `cov(x, y)`	Covariance entre `x` et `y`, ou, dans le cas où `x` et `y` sont des matrices ou des *data.frames*, entre les colonnes de `x` et `y`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`sin(x)`	Sinus de `x`
`cos(x)`	Cosinus de `x`
`tan(x)`	Tangente de `x`
`asin(x)`	Arc-sinus de `x`
`acos(x)`	Arc-cosinus de `x`
`atan(x)`	Arc-tangente de `x`
`sinh(x)`	Sinus hyperbolique de `x`
`cosh(x)`	Cosinus hyperbolique de `x`
`tanh(x)`	Tangente hyperbolique de `x`
`asinh(x)`	Arc-sinus hyperbolique de `x`
`acosh(x)`	Arc-cosinus hyperbolique de `x`
`atanh(x)`	Arc-tangente hyperbolique de `x`
`factorial(x)`	Factorielle de `x`
`choose(n, k)`	Coefficient binomial C_n^k
`any(x, na.rm)`	Indique si au moins un élément de `x` vaut `TRUE`. Si `na.rm = TRUE`, omet les valeurs non disponibles
`duplicated(x, fromLast)`	Indique les éléments de `x` qui commencent à être dupliqués. Si `fromLast = TRUE`, le calcul est effectué de droite à gauche
`head(x, n)`	Vecteur des `n` premières valeurs de `x`
`order(x, decreasing)`	Vecteur des rangs de classement des éléments de `x`. Si *ex æquo*, les rangs sont classés par ordre croissant
`rev(x)`	Retourne `x` avec ses éléments dans l'ordre inversé
`table(...)`	Tableau de contingence
`sample(x, size, replace, prob)`	Ré-échantillonnage pseudo-aléatoire (avec remise si `replace = TRUE`) de `x` de taille `size`. Le paramètre optionnel `prob` permet d'attribuer des poids aux éléments
`summary(object)`	Retourne une table indiquant pour `object` : le minimum, le premier quartile, la médiane, la moyenne, le troisième quartile, le maximum et, s'il y en a, le nombre de valeurs manquantes. Les calculs sont effectués en omettant les valeurs manquantes.
`sort(x, decreasing)`	`x` trié par ordre de valeurs croissantes (décroissantes si `decreasing = TRUE`)
`tail(x, n)`	Vecteur des `n` dernières valeurs de `x`

continue sur la page suivante

suite du tableau

Fonction	Description
`unique(x, fromLast)`	Conserve uniquement les valeurs uniques de `x`, en retirant les doublons. Si `fromLast = TRUE`, le calcul indiquant si l'élément est dupliqué est effectué de droite vers la gauche
`which.min(x)`	Position du (premier) minimum de `x`
`which.max(x)`	Position du (premier) maximum de `x`
`"["()`	Fonction d'extraction ou de remplacement de parties d'un objet
`"[["()`	Fonction d'extraction ou de remplacement de parties d'un objet. Les noms sont perdus.
`dim(x)`	Liste des tailles de la matrice ou *data frame* `x`
`nrow(x)`	Nombre de lignes de `x`
`ncol(x)`	Nombre de colonnes de `x`
`dimnames(x)`	Liste des noms des lignes et des colonnes de `x`
`names(x)`	Vecteur du (des) nom(s) de `x`
`colnames(x)`	Nom des colonnes de `x`
`rownames(x)`	Nom des lignes de `x`
`c()`	Concaténation
`cbind(x, y, ...)`	Fusion des matrices ou *data.frames* `x` et `y` par colonnes
`merge(x, y)`	Fusion de deux *data.frames* par noms de colonnes communs
`rbind()`	Fusion des matrices ou *data.frames* `x` et `y` par lignes

2.5 Exercices

Exercice 2.1 : Manipulation de vecteurs

Considérons le vecteur suivant : $x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \end{bmatrix}$.

2.1.1 Créer ce vecteur dans `R` et le stocker dans un objet que l'on appellera `x` ;

2.1.2 Afficher le mode de `x`, puis sa longueur ;

2.1.3 Extraire le premier élément, puis le dernier ;

2.1.4 Extraire les trois premier éléments et les stocker dans un vecteur que l'on nommera `a` ;

2.1.5 Extraire les éléments en position 1, 3, 5 ; les stocker dans un vecteur que l'on nommera `b` ;

2.1.6 Additionner le nombre `10` au vecteur `x`, puis multipliser le résultat par `2` ;

2.1.7 Effectuer l'addition de `a` et `b`, commenter le résultat ;

2.1.8 Effectuer l'addition suivante : `x+a`, commenter le résultat, puis regarder le résultat de `a+x` ;

2.1.9 Multiplier le vecteur par le scalaire `c` que l'on fixera à 2 ;

2.1.10 Effectuer la multiplication de `a` et `b`, commenter le résultat ;

2.1.11 Effectuer la multiplication suivante : `x*a`, commenter le résultat ;

2.1.12 Récupérer les positions des multiples de 2 et les stocker dans un vecteur que l'on nommera `ind`, puis conserver uniquement les multiples de 2 de `x` dans un vecteur que l'on nommera `mult_2` ;

2.1.13 Afficher les éléments de `x` qui sont multiples de 3 **et** multiples de 2 ;

2.1.14 Afficher les éléments de `x` qui sont multiples de 3 **ou** multiples de 2 ;

2.1.15 Calculer la somme des éléments de `x` ;

2.1.16 Remplacer le premier élément de `x` par un `4` ;

2.1.17 Remplacer le premier élément de `x` par la valeur `NA`, puis calculer la somme des éléments de `x` ;

2.1.18 Lister les objets en mémoire dans la session `R` ;

2.1.19 Supprimer le vecteur ;

2.1.20 Supprimer la totalité des objets de la session.

Exercice 2.2 : Manipulation de listes

2.2.1 Évaluer le code suivant : `TRUE+FALSE+TRUE*4` et le commenter ;

2.2.2 Évaluer les expressions suivantes : `c(1, 4, TRUE)`, et `c(1, 4, TRUE, "bonjour")`, commenter ;

2.2.3 Créer une liste que l'on appellera `l` et qui contient les éléments `1`, `4` et `TRUE` en première, seconde et troisième positions respectivement ;

2.2.4 Extraire le premier élément de la liste `l`, et afficher son mode. En faire de même avec le troisième élément, et commenter ;

2.2.5 Ajouter un quatrième élément à la liste `l` : `"bonjour"`, puis afficher la structure de `l` ;

2.2.6 Retirer le troisième élément de la liste `l` ;

2.2.7 Créer une liste de trois éléments : votre nom, votre prénom, et votre année de naissance. Ces trois éléments de la liste devront être nommés respectivement `"nom"`, `""prenom"` et `année de naissance`. Stocker la liste ainsi créée dans un objet nommé `moi` ;

2.2.8 Extraire le prénom de la liste `moi` de deux manières : en utilisant l'indice, et en utilisant le nommage ;

2.2.9 Créer une liste avec la même structure que celle de `moi`, en la remplissant avec les informations d'une autre personne et la nommer `toi`. Puis, créer la liste `personnes`, qui contiendra les listes `toi` et `moi` ;

2.2.10 Extraire la liste `toi` de `personnes` (en première position) ;

2.2.11 Extraire directement depuis `personne` le prénom de l'élément en première position.

Exercice 2.3 : Manipulation de matrices

2.3.1 Créer la matrice suivante : $A = \begin{bmatrix} -3 & 5 & 6 \\ -1 & 2 & 2 \\ 1 & -1 & -1 \end{bmatrix}$;

2.3.2 Afficher la dimension de `A`, son nombre de colonnes, son nombre de lignes et sa longueur ;

2.3.3 Extraire la seconde colonne de `A`, puis la première ligne ;

2.3.4 Extraire l'élément en troisième position à la première ligne ;

2.3.5 Extraire la sous-matrice de dimension 2×2 du coin inférieur de `A`, c'est-à-dire $\begin{bmatrix} 2 & 2 \\ -1 & -1 \end{bmatrix}$;

2.3.6 Calculer la somme des colonnes puis des lignes de `A` ;

2.3.7 Afficher la diagonale de `A` ;

2.3.8 Rajouter le vecteur $\begin{bmatrix}1 & 2 & 3\end{bmatrix}^\top$ à droite de la matrice `A` et stocker le résultat dans un objet appelé `B` ;

2.3.9 Retirer le quatrième vecteur de `B` ;

2.3.10 Retirer la première et la troisième ligne de `B` ;

2.3.11 Ajouter le scalaire `10` à `A` ;

2.3.12 Ajouter le vecteur $\begin{bmatrix}1 & 2 & 3\end{bmatrix}^\top$ à A ;

2.3.13 Ajouter la matrice identité I_3 à `A` ;

2.3.14 Diviser tous les éléments de la matrice `A` par 2 ;

2.3.15 Multiplier la matrice `A` par le vecteur $\begin{bmatrix}1 & 2 & 3\end{bmatrix}^\top$;

2.3.16 Afficher la transposée de `A` ;

2.3.17 Effectuer le produit avec transposition $A^\top A$.

Exercice 2.4 : Importation et exportation

2.4.1 Télécharger le fichier `csv` à l'adresse suivante : egallic.fr/Enseignement/R/Exercices/donnees/notes.csv et le placer dans le répertoire courant du projet. Importer son contenu dans `R` ;

2.4.2 Importer à nouveau les données dans `R`, mais en utilisant fournissant cette fois le l'url directement à la fonction d'importation ;

2.4.3 À présent, importer le contenu du fichier disponible à l'adresse egallic.fr/Enseignement/R/Exercices/donnees/notes_decim.csv. Le séparateur de champs est un point virgule et le séparateur décimal est une virgule ;

2.4.4 Importer le contenu du fichier egallic.fr/Enseignement/R/Exercices/donnees/notes_h.csv. Le nom des colonnes n'est pas présent ;

2.4.5 Importer le contenu du fichier egallic.fr/Enseignement/R/Exercices/donnees/notes_h_s.csv. La première ligne n'est pas à importer ;

2.4.6 Importer le contenu de la première feuille du fichier Excel egallic.fr/Enseignement/R/Exercices/donnees/notes.xlsx ;

2.4.7 Importer le contenu de la seconde feuille (`notes_h_s`) du fichier Excel egallic.fr/Enseignement/R/Exercices/donnees/notes.xlsx. La première ligne est un commentaire à ne pas considérer durant l'importaiton ;

2.4.8 Importer le fichier egallic.fr/Enseignement/R/Exercices/donnees/notes.rda dans `R` ;

2.4.9 Exporter le contenu de l'objet `notes` de la question précédente au format `csv` (virgule en séparateur de champs, point en séparateur décimal, ne pas conserver le numéro des lignes).

2.4.10 Importer le contenu du fichier `notes_2012.csv` contenu dans l'archive disponible à l'adresse suivante : http://egallic.fr/Enseignement/R/Exercices/donnees/notes.zip

Exercice 2.5 : Manipulation de tableaux de données

2.5.1 À l'aide de la fonction `read_excel()` du *package* `readr`, importer le contenu de la feuille intitulée `notes_2012` du fichier Excel disponible à l'adresse suivante : http://egallic.fr/Enseignement/R/Exercices/donnees/notes_etudiants.xlsx et le stocker dans une variable que l'on nommera `notes_2012` ;

2.5.2 Afficher les 6 premières lignes du jeu de données, puis les dimensions du tableau ;

2.5.3 Conserver uniquement la colonne `note_stat` du tableau de données `notes_2012` dans un objet que l'on appellera `tmp` ;

2.5.4 Conserver uniquement les colonnes `num_etudiant`, `note_stat` et `note_macro` dans l'objet `tmp` ;

2.5.5 Remplacer le contenu de `tmp` par les observations de `notes_2012` pour lesquelles l'individu a obtenu une note de stat supérieure (strictement) à 10 ;

2.5.6 Remplacer le contenu de `tmp` par les observations de `notes_2012` pour lesquelles l'individu a obtenu une note comprise dans l'intervalle $(10, 15)$;

2.5.7 Regarder s'il y a des doublons dans le tableau de données `notees_2012` ; le cas échéant, les retirer du tableau ;

2.5.8 Afficher le type des données de la colonne `num_etudiant`, puis afficher le type de toutes les colonnes de `notes_2012` ;

2.5.9 Ajouter au tableau `notes_2012` les colonnes suivantes :

(a) `note_stat_maj` : la note de stat (`note_stat`) majorée d'un point,

(b) `note_macro_maj` : la note de macro (`note_macro`) majorée de trois points (le faire en deux étapes : d'abord deux points en plus, puis un point) ;

2.5.10 Renommer la colonne `year` en `annee` ;

2.5.11 Depuis le fichier `notes_etudiants.xlsx` (c.f. question 1), importer le contenu des feuilles `notes_2013`, `notes_2014` et `prenoms` et le stocker dans les objets `notes_2013`, `notes_2014` et `prenoms` respectivement ;

2.5.12 Empiler le contenu des tableaux de données `notes_2012`, `notes_2013` et `notes_2014` dans un objet que l'on nommera `notes` ;

2.5.13 Fusionner les tableaux `notes` et `prenoms` à l'aide d'une jointure gauche, de manière à rajouter les informations contenues dans le tableau `prenoms` aux observations de `notes`. La jointure doit se faire par le numéro détudiant et l'année, l'objet final viendra remplacer le contenu de `notes` ;

2.5.14 Trier le tableau `notes` par années croissantes et notes de macro décroissantes ;

2.5.15 Changer le type des colonnes `annee` et `sexe` en facteur ;

2.5.16 Créer une colonne `apres_2012` qui prend la valeur `TRUE` si l'observation concerne une note attribuée après 2012 ;

2.5.17 À l'aide de la fonction `summarize()` du *package* `dplyr`, calculer :

(a) la moyenne et l'écart-type annuels des notes pour chacune des deux matières,

(b) la moyenne et l'écart-type annuels et par sexe des notes pour chacune des deux matières ;

2.5.18 En utilisant la fonction `gather()` du *package* `tidyr`, créer un tableau dans lequel chaque ligne renseigne le numéro d'étudiant, l'année, le prénom, le sexe, l'enseignement (macro ou stat) et la note ;

2.5.19 En repartant de l'objet obtenu à la question précédente, utiliser la fonction `spread()` du *package* `tidyr` pour retomber sur le même tableau que `notes`.

Exercice 2.6 : Manipulation de chaînes de caractères

2.6.1 Créer les objets `a` et `b` afin qu'il contiennent respectivement les chaînes de caractères suivantes : `23 à 0` et `C'est la piquette, Jack !` ;

2.6.2 Créer le vecteur `phrases` de longueur 2, dont les deux éléments sont `a` et `b` ;

2.6.3 À l'aide de la fonction appropriée dans le *package* `stringr`, afficher le nombre de caractètres de `a`, de `b`, puis appliquer la même fonction à l'objet `phrases` ;

2.6.4 En utilisant la fonction `str_c()`, concaténer `a` et `b` dans une seule chaîne de caractères, en choisissant la virgule comme caractère de séparation ;

2.6.5 Concaténer les deux éléments du vecteur `phrases` en une seule chaîne de caractères, en les séparant par le caractère de retour à la ligne, puis utiliser la fonction `cat()` pour afficher le résultat dans la console ;

2.6.6 Appliquer la même fonction que dans la question précédente à l'objet suivant : `c(NA, phrases)` et commenter ;

2.6.7 Mettre en majuscules, puis en minuscules les chaînes du vecteur `phrases` (afficher le résultat, ne pas modifier `phrases`) ;

2.6.8 À l'aide de la fonction `word()` du *package* `stringr`, extraire le mot `la`, puis `Jack` de la chaîne `b` ;

2.6.9 Même question que la précédente, en utilisant la fonction `str_sub()` ;

2.6.10 À l'aide de la fonction `str_detect()`, rechercher si le motif `piqu` puis `mauvais` sont présents dans `b` ;

2.6.11 À l'aide de la fonction `str_detect()`, rechercher si le motif `piqu` est présent dans les éléments du vecteur `phrases` ;

2.6.12 À l'aide de la fonction `str_detect()`, rechercher si le motif `piqu` ou le motif `à` sont présents dans les éléments du vecteur `phrases` ;

2.6.13 En utilisant la fonction `str_locate()`, retourner les positions de la première occurence du caractère `a` dans la chaîne `b`, puis essayer avec le caractère `w` pour observer le résultat retourné ;

2.6.14 Retourner toutes les positions du motif `a` dans la chaîne `b` ;

2.6.15 En utilisant la fonction `str_replace()`, remplacer la première occurence du motif `a`, par le motif `Z` (afficher le résultat, ne pas modifier `phrases`) ;

2.6.16 Remplacer toutes les occurences de `a` par `Z` dans la chaîne `b` (afficher le résultat, ne pas modifier `phrases`) ;

2.6.17 Utiliser la fonction `str_split()` pour séparer la chaîne `b` en utilisant la virgule comme séparateur de sous-chaînes ;

2.6.18 Retirer tous les caractères de ponctuation de la chaîne `b`, puis utiliser la fonction `str_trim()` sur le résultat pour retirer les caractères blancs du début et de la fin de la chaîne.

Exercice 2.7 : Manipulation de dates

2.7.1 En utilisant la fonction `as.Date()`, stocker la date du 29 août 2015 dans un objet que l'on appellera `d` puis afficher la classe de l'objet ;

2.7.2 À l'aide de la fonction appropriée, afficher la date du jour ;

2.7.3 À l'aide de la fonction `as.Date()`, stocker sous forme de date la chaîne de caractères suivante : `29-08-2015` ;

2.7.4 Utiliser les fonctions `as.POSIXct()` et `as.POSIXlt` pour stocker la chaîne de caractères `2015-08-29 20:30:56` sous forme de dates dans des objets nommés `d_ct` et `d_lt` respectivement ; utiliser ensuite la fonction `unclass()` sur les deux objets pour comparer la façon dont `R` a stocké l'information :

2.7.5 Utiliser la fonction appropriée du *package* `lubridate` pour stocker la chaîne de caractères `2015-08-29` sous forme de date ;

2.7.6 Même question avec la chaîne `2015-08-29 20:30:56` ;

2.7.7 Utiliser la fonction `ymd_hms()` pour stocker la date et l'heure actuelle, en précisant le fuseau horaire, puis afficher la date et l'heure correspondantes à New York City ;

2.7.8 Considérons le vecteur `x` :

```
x <- c(ymd_hms("2015-08-29 20:30:56", tz = "Europe/Paris"),
       ymd_hms("2015-09-15 08:10:33", tz = "Europe/Paris"))
```

Extraire l'année, le mois, le jour, les heures, les minutes et les secondes du premier élément de `x` à l'aide des fonctions appropriées du *package* `lubridate` ;

2.7.9 Appliquer les mêmes fonctions au vecteur `x` ;

2.7.10 Au premier élément de `x`, ajouter :
— une seconde,
— un jour,
— un mois
— deux années ;

2.7.11 Tester si la date du premier élément de `x` vient avant celle du second élément ;

2.7.12 En utilisant la fonction `new_interval()` du *package* `lubridate`, créer un intervalle de dates entre les deux éléments de `x`, puis afficher le nombre de jours, puis le nombre d'heures, puis le nombre d'années séparant les deux dates ;

2.7.13 En utilisant la fonction `seq()`, créer une séquence de dates avec un intervalle de 5 jours entre chaque date, commençant à la date du premier élément de `x` et se terminant à la date du second élément de `x` (la séquence sera tronquée avant) ;

2.7.14 Convertir en date les deux chaînes de caracères suivantes : `Sam 29 Août 2015` et `Sat 29 Aug 2015` ;

3

Fonctions

Jusqu'ici, nous avons utilisé des fonctions incluses dans les *packages*, rédigées par d'autres personnes. Dans ce chapitre, nous allons voir comment créer ses propres fonctions.

3.1 Définition

La définition d'une nouvelle fonction suit la syntaxe suivante :

```
name <- function(arguments) expression
```

avec `name` le nom que l'on décide de donner à la fonction, qui doit respecter les règles de nommage abordés en section 1.2.2.2, `arguments` les paramètres de la fonction, et `expression` le corps de la fonction. Comme on peut le remarquer, on utilise le symbole d'assignation : les fonctions sont des objets. L'appel de la fonction aura la syntaxe suivante :

```
name()
```

Il suffit donc de rajouter des parenthèses au nom de la fonction pour l'appeler. En effet, `name` désigne l'objet `R` qui contient la fonction qui est appelée à l'aide de l'expression `name()`.
Par exemple, si on souhaite définir la fonction qui calcule le carré d'un nombre, voici ce que l'on peut écrire :

```
carre <- function(x) x^2

# Le carré de 2
carre(2)

## [1] 4

# Le carré de -3
carre(-3)

## [1] 9
```

3.2 La structure d'une fonction

Les fonctions en `R`, excepté les fonctions primitives du *package base*, sont composées de trois parties :

- une liste de paramètres ;
- un corps, contenant du code exécuté lors de l'appel à la fonction ;
- un environnement, qui définit l'endroit où sont stockées les variables.

On peut accéder à ces trois parties (et les modifier) avec les fonctions `formals()` pour les paramètres, `body()` pour le corps et `environment()` pour l'environnement.

3.2.1 Le corps d'une fonction

Dans le cas le plus simple, le corps d'une fonction est constitué d'une seule instruction. Si on désire en écrire plusieurs, il est nécessaire de les entourner par des accolades, pour réaliser un regroupement. Le résultat est la valeur de la dernière commande contenue dans le corps de la fonction.

```
f <- function(x) {
  x^2
  y <- x
  y
}
f(2)

## [1] 2
```

Si on souhaite retourner une valeur autre part qu'à la dernière ligne, il faut utiliser la fonction `return()` (utile lorsque l'on emploie des conditions, comme nous le verrons par la suite, ou pour prévenir d'une erreur).

```
f <- function(x) {
  return(x^2)
  # Un commentaire de dernière ligne
}
f(2)

## [1] 4
```

Remarque 3.2.1

Certains conseillent de ne pas utiliser la fonction `return()` en dernière ligne, dans la mesure où cela est d'une part inutile, et perturbe la lecture du code d'autre part.

Il est possible de retourner une liste, pouvant contenir autant d'objet que l'on souhaite.

```
# Calculer la moyenne et l'écart-type pour un vecteur
stat_des <- function(x) {
  list(moyenne = mean(x), ecart_type = sd(x))
}
x <- runif(10)
stat_des(x)
```

```
## $moyenne
## [1] 0.4561442
##
## $ecart_type
## [1] 0.3047165
```

Il est également possible de ne pas afficher dans la console le résultat de l'appel à une fonction à l'aide de la fonction `invisible()`.

```
stat_des_2 <- function(x) {
  invisible(list(moyenne = mean(x), ecart_type = sd(x)))
}
x <- runif(10)
stat_des_2(x)
str(stat_des_2(x))

## List of 2
##  $ moyenne   : num 0.461
##  $ ecart_type: num 0.279

stat_des_2(x)$moyenne

## [1] 0.4606871
```

On peut afficher malgré tout le résultat d'une fonction retournant un résultat invisible en ayant recours aux parenthèses.

```
(stat_des_2(x))

## $moyenne
## [1] 0.4606871
##
## $ecart_type
## [1] 0.2793432
```

Lorsque la dernière instruction est une assignation, nous sommes dans le cas d'un résultat invisible.

```
f <- function(x){
  res <- x^2
}
f(2)
(f(2))

## [1] 4

x <- f(2)
x

## [1] 4
```

3.2.2 Les paramètres d'une fonction

Dans l'exemple de la fonction `carre()` que nous avons crée, nous avons renseigné un seul paramètre, appelé `x`. Si la fonction que l'on souhaite créer nécessite plusieurs paramètres, il faut les séparer par une virgule.

Considérons par exemple le problème suivant. Nous disposons d'une fonction de production $Y(L, K, M)$, qui dépend du nombre de travailleurs L et de la quantité de capital K, et du matériel M, telle que $Y(L, K, M) = L^{0.3}K^{0.5}M^2$. Cette fonction pourra s'écrire, en `R` de la manière suivante :

```
production <- function(l, k, m) l^(0.3) * k^(0.5) * m^(0.2)
```

3.2.2.1 Appel sans noms

En reprenant l'exemple précédent, si on nous donne $L = 60$ et $K = 42$ et $M = 40$, on peut en déduire la production :

```
production(60, 42, 40)

## [1] 46.28945
```

On peut noter que le nom des paramètres n'a pas été mentionné ici. Lors de l'appel de la fonction, `R` cherche d'abord s'il y a des paramètres nommés afin de leur associer des valeurs. S'il n'y a pas de nom, il se basera sur la position donnée aux paramètres.

```
production(k = 42, m = 40, l = 60)

## [1] 46.28945

production(k = 42, 60, 40)

## [1] 46.28945
```

3.2.2.2 Paramètres effectifs

On peut, lors de la définition de la fonction, choisir de donner une valeur par défaut aux paramètres. On parle de paramètre **formel** pour désigner les paramètres de la fonction (les variables utilisées dans le corps de la fonction) et de paramètre **effectif** pour désigner la valeur que l'on souhaite donner au paramètre formel. Pour définir la valeur à donner à un paramètre formel, on utilise le symbol d'égalité. Lors de l'appel de la fonction, si l'utilisateur ne définit pas explicitement une valeur, celle par défaut sera affectée.

```
# On propose de définir la valeur du capital à 42 par défaut
production_2 <- function(l, m, k = 42) l^(0.3) * k^(0.5) * m^(0.2)
production_2(l = 42, m = 40)

## [1] 41.59216

production_2(l = 42, m = 40, k = 2)

## [1] 9.076152
```

Dans l'exemple ci-avant, le paramètre à qui nous avons donné une valeur est placé en dernier. Ce n'est pas obligatoire, mais plus pratique, si le but recherché est de ne pas avoir à saisir le paramètre effectif lors de l'appel de la fonction. De plus, si l'utiliateur ne nomme pas les paramètres lors de l'appel, des problèmes liés à l'ordre peuvent apparaître...

```
production_3 <- function(l, k = 42, m) l^(0.3) * k^(0.5) * m^(0.2)
production_3(l = 42, m = 40)

## [1] 41.59216

production_3(42, 40)

## Error in production_3(42, 40): l'argument "m" est manquant, avec aucune valeur
par défaut
```

3.2.2.3 Appel avec des noms partiels

Par ailleurs, il est possible de ne pas saisir le nom complet des paramètres lors de l'appel d'une fonction. En effet, on peut utiliser une abréviation du nom du paramètre. S'il existe une ambiguïté, R retourne un message d'erreur.

```
f <- function(premier, second, troisieme) premier + second + troisieme
f(p = 1, s = 2, t = 3)

## [1] 6

# Problème d'ambiguïté
f <- function(texte, nombre, nom) print(nom)
f("hello", 2, no = 3)

## Error in f("hello", 2, no = 3): l'argument 3 correspond à plusieurs arguments
formels
```

Remarque 3.2.2

Si parmi les paramètres de la fonction, figure le paramètre "..." (plus de détails sont donnés dans la section 3.2.2.5), il n'est pas possible d'utiliser les abréviations.

3.2.2.4 Fonctions sans paramètres

On est parfois amené à créer des fonctions qui ne prennent pas de paramètre. Il suffit alors de laisser la liste de paramètres formels vide.

```
f <- function() sample(letters, size = 10, replace = TRUE)
f()

##  [1] "b" "j" "n" "z" "t" "i" "d" "m" "t" "p"
```

3.2.2.5 La paramètre spécial ...

Le paramètre ... que l'on peut voir dans certaines fonctions (essayez d'évaluer `sum` par exemple) sert à indiquer que la fonction peut admettre d'autres paramètres que ceux qui ont été définis. Cela sert à, dans la plupart des cas, à passer un paramètre à une autre fonction contenue dans le corps de la fonction que l'on appelle.

```
f <- function(...) names(list(...))
f(premier = 1, second = 2)

## [1] "premier" "second"
```

Attention toutefois, l'utilisation de ... peut induire des soucis. En effet, un paramètre mal écrit sera passé à ... et il n'y aura pas d'erreur de retournée. Par ailleurs, tous les paramètres placés après ... doivent être complètement nommés, pas abrégés.

```
sum(3, NA, 4, na.rm = TRUE)

## [1] 7

sum(3, NA, 4, an.rm = TRUE) # Mauvaise écriture

## [1] NA

sum(3, NA, 4, na = TRUE)  # Abréviation

## [1] NA
```

3.2.3 Portée des fonctions

Lorsque une fonction est appelée, le corps de cette fonction est interprété. Les variables ayant été définies dans le corps de la fonction ne vivent qu'à l'intérieur de celle-ci à moins d'avoir spécifié le contraire. On parle alors de **portée** des variables. Ainsi, une variable ayant une portée locale — c'est-à-dire vivant uniquement à l'intérieur du corps de la fonction — peut avoir le même nom qu'une variable globale — c'est à dire définie dans l'espace de travail de la session —, sans pour autant désigner le même objet, ou écraser cet objet.

```
# Définition d'une variable globale
valeur <- 1

# Définition d'une variable locale à la fonction f
f <- function(x){
  valeur <- 2
  nouvelle_valeur <- 3
  print(paste0("valeur vaut : ",valeur))
  print(paste0("nouvelle_valeur vaut : ",valeur))
  x + valeur
}

f(3)

## [1] "valeur vaut : 2"
```

```
## [1] "nouvelle_valeur vaut : 2"
## [1] 5

# valeur n'a pas été modifiée
valeur

## [1] 1

# nouvelle_valeur n'existe pas en dehors de f()
nouvelle_valeur

## Error in eval(expr, envir, enclos): objet 'nouvelle_valeur' introuvable
```

Sans trop rentrer trop dans les détails, il semble important de connaître quelques principes à propos de la portée des variables. Les variables sont définies dans des environnements, qui sont embriqués les uns dans les autres. Si une variable n'est pas définie dans le corps d'une fonction, R ira chercher dans un environnement parent.

```
valeur <- 1
f <- function(x){
  x + valeur
}
f(2)

## [1] 3
```

Si on définit une fonction à l'intérieur d'une autre fonction, et qu'on appelle une variable non définie dans le corps de cette fonction, R ira chercher dans l'environnement directement supérieur. S'il ne trouve pas, il ira chercher dans l'environnement encore supŕieur, et ainsi de suite.

```
# La variable valeur n'est pas définie dans g(). R va alors chercher dans f().
valeur <- 1
f <- function(){
  valeur <- 2
  g <- function(x){
    x + valeur
  }
  g(2)
}
f()

## [1] 4

# La variable valeur n'est définie ni dans g() ni dans f()
# mais dans l'environnement supérieur (global, ici)
valeur <- 1
f <- function(){
  g <- function(x){
    x + valeur
  }
  g(2)
```

```
}
f()

## [1] 3
```

Si on définit une variable dans le corps d'une fonction et que l'on souhaite qu'elle soit accessible dans l'environnement global, on peut utiliser le symbole `«-`, ou bien la fonction `assign`.

```
rm(x)
f <- function(x) {
    x <<- x + 1
}
f(1)
x

## [1] 2

# En utilisant assign
rm(x)
f <- function(x) {
    # envir = .GlobalEnv signifique que l'on veut définir dans l'environnement
    # global
    assign(x = "x", value = x + 1, envir = .GlobalEnv)
}
f(4)
x

## [1] 5
```

3.3 Exercices

Exercice 3.1 : création de fonctions

3.1.1 Créer une fonction nommée `somme_n_entiers` qui retourne la somme des n premiers entiers. Son seul paramètre sera `n` ;

3.1.2 Utiliser la fonction `somme_n_entiers()` pour calculer la somme des 100 premiers entiers ;

3.1.3 Terminer la fonction par l'assignation du résultat dans un objet nommé `res`, puis évaluer l'expression suivante : `somme_n_entiers(100)`. Que peut-on constater ?

3.1.4 Charger les données `diamonds` du *package* `ggplot2` dans la session `R` à l'aide de l'expression suivante :

```
data(diamonds, package = "ggplot2")
```

Créer une fonction que l'on appellera `prix_diamant_coupe()`, qui, quand on lui fournit la valeur de la coupe du diamant sous forme de caractères (`Fair`, `Good`, `Very Good`, `Premium`, ou `Ideal`), filtre le tableau de données `diamonds` pour ne conserver que les observations pour lesquelles la coupe du diamant correspond à celle indiquée en paramètre, et retourne le prix moyen des observations de la base ainsi filtrée ;

3.1.5 Reprendre le code de la fonction précédente, et le modifier pour retourner à présent une liste de deux éléments : (i) la moyenne des prix et (ii) l'écart-type ;

3.1.6 Créer la fonction `resume_diamant_coupe_couleur()`, qui pour une coupe et une couleur de diamant données, retourne une liste de deux éléments : (i) la moyenne des prix et (ii) l'écart-type pour les diamants possédant cette coupe et cette couleur (la couleur du diamant est une lettre allant de `J` pour les pires, à `D` pour les meilleurs). Tester la fonction pour la coupe `Fair` et la couleur `D` ;

3.1.7 Reprendre la fonction précédente, et lui attribuer la valeur `D` (en chaîne de caractères) comme paramètre effectif pour la couleur. Tester alors l'appel à la fonction en précisant :

(a) la coupe `Fair` et la couleur `D`,

(b) la coupe `Fair`, mais pas de paramètre pour la couleur,

(c) la coupe `Fair` et la couleur `E`,

(d) la coupe non précisée mais la couleur `E` ;

3.1.8 Soit le code suivant :

```r
# @x : (int)
f_test <- function(x){
  x^2
}# Fin de f_test()

# @x : (int)
f_test_2 <- function(y){
  x^2
}# Fin de f_test_2()

# @x : (int)
f_test_3 <- function(y){
  x <- y
  x^2
}# Fin de f_test_3()


x <- 3
```

Expliquez ce qui se passe dans chacun des cas suivants :

```r
f_test(x = 2)
```

```r
f_test_2(y = 2)
```

```r
f_test_2()
```

```r
f_test_3(4)
x
```

Exercice 3.2 : création de fonctions

Supposons que les adresses e-mails des étudiants de l'Université de Rennes 1 sont constituées de la manière suivante : le prénom et le nom de famille séparés par un point, le symbole arobase et le enfin le nom de domaine. Supposons de plus que les étudiants ont un seul prénom, et aucune particule au nom de famille. La syntaxe des adresses e-mail est donc comme suit : `nom.prenom@etudiant.univ-rennes1.fr`.

```
emails <- c("marie.petit@etudiant.univ-rennes1.fr",
            "jean.dupont@etudiant.univ-rennes1.fr",
            "isabelle.martinez@etudiant.univ-rennes1.fr",
            "pierre.moreau@etudiant.univ-rennes1.fr")
```

Créer une fonction, qui à partir d'**une** adresse e-mail d'un étudiant, retourne un `data.frame` contenant trois variables : le prénom, le nom et l'adresse e-mail de cet étudiant.

4

Boucles et calculs vectoriels

Il existe deux sortes de boucles dans `R`. Celles pour lesquelles les itérations continuent tant qu'une condition n'est pas invalidée (`while()`), et celles pour lesquelles le nombre d'itérations est défini au moment de lancer la boucle (`for()`).

Avant de présenter chacune de ces fonctions, il est nécessaire de préciser que les boucles ne sont pas le point fort de `R`. Dès que l'on souhaite appliquer une fonction à chaque élément d'un vecteur, et/ou que le résultat de chaque itération ne dépend pas de l'itération précédente, il est préférable de vectoriser les calculs (voir section 4.2).

4.1 Les boucles

4.1.1 Les boucles avec `while()`

Quand on souhaite répéter un calcul tant qu'une condition est satisfaite, on utilise la fonction `while()`, avec la syntaxte suivante :

```
while(condition) instruction
```

avec `condition` un logique, comme vu dans la section 2.4.1.3, et `instruction` du code, qui peut être entouré d'accolades si on souhaite évaluer plusieurs instructions.

```
x <- 100
while(x/3 > 1){
  x <- x/3
}
x/3 > 1

## [1] FALSE

x

## [1] 1.234568
```

4.1.2 Les boucles avec `for()`

Quand on connaît le nombre d'itérations à l'avance, on peut utiliser la fonction `for()`. La syntaxe est la suivante :

```
for(variable in vector) instruction
```

avec `variable` le nom d'une variable locale à la fonction `for()`, `vector` un vecteur à `n` éléments définissant les valeurs que prendra `variable` pour chacun des `n` tours, et `instruction` le code à exécuter à chaque itération.

```
for(nom in  c("Sonia", "Anne-Hélène", "Julien-Yacine")) print(nom)

## [1] "Sonia"
## [1] "Anne-Hélène"
## [1] "Julien-Yacine"
```

On peut utiliser la fonction `for()` pour remplir les éléments d'une liste, ou d'un vecteur. À chaque itération, `R` doit trouver le vecteur de destination en mémoire, créer un nouveau vecteur qui permettra de contenir plus de données, copier données depuis l'ancien vecteur pour les insérer dans le nouveau, et enfin supprimer l'ancien vecteur (Ross, 2014). C'est une opération coûteuse en temps. Un moyen de rendre cette allocation plus efficace est de créer *a priori* le vecteur ou la liste en le remplissant avec des données manquantes. Ainsi, `R` n'aura pas besoin de ré-allouer la mémoire à chaque itération.

```
# Mauvaise manière
resultat <- NULL
for(i in seq_len(3)) {
  resultat[i] <- i
}
resultat

## [1] 1 2 3

# Manière plus économique
resultat <- rep(NA, 3)
for(i in seq_len(3)) {
  resultat[i] <- i
}
resultat

## [1] 1 2 3
```

4.1.3 Les conditions

On peut soumettre l'exécution de codes en `R` à conditions que certaines conditions soient honorées.

4.1.3.1 Les instructions `if ... else`

Les instructions `if` et `else` fournissent un moyen d'exécuter du code si une condition est respectée ou non. La syntaxe prend deux formes :

```
# Première forme (pas de code si condition == FALSE)
if (condition) instruction

# Seconde forme
if (condition) instruction si vrai else instruction si faux
```

avec `condition` un logique et `instruction` du code à évaluer si la condition est satisfaite. À nouveau, on peut avoir recours aux accolades pour créer des regroupements.

```
# Simple condition
x <- 2
if(x == 2) print("Hello")

## [1] "Hello"

x <- 3
if(x == 2) print("Hello")

# Avec des instructions dans le cas contraire
if(x == 2) print("Hello") else print("x est différent de 2")

## [1] "x est différent de 2"

if(x == 2){
  print("Hello")
} else {# x != 2
  x <- x-1
  print(paste0("La nouvelle valeur de x : ", x))
}

## [1] "La nouvelle valeur de x : 2"
```

Remarque 4.1.1

Attention, lorsque l'on fait des regroupements et qu'on utilise la structure `if` et `else`, il est nécessaire d'écrire le mot `else` sur la même ligne que la parenthèse fermante du groupe d'instructions à réaliser dans le de la condition du `if` vérifiée.

4.1.3.2 La fonction `switch`

Avec la fonction `switch()`, on peut indiquer à `R` d'exécuter un code en fonction du résultat obtenu lors d'un test. La syntaxe est la suivante :

```
switch(valeur_test,
       cas_1 = {
         instruction_cas_1
       },
       cas_2 = {
         instruction_cas_2
       },
       ...
  )
```

avec `valeur_test` un nombre ou une chaîne de caractères. Si `valeur_test` vaut `cas_1`, alors uniquement `instruction_cas_1` sera évaluée, si `valeur_test` vaut `cas_2`, alors ce sera `instruction_cas_2` qui le sera, et ainsi de suite. On peut rajouter une valeur par défaut en utilisant la syntaxte suivante :

```
switch(valeur_test,
       cas_1 = {
         instruction_cas_1
       },
       cas_2 = {
         instruction_cas_2
       },
       {
        instruction_defaut
       }
  )
```

Voici un exemple d'utilisation, issu de la page d'aide de la fonction.

```
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x))
}
x <- rcauchy(10)
centre(x, "mean")

## [1] -9.890611

centre(x, "median")

## [1] 0.4850023
```

4.1.4 L'instruction `repeat ...`, `break`

L'instruction `repeat`, ... `break` permet de répéter une expression. Il est nécessaire d'ajouter un test d'arrêt, à l'aide de l'instruction `break`.

```
i <- 1
repeat {
  i <- i + 1
  if(i == 3) break
}
i

## [1] 3
```

4.1.5 L'instruction `next ...`, `break`

L'instruction `next` ... `break` autorise de passer immédiatement à l'itération suivante d'une boucle `for`, `while` ou `repeat`

```
resul <- rep(NA, 10)
for(i in 1:10) {
  if(i == 5) next
  resul[i] <- i
}
# Le 5e élément de resul est resté non-disponible
resul

##  [1]  1  2  3  4 NA  6  7  8  9 10
```

4.1.6 Barre de progression

Lorsque l'exécution d'une boucle prend du temps, il peut être intéressant d'avoir une idée de l'état d'avancement des itérations. Pour cela, il est bien sûr possible d'afficher une valeur dans la console à chaque tour, chaque 10 tours, etc.
La fonction `txtProgressBar()` du *package* `utils` permet un affichage d'une barre de progression dans la console. Il suffit de lui fournir une valeur minimale et maximale, et de la mettre à jour à chaque itération. Le paramètre `style` autorise de surcroit à choisir un "style" pour la barre. Le style numéro 3 affiche un pourcentage de progression, et est utile lorsque d'autres résultats sont affichés dans la console lors de l'exécution de la boucle, dans la mesure où la barre est de nouveau affichée au complet dans la console si nécessaire.
Dans l'exemple qui suit, à chacun des dix tours, une pause 0.1 seconde est effectuée, puis la barre de progression est mise à jour.

```
nb_tours <- 10
p_b <- txtProgressBar(min = 1, max = nb_tours, style = 3)
for(i in 1:nb_tours){
  Sys.sleep(0.1)
  setTxtProgressBar(p_b, i)
}
```

Si l'exécution est vraiment longue, et qu'on est impatient de connaître les résultats, il existe de plus une fonction amusante dans le *package* `beepr`, qui porte le nom de `beep()`. Plusieurs sons peuvent être utilisés (voir la page d'aide de la fonction).

```
library(beepr)
beep("mario")
```

4.2 La vectorisation

Comme indiqué plus haut, les boucles sont des opérations lentes en `R`. Il est cependant possible, dans de nombreux cas, d'éviter de les employer, en ayant recours à la vectorisation : au lieu d'appliquer une fonction à un scalaire, on l'applique à un vecteur. En fait, nous avons déjà eu recours à maintes reprises aux calculs vectoriels. En effet, lorsque nous avons procédé à des additions, des multiplications, etc. sur des vecteurs, nous avons effectué des calculs vectoriels.

Empruntons un exemple à Burns (2011) : dans des langages comme le `C`, pour effectuer la somme des logarithmes naturels des n premiers entiers, voici une manière de faire :

```
# Somme des logarithmes des 10 premiers entiers
somme_log <- 0
for(i in seq_len(10)){
  somme_log <- somme_log + log(i)
}
somme_log

## [1] 15.10441
```

Il est possible d'obtenir le même résultat, à la fois d'une manière plus élégante, mais surtout plus efficace en vectorisant le calcul :

```
sum(log(seq_len(10)))

## [1] 15.10441
```

Derrière ce code, la fonction `log()` applique la fonction logarithme sur toutes les valeurs du vecteur donné en paramètre. La fonction `sum()`, quant à elle, se charge d'additionner tous les éléments du vecteur qui lui est donné en paramètre. Ces deux fonctions utilisent la vectorisation, mais d'une manière différente : la fonction `log()` applique une opération à chaque élément d'un vecteur, tandis que la fonction `sum()` produit un résultat basé sur la totalité du vecteur. L'avantage d'utiliser des fonctions vectorielles plutôt que d'écrire une boucle pour effectuer le calcul, est que ces premières font appel à des fonctions rédigées en `C` ou `FORTRAN`, qui utilisent aussi des boucles, mais comme ce sont des langages compilés et non pas interprétés, les itérations sont réalisées dans un temps réduit.

Il existe des fonctions, rédigées en `C` qui effectuent des boucles `for`. On leur donne souvent le nom de "*fonctions de la famille apply*". Il ne s'agit pas de la vectorisation, mais ces fonctions sont souvent mentionnées dès que l'on parle de ce sujet. Ce sont des fonctionnelles qui prennent une fonction en *input* et retournent un vecteur en *output* (Wickham, 2014). Ces fonctions sont très utilisées, mais elles souffrent d'un manque d'uniformité. En effet, elles ont été rédigées par des personnes différentes, ayant chacune leur convention. Le *package* `plyr` remédie à ce problème, et ajoute par la même occasion des fonctions supplémentaires, pour couvrir plus de cas que les "*fonctions de la famille apply*".

Nous allons donc présenter dans un premier temps les fonctions du *package* `plyr`. Les fonctions du même type du *package* `base` seront tout de même présentées par la suite.

4.2.1 Les fonctions du package `plyr`

Les fonctions que nous allons aborder dans cette section possèdent des noms faciles à se remémorer : la première lettre correspond au format d'entrée des données, la seconde au format de sortie souhaité, et la fin du nom se termine par le suffixe `ply`. Ainsi, la fonction `llapply()` prend en entrée une liste, effectue une opération sur les éléments, et retourne une liste (Anderson, 2012). Les différentes fonctions que nous allons passer en revue sont consignées dans le tableau ci-après, où les lignes correspondent aux formats d'entrée, et les lignes aux formats de sortie. Pour y avoir accès, il faut charger le *package* :

```
library(plyr)
```

	array	data frame	list
array	aaply()	adply()	alply()
data frame	daply()	ddply()	dlply()
list	laply()	ldply()	llply()

Remarque 4.2.1

Il est possible d'avoir plusieurs paramètres en *input* au lieu d'un seul objet. Les fonctions `mlply()`, `mdply()` et `maply()` s'en chargent. Si à la place du `m`, la première lettre est un `r`, il s'agit alors de fonction de réplications. Enfin, si la seconde lettre est un trait de soulignement (\), alors le résultat retourné n'est pas affiché (le code utilise la fonction `invisible()`).

Tous les paramètres de ces fonctions commencent par un point (`.`), afin d'éviter des incompatibilités avec la fonction à appliquer.

4.2.1.1 *Array* en *input* : `a*ply()`

Les fonctions `aaply()`, `adply()` et `alply()` appliquent une fonction à chaque portion d'un *array* et ensuitent joignent le résultat sous forme d'un *array*, d'un *data frame* ou d'une *list* respectivement.

Le paramètre `.margins` détermine la manière de découper le tableau. Il y en a quatre pour un tableau en deux dimensions :

1. `.margins = 1` : par lignes ;
2. `.margins = 2` : par colonnes ;
3. `.margins = c(1,2)` : par cellule ;
4. `.margins = c()` : ne pas faire de découpement ;

Pour un tableau en trois dimensions, il y a trois découpages possibles en deux dimensions, trois en une dimension et une en zéro dimension (voir (Wickham, 2011)) au besoin.

```
tableau <- array(1:24, dim = c(3, 4, 2),
                 dimnames = list(ligne = letters[1:3],
                                 colonne = LETTERS[1:4],
                                 annee = 2001:2002))
tableau
```

```
## , , annee = 2001
##
##      colonne
## ligne A B C  D
##     a 1 4 7 10
##     b 2 5 8 11
##     c 3 6 9 12
##
## , , annee = 2002
##
##      colonne
## ligne  A  B  C  D
```

```
##     a 13 16 19 22
##     b 14 17 20 23
##     c 15 18 21 24
```

```
# La moyenne des valeurs pour chaque ligne
aaply(tableau, 1, mean) # résultat sous forme de tableau
```

```
##    a    b    c
## 11.5 12.5 13.5
```

```
adply(tableau, 1, mean) # résultat sous forme de liste
```

```
##   ligne   V1
## 1     a 11.5
## 2     b 12.5
## 3     c 13.5
```

```
alply(tableau, 1, mean) # résultat sous forme de liste
```

```
## $`1`
## [1] 11.5
##
## $`2`
## [1] 12.5
##
## $`3`
## [1] 13.5
##
## attr(,"split_type")
## [1] "array"
## attr(,"split_labels")
##   ligne
## 1     a
## 2     b
## 3     c
```

```
# La moyenne des valeurs pour chaque longitude
# en ne simplifiant pas le résultat
aaply(tableau, 2, mean, .drop = FALSE)
```

```
##
## colonne  1
##       A  8
##       B 11
##       C 14
##       D 17
```

```
# Par lignes et colonnes
aaply(tableau, c(1,2), mean)
```

```
##      colonne
## ligne A  B  C  D
##     a 7 10 13 16
##     b 8 11 14 17
##     c 9 12 15 18
```

```
adply(tableau, c(1,2), mean)

##    ligne colonne V1
## 1      a       A  7
## 2      b       A  8
## 3      c       A  9
## 4      a       B 10
## 5      b       B 11
## 6      c       B 12
## 7      a       C 13
## 8      b       C 14
## 9      c       C 15
## 10     a       D 16
## 11     b       D 17
## 12     c       D 18

# L'affichage prend beaucoup de place
# alply(tableau, c(1,2), mean)

# Avec une fonction définie par l'utilisateur
standardise <- function(x) (x - min(x)) / (max(x) - min(x))
# Standardiser les valeurs par colonne
aaply(tableau, 2, standardise)

## , , annee = 2001
##
##        ligne
## colonne a          b         c
##       A 0 0.07142857 0.1428571
##       B 0 0.07142857 0.1428571
##       C 0 0.07142857 0.1428571
##       D 0 0.07142857 0.1428571
##
## , , annee = 2002
##
##        ligne
## colonne         a         b c
##       A 0.8571429 0.9285714 1
##       B 0.8571429 0.9285714 1
##       C 0.8571429 0.9285714 1
##       D 0.8571429 0.9285714 1
```

4.2.1.2 *Data frame* en *input* : `d*ply()`

En économétrie, les *data frames* sont très présents. Aussi, la connaissance des fonction `daply()`, `ddply()` et `dlply()` est importante. En effet, elles sont très utiles pour appliquer des fonctions à des groupes basés sur des combinaisons de variables.

Avec les fonctions `d*ply()`, il est nécessaire d'indiquer quelles variables, ou fonctions de variables on souhaite utiliser, en l'indiquant au paramètre `.variables`. Elles peuvent être contenue dans le *data frame* fourni au paramètre `.data`, ou bien provenir de l'environnement global. `R` cherchera

dans un premier temps si la variable est contenue dans le *data frame*, et, s'il ne trouve pas, ira chercher dans l'environnement global.
Pour indiquer que l'on désire faire le regroupement selon une variable — mettons `variable_1` — il faudra fournir l'expression `.(variable_1)` au paramètre `.variables`. Si on souhaite effectuer les regroupement selon les interactions de plusieurs variables — `variable_1`, `variable_2` et `variable_3`, il faut alors utiliser l'expression suivante : `.(variable_1, variable_2, variable_3)`.

```
chomage <- data.frame(region = rep(c(rep("Bretagne", 4), rep("Corse", 2)), 2),
                      departement = rep(c("Cotes-d'Armor", "Finistere",
                                          "Ille-et-Vilaine", "Morbihan",
                                          "Corse-du-Sud", "Haute-Corse"), 2),
                      annee = rep(c(2011, 2010), each = 6),
                      ouvriers = c(8738, 12701, 11390, 10228, 975, 1297,
                                   8113, 12258, 10897, 9617, 936, 1220),
                      ingenieurs = c(1420, 2530, 3986, 2025, 259, 254,
                                     1334, 2401, 3776, 1979, 253, 241))

chomage
```

```
##      region     departement annee ouvriers ingenieurs
## 1  Bretagne   Cotes-d'Armor  2011     8738       1420
## 2  Bretagne       Finistere  2011    12701       2530
## 3  Bretagne Ille-et-Vilaine  2011    11390       3986
## 4  Bretagne        Morbihan  2011    10228       2025
## 5     Corse    Corse-du-Sud  2011      975        259
## 6     Corse     Haute-Corse  2011     1297        254
## 7  Bretagne   Cotes-d'Armor  2010     8113       1334
## 8  Bretagne       Finistere  2010    12258       2401
## 9  Bretagne Ille-et-Vilaine  2010    10897       3776
## 10 Bretagne        Morbihan  2010     9617       1979
## 11    Corse    Corse-du-Sud  2010      936        253
## 12    Corse     Haute-Corse  2010     1220        241
```

```
# Total chomeurs en Bretagne et en Corse pour les années 2010 et 2011
# Sous forme de data frame
ddply(chomage, .(annee), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
##   annee total_chomeurs
## 1  2010          53025
## 2  2011          55803
```

```
# Sous forme de tableau
daply(chomage, .(annee), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
## $`2010`
## [1] 53025
##
## $`2011`
## [1] 55803
```

```
# Sous forme de liste
dlply(chomage, .(annee), summarise, total_chomeurs = sum(ouvriers + ingenieurs))
```

```
## $`2010`
##   total_chomeurs
## 1          53025
##
## $`2011`
##   total_chomeurs
## 1          55803
##
## attr(,"split_type")
## [1] "data.frame"
## attr(,"split_labels")
##   annee
## 1  2010
## 2  2011
```

```
# Total chomeurs pour les années 2010 et 2011, par région du data frame
ddply(chomage, .(annee, region), summarise,
      total_chomeurs = sum(ouvriers + ingenieurs))
```

```
##   annee   region total_chomeurs
## 1  2010 Bretagne          50375
## 2  2010    Corse           2650
## 3  2011 Bretagne          53018
## 4  2011    Corse           2785
```

```
# Nombre d'observations pour chaque groupe
ddply(chomage, .(annee, region), nrow)
```

```
##   annee   region V1
## 1  2010 Bretagne  4
## 2  2010    Corse  2
## 3  2011 Bretagne  4
## 4  2011    Corse  2
```

```
# En utilisant une fonction définie par l'utilisateur
ddply(chomage, .(annee, region), function(x){
  moy_ouvriers <- mean(x$ouvriers)
  moy_ingenieurs <- mean(x$ingenieurs)
  data.frame(moy_ouvriers = moy_ouvriers, moy_ingenieurs = moy_ingenieurs)
})
```

```
##   annee   region moy_ouvriers moy_ingenieurs
## 1  2010 Bretagne     10221.25        2372.50
## 2  2010    Corse      1078.00         247.00
## 3  2011 Bretagne     10764.25        2490.25
## 4  2011    Corse      1136.00         256.50
```

4.2.1.3 *List* en *input* : `l*ply()`

Les fonctions du type `l*ply()` prennent une liste en entrée. Il n'y a donc pas de paramétrage à effectuer pour choisir un découpage, il est déjà fait.

```r
set.seed(1)
liste <-list(normale =rnorm(10), logiques =c(TRUE, TRUE, FALSE), x =c(0,NA, 3))

# Obtenir la longueur de chaque élément de la liste
laply(liste, length)

## [1] 10  3  3

ldply(liste, length)

##        .id V1
## 1  normale 10
## 2 logiques  3
## 3        x  3

llply(liste, length)

## $normale
## [1] 10
##
## $logiques
## [1] 3
##
## $x
## [1] 3

# Calculer la moyenne pour chaque élément
unlist(llply(liste, mean, na.rm = TRUE))

##   normale  logiques         x
## 0.1322028 0.6666667 1.5000000

# Appliquer une fonction définie par l'utilisateur
llply(liste,function(x, y) x /mean(x, na.rm = TRUE) + y, y = 2)

## $normale
##  [1] -2.7385827  3.3891033 -4.3208096 14.0669232  4.4924421 -4.2061356
##  [7]  5.6869803  7.5847895  6.3552892 -0.3099997
##
## $logiques
## [1] 3.5 3.5 2.0
##
## $x
## [1]  2 NA  4
```

4.2.1.4 Calcul parallèle

En utilisant plusieurs processeurs, on peut effectuer des calculs parallèles, ce qui accélère les calculs dans certains cas. En effet, quand il est possible de fractionner les opérations à effectuer en morceaux, on peut en réaliser une partie sur un processeur, une autre sur un second processeur, et ainsi de suite. Les résultats obtenus sont ensuite rassemblés avant d'être retournés. Le *package* `doMC` (ou `doSMP` sur Windows) peut être chargé pour utiliser la fonction de calcul parallèle

proposé par les fonctions `**ply()`. Il suffit de préciser le nombre de cœurs souhaité en faisant appel à la fonction `registerDoMC()`, et de fixer la valeur `TRUE` au paramètre `.parallel` de la fonction `**ply()`.

```
df <- data.frame(valeur_1 = rnorm(12000), id = rep(month.abb, each = 1000))

# Fonction lente (et inutile)
f <- function(x){
  res <- rep(NA, nrow(x))
  for(i in 1:length(res)){
    res[i] <- mean(x[1:i, "valeur_1"],)
  }
  mean(res)
}

library(doMC)

## Loading required package: foreach
## Loading required package: iterators
## Loading required package: parallel

# Définir le parallel backend avec le package foreach
registerDoMC(cores=4)

system.time(ddply(df, .(id), f))

##    user  system elapsed
##   0.394   0.017   0.418

system.time(ddply(df, .(id), f, .parallel = TRUE))

##    user  system elapsed
##   0.605   0.257   0.379
```

4.2.2 Les fonctions de la famille `apply` du package `base`

Le tableau ci-après recense les fonctions principales *de la famille apply* du *package* `base`.

Fonction	*Input*	*Output*
`apply()`	Matrice ou tableau	Vecteur ou tableau ou liste
`lapply()`	Liste ou vecteur	Liste
`sapply()`	Liste ou vecteur	Vecteur ou matrice ou liste
`vapply()`	Liste ou vecteur	Vecteur ou matrice ou liste
`tapply()`	Tableau et facteurs	Tableau ou liste
`mapply()`	Listes et/ou vecteurs	Vecteur ou matrice ou liste

4.2.2.1 La fonction `lapply`

La fonction `lapply()` applique à chaque élément du premier paramètre qui lui est donné une fonction indiquée en second paramètre et retourne le résultat sous forme de liste. La syntaxe est la suivante :

```
lapply(X, FUN, ...)
```

avec `X` la liste ou le vecteur donné en paramètre sur lequel on désire appliquer la fonction `FUN`. La paramètre `...` permet comme expliqué dans la remarque 3.2.2, de fournir des paramètres à une fonction imbriquée, en l'occurance à celle que l'on souhaite appliquer à tous les éléments de `X`.

```
liste <- list(normale = rnorm(10), logiques = c(TRUE, TRUE, FALSE), x = c(0,
    NA, 3))

# Obtenir la liste des longueurs de chaque élément
lapply(liste, length)

## $normale
## [1] 10
##
## $logiques
## [1] 3
##
## $x
## [1] 3

# Calculer la moyenne pour chaque élément
lapply(liste, mean, na.rm = TRUE)

## $normale
## [1] 0.04319732
##
## $logiques
## [1] 0.6666667
##
## $x
## [1] 1.5
```

On peut créer une fonction à l'intérieur de l'appel à la fonction `lapply()`. Le premier paramètre est nécessairement un élément du vecteur auquel on souhaite appliquer la fonction.

```
lapply(liste, function(x) x / mean(x, na.rm = TRUE))

## $normale
##  [1] -17.375456  -6.647068  19.251983 -12.199524 -42.424943  24.224455
##  [7] -12.030570  24.388223  31.518789   1.294111
##
## $logiques
## [1] 1.5 1.5 0.0
##
## $x
## [1]  0 NA  2

# Si la fonction doit posséder plusieurs paramètres
lapply(liste, function(x, y) x / mean(x, na.rm = TRUE) + y, y = 2)
```

```
## $normale
##  [1] -15.375456  -4.647068  21.251983 -10.199524 -40.424943  26.224455
##  [7] -10.030570  26.388223  33.518789   3.294111
##
## $logiques
## [1] 3.5 3.5 2.0
##
## $x
## [1]  2 NA  4
```

On peut appliquer la fonction `lapply()` sur des tableaux de données, dans la mesure où ces derniers sont des listes. Cela s'avère pratique pour réaliser des opérations pour chaque colonne d'un tableau de données. Afin de prendre moins de place dans l'affichage, l'exemple suivant utilise la fonction `unlist()` pour aplatir la liste.

```
data(cars)

# Afficher le type de chaque colonne de la data frame "cars"
unlist(lapply(cars, class))

##     speed      dist
## "numeric" "numeric"

# Calculer la moyenne pour chaque colonne
unlist(lapply(cars, mean))

## speed  dist
## 15.40 42.98
```

Remarque 4.2.2

Attention, ce qui suit relève plus d'un tour de passe-passe que de la programmation élégante.

Si la fonction que l'on souhaite appliquer aux éléments de notre vecteur retourne un vecteur ligne de même longueur pour chaque élément, la fonction `do.call()` peut devenir un outil très pratique pour créer une *data frame*. Voyons-le à travers un exemple.

```
l <- lapply(1:3, function(x) cbind(valeur = x, lettre = LETTERS[x]))
data.frame(do.call("rbind", l))

##   valeur lettre
## 1      1      A
## 2      2      B
## 3      3      C
```

L'appel de `do.call("rbind", x)` revient à faire `rbind(x[1], x[2], ..., x[n])` avec `x` un vecteur de taile n.

4.2.2.2 La fonction `sapply`

La fonction `sapply()` applique une fonction aux éléments d'un vecteur ou d'une liste et peut retourner un vecteur, une liste ou une matrice. Elle possède la syntaxe suivante :

```
sapply(X, FUN, simplify, USE.NAMES)
```

où `X` est le vecteur ou la liste auquel on souhaite appliquer la fonction `FUN`. Lorsque `simplify` vaut `FALSE`, le résultat est retourné sous forme de liste, exactement comme `lapply()` (la fonction `sapply()` s'appuie sur la fonction `lapply()`). Lorsque `simplify` vaut `TRUE` (par défaut), le résultat est retourné dans une forme simplifiée, si cela est possible. Si tous les éléments retournés par la fonction `FUN` sont des scalaires, alors `sapply()` retourne un vecteur ; sinon, si les éléments retournés ont la même taille, `sapply()` retourne une matrice avec une colonne pour chaque élément de `X` auquel la fonction `FUN` est appliquée. Le paramètre `USE.NAMES`, quand il vaut `TRUE` (par défaut), et si `X` est de type `character`, utilise `X` comme nom pour le résultat, à moins que le résultat possède déjà des noms.

```
(x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE)))

## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $beta
## [1]  0.04978707  0.13533528  0.36787944  1.00000000  2.71828183  7.38905610
## [7] 20.08553692
##
## $logic
## [1]  TRUE FALSE FALSE  TRUE

# Application de la fonction quantile() à chaque élément
# pour obtenir la médiane et les quartiles

# Avec lapply()
lapply(x, quantile)

## $a
##    0%   25%   50%   75%  100%
##  1.00  3.25  5.50  7.75 10.00
##
## $beta
##          0%         25%         50%         75%        100%
##  0.04978707  0.25160736  1.00000000  5.05366896 20.08553692
##
## $logic
##   0%  25%  50%  75% 100%
##  0.0  0.0  0.5  1.0  1.0

# Avec sapply
sapply(x, quantile)

##          a        beta logic
## 0%    1.00  0.04978707   0.0
## 25%   3.25  0.25160736   0.0
## 50%   5.50  1.00000000   0.5
## 75%   7.75  5.05366896   1.0
## 100% 10.00 20.08553692   1.0
```

```r
# Exemple avec USE.NAMES
sapply(LETTERS[1:3], nchar)

## A B C
## 1 1 1

sapply(LETTERS[1:3], nchar, USE.NAMES = FALSE)

## [1] 1 1 1
```

4.2.2.3 La fonction `vapply`

La fonction `vapply()` est similaire à `sapply()`, mais elle possède un type de valeurs spécifié, ce qui peut rendre l'utilisation plus sûre (et parfois plus rapide). Lorsqu'on lui fournit un *data frame*, `vapply()` retourne le même résultat que `sapply()`. Cependant, quand on lui fournit une liste vide, `vapply()` retourne un vecteur logique de longueur nulle (ce qui est plus sensé que la liste vide que returne `sapply()`).

```r
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES)
```

avec `X`, `FUN`, `...` et `USE.NAMES` les mêmes paramètres que pour `sapply()`. Le paramètre `FUN.VALUE` doit être un vecteur, un masque pour la valeur retournée par la fonction de `FUN`.

```r
# Retourner le vecteur
sapply(cars, is.numeric)

## speed  dist
##  TRUE  TRUE

vapply(cars, is.numeric, FUN.VALUE = logical(1))

## speed  dist
##  TRUE  TRUE

# Avec la liste vide
sapply(list(), is.numeric)

## list()

vapply(list(), is.numeric, FUN.VALUE = logical(1))

## logical(0)
```

4.2.2.4 La fonction `apply`

La fonction `apply()` possède la syntaxe suivante :

```r
apply(X, MARGIN, FUN, ...)
```

avec `X` une matrice ou un tableau, `MARGIN` indiquant si on souhaite appliquer la fonction `FUN` aux lignes (`MARGIN = 1`) ou aux colonnes (`MARGIN = 2`), et `...` des paramètres supplémentaires éventuels à passer à la fonction `FUN`.

```r
(X <- matrix(1:9, ncol = 3))
```

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

```r
# Somme par ligne
apply(X, MARGIN = 1, sum)
```

```
## [1] 12 15 18
```

```r
# Somme par colonne
apply(X, MARGIN = 2, sum)
```

```
## [1]  6 15 24
```

```r
# Fonction définie par l'utilisateur
apply(X, MARGIN = 1, function(x) sum(x) / sum(X))
```

```
## [1] 0.2666667 0.3333333 0.4000000
```

4.2.2.5 La fonction `tapply`

La fonction `tapply()` s'applique à chaque cellule d'un tableau, sur des regroupements définis par les variables catégorielles fournies. La syntaxe est la suivante :

```r
tapply(X,INDEX, FUN, ..., simplify)
```

avec `X` le taleau de données, `INDEX` une liste d'un ou plusieurs facteurs, chacun de même taille que `X`. Le paramètre `FUN` renseigne la fonction que l'on souhaite appliquer. Si `simplify` vaut `FALSE`, le résultat est un tableau de mode `list`. Sinon (par défaut), le résultat est un tableau de scalaires.

```r
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
# Moyenne de la longueur des sépales par espèce
tapply(iris$Sepal.Length, iris$Species, mean)
```

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

```r
# Pour retourner le résultat sous forme de liste
tapply(iris$Sepal.Length, iris$Species, mean, simplify = FALSE)
```

```
## $setosa
## [1] 5.006
##
## $versicolor
## [1] 5.936
##
## $virginica
## [1] 6.588
```

4.2.2.6 La fonction `mapply`

La fonction `mapply()` applique une fonction à plusieurs listes ou vecteurs. La syntaxe est la suivante :

```
mapply(FUN, ..., MoreArgs, SIMPLIFY, USE.NAMES)
```

avec `FUN` la fonction à appliquer aux vecteurs ou listes fournies (grâce à `...`), `MoreArgs` une liste de paramètres supplémentaires à fournir à la fonction à appliquer. Les paramètres `SIMPLIFY` et `USE.NAMES` ont le même usage que pour la fonction `sapply()`.

```
(l1 <- list(a = c(1:5), b = c(6:10)))

## $a
## [1] 1 2 3 4 5
##
## $b
## [1]  6  7  8  9 10

(l2 <- list(c = c(11:15), d = c(16:20)))

## $c
## [1] 11 12 13 14 15
##
## $d
## [1] 16 17 18 19 20

# La somme des éléments correspondants de l1 et l2
mapply(sum, l1$a, l1$b, l2$c, l2$d)

## [1] 34 38 42 46 50

# Attention au recyclage silencieux !
(l1 <- list(a = c(1:5), b = c(6:20)))

## $a
## [1] 1 2 3 4 5
##
## $b
##  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

mapply(sum, l1$a, l1$b, l2$c, l2$d)

##  [1] 34 38 42 46 50 39 43 47 51 55 44 48 52 56 60
```

4.2.3 La fonction `Vectorize`

La fonction `Vectorize()` permet de convertir une fonction scalaire en une fonction vectorielle. Attention, cela ne permet pas d'améliorer la rapidité d'exécution du code. Par contre, son utilisation assez intuitive permet de gagner du temps. Il s'agit donc de faire l'arbitrage entre le temps passé à trouver un moyen élégant et efficace pour effectuer une opération en passant par de réels calculs vectoriels et le gain d'exécution que ce calcul vectoriel apporte vis-à-vis d'une boucle. La syntaxe de la fonction `Vectorize()` est la suivante :

```
Vectorize(FUN, vectorize.args, SIMPLIFY, USE.NAMES)
```

avec `FUN` une fonction à appliquer, `vectorize.args` un vecteur de paramètres (de type caractère) qui devraient être vectorisés (par défaut, tous les paramètre de `FUN`). Les paramètres `SIMPLIFY` et `USE.NAMES` on le même emploi que dans la fonction `sapply()`.

```
f <- function(x = 1:3, y) c(x, y)
# On "vectorise" la fonction f
vf <- Vectorize(f, SIMPLIFY = FALSE)
f(1:3, 1:3)

## [1] 1 2 3 1 2 3

vf(1:3, 1:3)

## [[1]]
## [1] 1 1
##
## [[2]]
## [1] 2 2
##
## [[3]]
## [1] 3 3

# Vectorise seulement y, pas x
vf(y = 1:3)

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

4.3 Exercices

Exercice 4.1 : boucle while

4.1.1 À l'aide de la fonction `while()`, créer une boucle qui permet de calculer la factorielle d'un nombre ;

4.1.2 Réutiliser le code de la question précédente pour en faire une fonction qui, lorsqu'on lui donne un nombre, retourne sa factorielle. Comparer le résultat avec la fonction `factorial()`.

Exercice 4.2 : boucles while et for

4.2.1 Choisir un nombre mystère entre 1 et 100, et le stocker dans un objet que l'on nommera `nombre_mystere`. Ensuite, créer une boucle qui à chaque itération effectue un tirage aléatoire d'un entier compris entre 1 et 100. Tant que le nombre tiré est différent du nombre mystère, la boucle doit continuer. À la sortie de la boucle, une variable que l'on appellera `nb_tirages` contiendra le nombre de tirages réalisés pour obtenir le nombre mystère ;

4.2.2 Utiliser le code de la question précédente pour réaliser la fonction `trouver_nombre`, qui, lorsqu'on lui donne un nombre compris entre 1 et 100, retourne le nombre de tirages aléatoires d'entiers compris entre 1 et 100 nécessaires avant de tirer le nombre mystère ;

4.2.3 En utilisant une boucle `for`, faire appel 1000 fois à la fonction `trouver_nombre()` qui vient d'être créée. À chaque itération, stocker le résultat dans un élément d'un vecteur que l'on appellera `nb_essais_rep`. Enfin, afficher la moyenne du nombre de tirages nécessaires pour retrouver le nombre magique.

```
nb_essais_rep <- rep(NA, 1000)
```

Exercice 4.3 : boucles for

4.3.1 Parcourir les entiers de 1 à 20 à l'aide d'une boucle `for` en affichant dans la console à chaque itération si le nombre courant est pair ;

4.3.2 L'objet `month.name` est un vecteur contenant les noms des mois du calendrier, en anglais. Parcourir chacun des éléments de ce vecteur, et afficher dans la console pour chacun des mois si le nombre de caractères composant le nom du mois est pair ou impair.

Exercice 4.4 : suite de Fibonacci

Utiliser une boucle `for` pour reproduire la suite de Fibonacci jusqu'à son dixième terme (la séquence F_n est définie par la relation de récurrence suivante : $F_n = F_{n-1} + F_{n-2}$; les valeurs initiales sont : $F_0 = 0$ et $F_1 = 1$).

Exercice 4.5 : barre de progression

Considérons le vecteur de chaînes de caractères `ids` ainsi que la liste `res` :

```
library(magrittr)
n <- 1000
ids <-
  str_c(sample(LETTERS, n, replace = TRUE),
              sample(letters, n, replace = TRUE)) %>%
  unique()
res <- vector("list", length(ids))
```

Parcourir les éléments du vecteur `ids` à l'aide d'une boucle `for`. À chaque itération, stocker dans l'élément de la liste `res` dont la position correspond à celle de l'identifiant courant dans `ids` les informations suivantes : (i) l'identifiant courant et (ii) la somme de 50000 tirages aléatoires selon une $\mathcal{N}(0, 1)$.
Afficher l'état d'avancement de la boucle à l'aide d'une barre de progression.

Exercice 4.6 : fonctions appliquées aux éléments d'une liste

Soit une liste nommée `twittos`, disponible à l'adresse suivante : http://egallic.fr/Enseignement/R/Exercices/donnees/twittos.rda. Elle contient des informations fictives sur des utilisateurs de Twitter ; chaque élément de cette liste est une liste dont les éléments sont les suivants :

- `screen_name` : nom d'utilisateur ;
- `nb_tweets` : nombre de tweets ;
- `nb_followers` : nombre de followers ;
- `nb_friends` : nombre de followings ;
- `created_at` : date de création du compte ;
- `location` : ville renseignée ;

4.6.1 Importer le contenu du fichier dans la session `R` ;

4.6.2 Utiliser la fonction `lapply()` sur `twittos` pour récupérer une liste contenant uniquement les noms d'utilisateurs. Faire de même pour le nombre de followers, puis appliquer `unlist()` au résultat ;

4.6.3 Créer une fonction qui, quand on lui fournit un élément de la liste `twittos`, c'est-à-dire les informations sous forme de liste d'un seul utilisateur, retourne ces informations sous forme de tableau de données. Nommer cette fonction `twittos_to_df` ;

4.6.4 Appliquer la fonction `twittos_to_df()` au premier élément de la liste `twittos`, puis utiliser la fonction `lapply()` pour appliquer la fonction `twittos_to_df()` à tous les éléments de la liste. Stocker ce dernier résultat dans un objet appelé `res` ;

4.6.5 Quelle est la structure de l'objet `res` obtenu à la question précédente ?

4.6.6 Utiliser la fonction appropriée dans le *package* `dplyr` pour transformer `res` en tableau de données ;

4.6.7 Importer le fichier disponible à cette adresse dans la session `R` : http://egallic.fr/Enseignement/R/Exercices/donnees/dates_tw.rda. Il s'agit d'une liste donc chaque élément contient une liste indiquant le nom d'un utilisateur et la date de chacun de ses tweets.

4.6.8 Appliquer la fonction `lapply()` à la liste `dates_tw` qui vient d'être importée dans `R`, pour afficher l'heure moyenne des tweets pour chaque utilisateur, puis faire de même pour l'écart-type.

Exercice 4.7 : fonctions appliquées aux éléments d'une matrice

4.7.1 Créer une matrice de dimension 100×5, donc chaque vecteur colonne est composé de tirages issus d'une loi Normale centrée réduite ;

4.7.2 Utiliser la fonction `apply()` pour calculer la moyenne des valeurs de chaque colonne ;

4.7.3 Utiliser la fonction `apply()` pour calculer l'écart-type des valeurs de chaque colonne.

5

Graphiques

L'analyse de données passe souvent par une phase de visualisation. `R` propose différentes méthodes pour créer des graphiques. Le *package* `graphics`, qui est installé d'office et chargé par défaut au démarrage de `R`, contient de nombreuses fonctions pour réaliser des graphiques. Cependant, nous allons plutôt présenter les bases de la confection de graphiques à partir du *package* `ggplot2`. Comme ce document a vocation à être une introduction, seule une infime partie des possibilités offertes par `ggplot2` est abordée. Le lecteur intéressé est invité à consulter des excellents ouvrages ayant été publiés (Chang, 2013; Wickham, 2009).

Les graphiques avec `ggplot2` sont crées par couches (*layers*), avec la première constituée des données brutes, puis les suivantes d'annotations, d'informations statistiques. `ggplot2` s'appuie sur une grammaire. L'apprentissage de cette grammaire nécessite un peu de temps, mais permet de créer des graphiques intelligibles. La grammaire graphique crée une carte, un plan (*mapping*) pour passer des données aux attributs esthétiques des objets géométriques (comme la couleur, la forme, ou la taille des points, lignes, etc.). Elle permet également d'appliquer des transformations aux données avant de les représenter, ou encore de faire du facettage (*faceting*).

Pour pouvoir utiliser les fonctions de `ggplot2`, il est nécessaire d'installer le *package* lors de la première utilisation, puis de le charger les fois suivantes.

```
# Lors de la première utilisation
install.packages("ggplot2")

# Les fois suivantes
library("ggplot2")
```

Avant de rentrer dans le vif du sujet, il est important de mentionner la page d'aide du *package* sur Internet, qui est très bien réalisée, et s'avère être un excellent aide-mémoire : `http://docs.ggplot2.org/current/`.

5.1 Structure

Les éléments de la grammaire graphique utilisés par `ggplot2` sont les suivants :

- des données brutes (***data***) ;
- une projection graphique pour lier ces données aux attributs esthétiques (***mapping***) ;
- des objets géométriques (***geom***) (points, lignes, polygones, etc.) ;
- des transformations statistiques (***stats***) ;
- des échelles (***scale***) qui lient les données à un espace esthétique ;
- un système de coordonnées (***coord***), indiquant comment les coordonnées issues des données sont positionnées sur le graphique ;
- une indication de regroupement à effectuer dans les données pour créer des sous-groupes, afin de créer des facettes (***facet***).

Les mots en gras dans la liste ci-avant servent de base dans les noms des fonctions du *package* `ggplot2`. Par exemple, les fonctions pour définir le type de géométrie souhaité commencent par `geom_`.

La création d'un graphique avec `ggplot2` débute avec l'appel de la fonction `ggplot()`. Les couches supplémentaires (*layers*) sont ajoutées en ayant recours au symbole `+`. La syntaxe est la suivante :

```
ggplot(data, aes(x, y)) + layers
```

Il est possible de stocker un graphique créé avec `ggplot2` dans une variable, en utilisant la flèche d'assignation. Des nouvelles couches peuvent ensuite être ajoutées à l'objet stocké.

```
p <- ggplot(data, aes(x, y)) + layers
p + new_layers
```

Les données doivent être fournies sous forme de *data.frame* à la fonction `ggplot()`.

5.2 Des graphiques élaborés avec `ggplot()`

Pour illustrer les différentes notions, un échantillon de 135 films est utilisé. Il st issus de la base "film" obtenue sur *freebase*[1], et contient des informations relatives à la date de première date de sortie, à la durée la plus longue du film, l'estimation du budget, les revenus bruts et le premier pays indiqué concernant la réalisation du film.

```
load(url("http://egallic.fr/R/films.rda"))
head(films)
```

```
##                                    name initial_release_date runtime
## 1                          Galaxy Quest           1999-12-23     102
## 2          Star Trek: First Contact           1996-11-18     111
## 3                          Forrest Gump           1994-06-23     142
## 4 Star Wars Episode I: The Phantom Menace           1999-05-19     136
## 5                               Titanic           1997-11-01     194
## 6                     Independence Day           1996-06-25     153
##   year estimated_budget gross_revenue                 country country_abr
## 1 1999         4.50e+07      90683916 United States of America         USA
## 2 1996         4.50e+07      92001027 United States of America         USA
## 3 1994         5.50e+07     677387716 United States of America         USA
## 4 1999         1.15e+08    1027044330 United States of America         USA
```

1. https://www.freebase.com/film/film?schema=

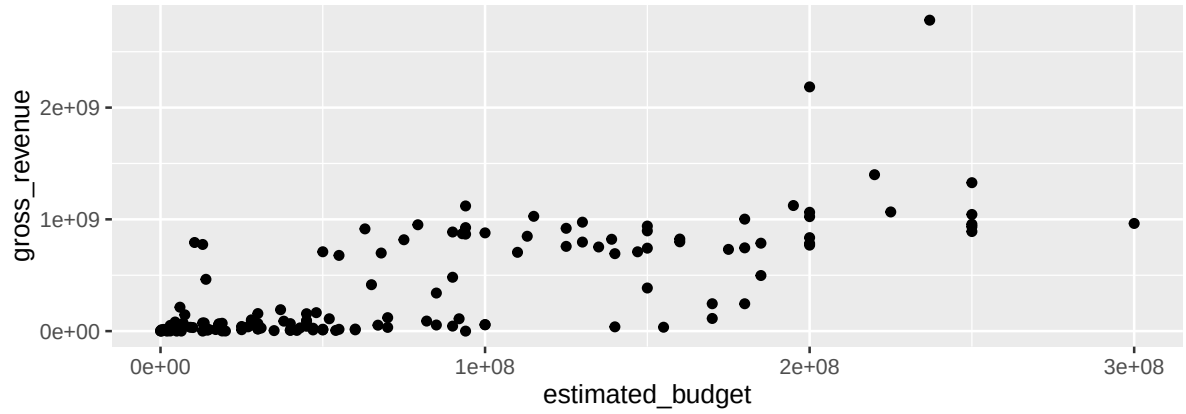
```
## 5 1997         2.00e+08    2185240703 United States of America         USA
## 6 1996         7.50e+07     817400891 United States of America         USA
```

Créons une sous-base, qui ne concerne que quelques pays :

```
library(dplyr)
pays_liste <-
  c("United States of America", "New Zealand", "United Kingdom", "Spain")
films_reduit <-
  films %>%
  filter(country %in% pays_liste)
```

Le graphique le plus commun est sans doûte le nuage de points (ou *scatterplot*).
L'exemple suivant trace des un nuage de points représentant le budget estimé de chaque film en fonction des recettes brutes. Nous avons besoin d'utiliser la fonction `geom_point()`, qui définit le type de géométrie. D'autres types de géométries sont présentés à la section 5.2.2.

```
ggplot(data = films, aes(x = estimated_budget, y = gross_revenue)) +
  geom_point()
```



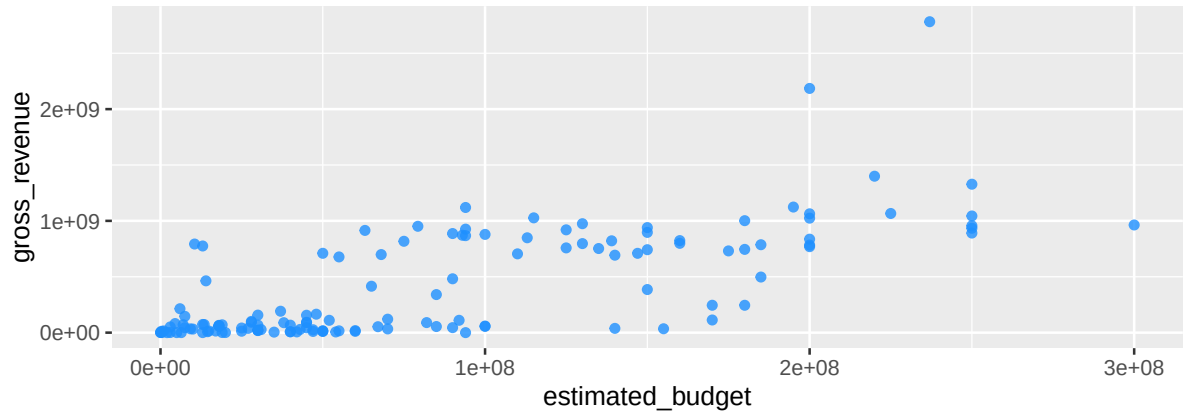
5.2.1 Paramètres esthétiques

Parmi les paramètres esthétiques que l'on peut modifier, on retrouve :

- `colour` : la couleur ;
- `shape` : la forme ;
- `size` : la taille ;
- `alpha` : la transparence ;
- `fill` : le remplissage.

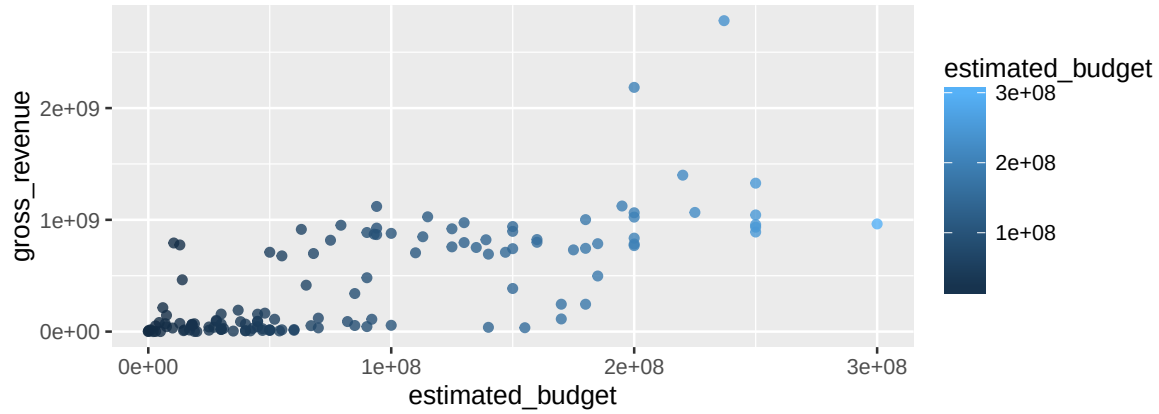
Dans l'exemple précédent, ces paramètres n'ayant pas étés modifiés, ils ont conservé leur valeur par défaut. Pour les modifier, il suffit d'indiquer la valeur voulue à l'intérieur de la fonction `ggplot()` ou d'une fonction de géométrie.
Par exemple, pour que la couleur des points soit `dodger blue` et que la transparence soit de `0.8` :

```
ggplot(data = films, aes(x = estimated_budget, y = gross_revenue)) +
  geom_point(colour = "dodger blue", alpha = .8)
```



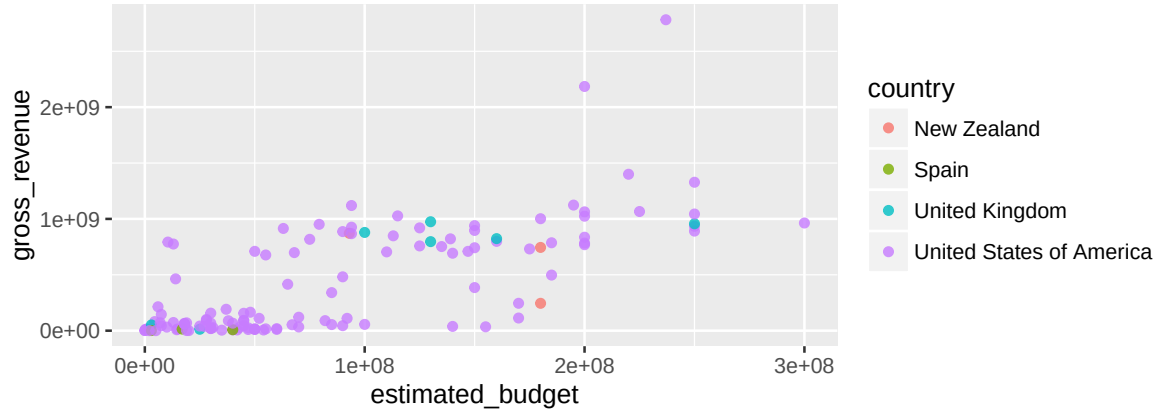
Il est également possible de faire dépendre les paramètres esthétiques des valeurs prises par les variables du *data.frame*. En fonction du mode de la variable utilisée, les effets seront différents. Par exemple, si on décide de faire dépendre la couleur d'une variable continue, une échelle de couleurs sera utilisée ; si la variable est discrète, une palette de couleurs est utilisée. Pour ce faire, il faut indiquer le *mapping* à l'intérieur de la fonction `aes()`.
Par exemple, pour faire dépendre la couleur des points de la valeur prise par l'estimation de budget (variable continue) :

```
ggplot(data = films_reduit, aes(x = estimated_budget, y = gross_revenue)) +
  geom_point(alpha = .8, aes(colour = estimated_budget))
```



Et pour faire dépendre la couleur des points du pays d'origine :

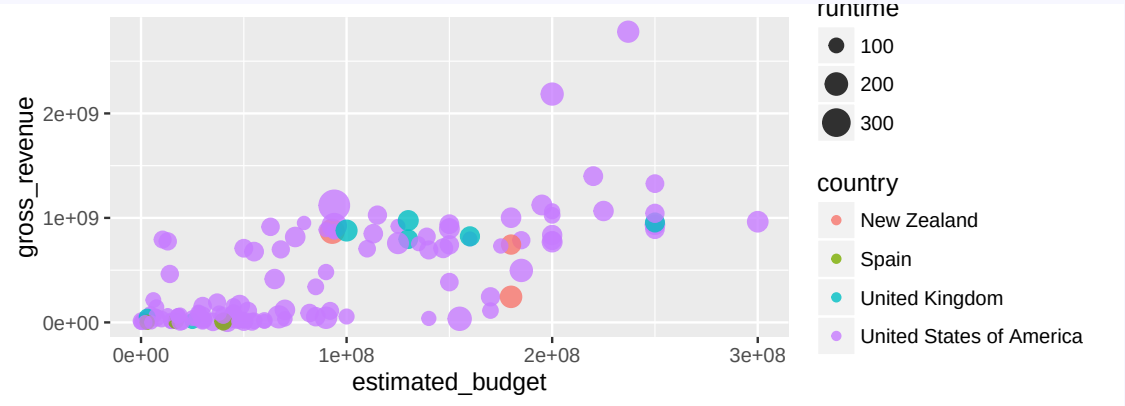
```
ggplot(data = films_reduit, aes(x = estimated_budget, y = gross_revenue)) +
  geom_point(alpha = .8, aes(colour = country))
```



Remarque 5.2.1

Une petite astuce pour créer un *bubble chart* consiste à attribuer au paramètre `size` des valeurs prises par une variable.

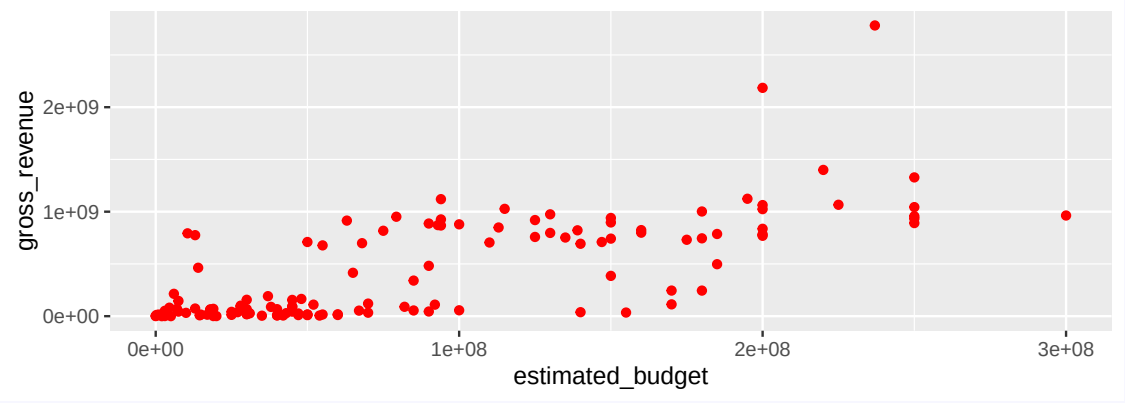
```
ggplot(data = films_reduit, aes(x = estimated_budget, y = gross_revenue)) +
  geom_point(alpha = .8, aes(colour = country, size = runtime))
```



Remarque 5.2.2

Si on souhaite que la couleur de la forme géométrique soit la même pour toutes les observations, il faut renseigner le paramètre `color` dans la fonction `geom_*` et non pas `ggplot()`.

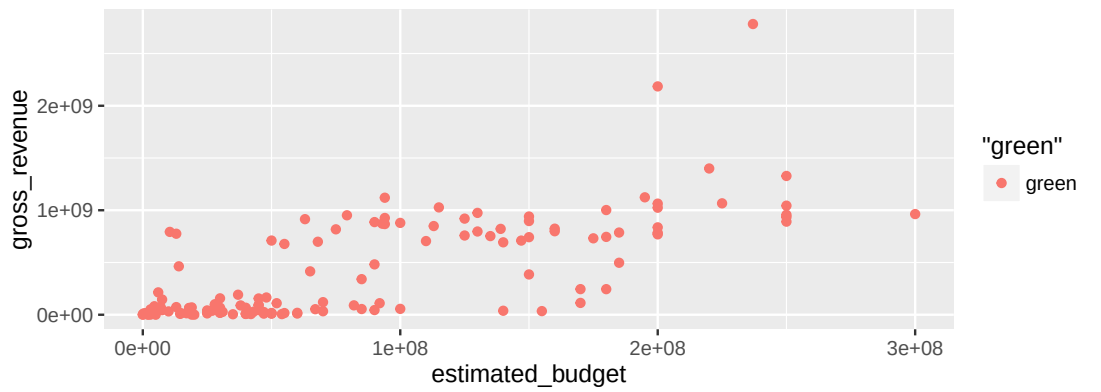
```
p_rem <- ggplot(data = films_reduit, aes(x = estimated_budget,
                                         y = gross_revenue)) +
  geom_point(colour = "red")
p_rem
```



Remarque 5.2.3

Si on fournit un paramètre esthétique à la fonction `aes()`, à l'intérieur de la fonction `geom_*()`, une nouvelle variable est créé, qui vaut, pour toutes les observations, la valeur attribuée au paramètre esthétique. De fait, `ggplot2` se charge d'interpréter cela comme une variable de mode facteur, qui ne contient qu'un niveau.

```
p_rem <- ggplot(data = films_reduit, aes(x = estimated_budget,
                                         y = gross_revenue)) +
  geom_point(aes(colour = "green"))
# La couleur pour les points va s'appuyer sur une variable qui
# vient d'être créée, et qui vaut "green" pour toutes les obs.
# Il est alors considéré qu'il n'y a qu'un facteur.
# Donc la couleur affichée sera le rose !
p_rem
```



5.2.2 Paramètres géométriques (fonctions `geom_*()`)

Pour définir le type d'objet géométrique utilisé pour représenter les données, on ajoute une couche au graphique. Le nom des fonctions qui ajoutent ce type de couches possède la syntaxe suivante : `geom_*`, où l'étoile doit être remplacée par le nom d'une forme géométrique. Les principales fonctions sont les suivantes :

- `geom_point()` : tracer des points ;
- `geom_line()` : tracer des lignes ;
- `geom_polygon()` : tracer des lignes ;
- `geom_path()` : tracer des points dans l'ordre du *data.frame* ;
- `geom_step()` : faire un graphique en escalier ;
- `geom_boxplot()` : tracer une boîte à moustache ;
- `geom_jitter()` : mettre des points côte à côte pour une variable catégorielle ;
- `geom_smooth()` : ajouter une courbe de tendance ;
- `geom_histogram()` : tracer un histogramme ;
- `geom_bar()` : tracer un diagramme en bâton ;
- `geom_density()` : tracer une estimation de densité.

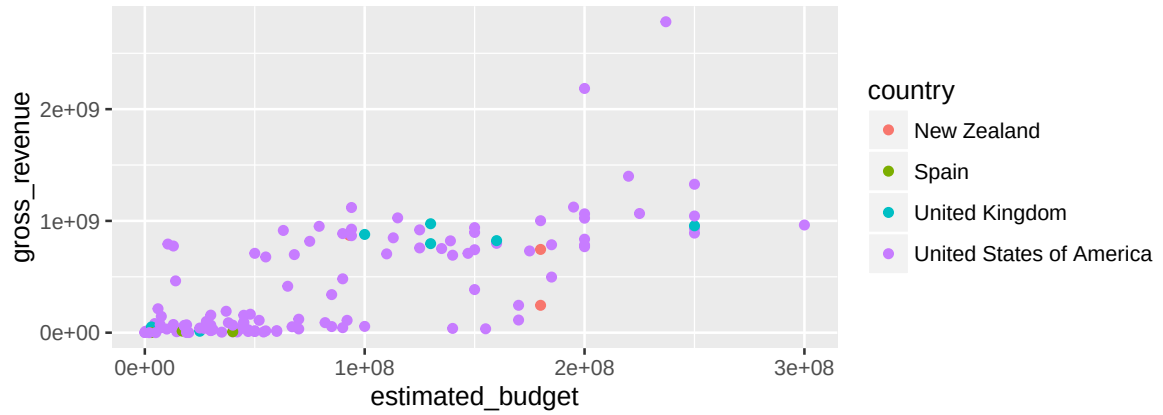
Toutes les fonctions `geom_*()` possèdent les paramètres optionnels suivants : `data`, `mapping`, `...`, `geom` (ou `stat`) et `position`. S'ils sont omis, leur valeur est héritée de celle des paramètres correspondants de la fonction `ggplot()`. Si à l'inverse ils sont renseignés, alors leur valeur vient remplacer celle héritée. Certaines fonctions ont d'autres paramètres ; le lecteur est invité à consulter l'aide pour de plus amples détails.

Le but ici n'est pas de passer en revue chaque fonction, mais de permettre au lecteur de comprendre le fonctionnement des graphiques en couche avec `ggplot2`. Toutes les fonctions `geom_*()` fonctionnent de la même manière. En étant capable de comprendre les exemples qui suivent, il ne devrait pas être compliqué d'explorer les cas non traités dans ce document.

5.2.2.1 Points

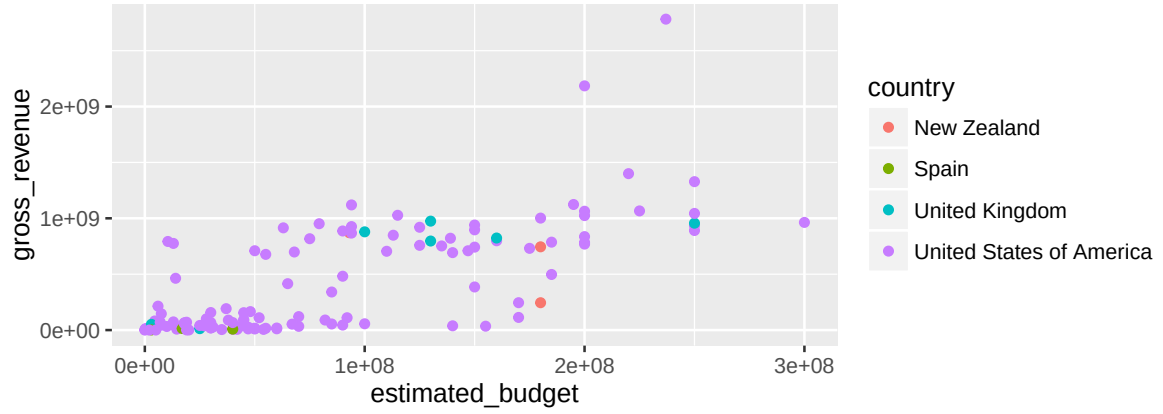
La fonction qui permet de tracer des points a déjà été utilisées à plusieurs reprises : `geom_point()`.

```
ggplot(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue, col = country)) +
  geom_point()
```



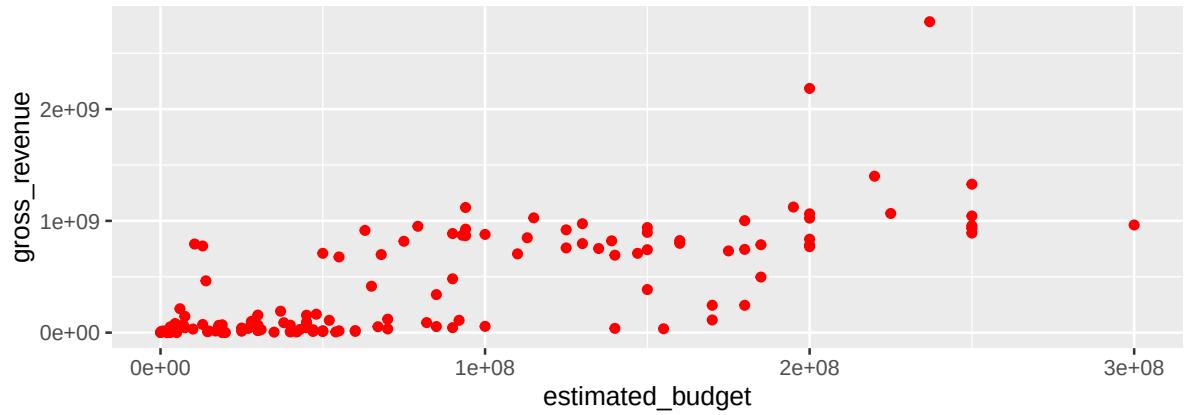
Si on renseigne le chemin vers les données à l'intérieur de la fonction `geom_point()` plutôt que dans `ggplot()`, il faut veiller à bien préciser le pramaètre `data =` !

```
ggplot() +
  geom_point(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue, col = country))
```



Attention, pour que tous les points aient la même couleur, il faut que le paramètre esthétique soit fourni à `geom_point()` et non pas à `ggplot()`.

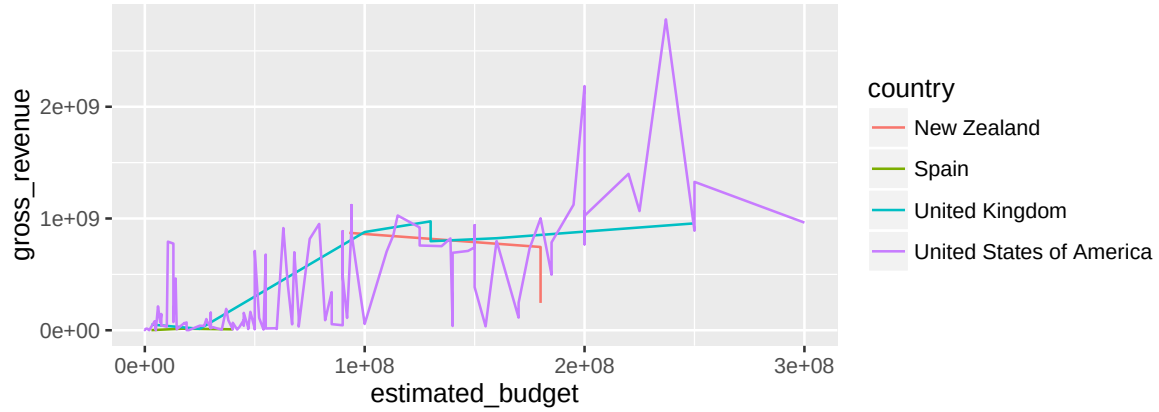
```
ggplot(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue)) +
  geom_point(colour = "red")
```



5.2.2.2 Lignes

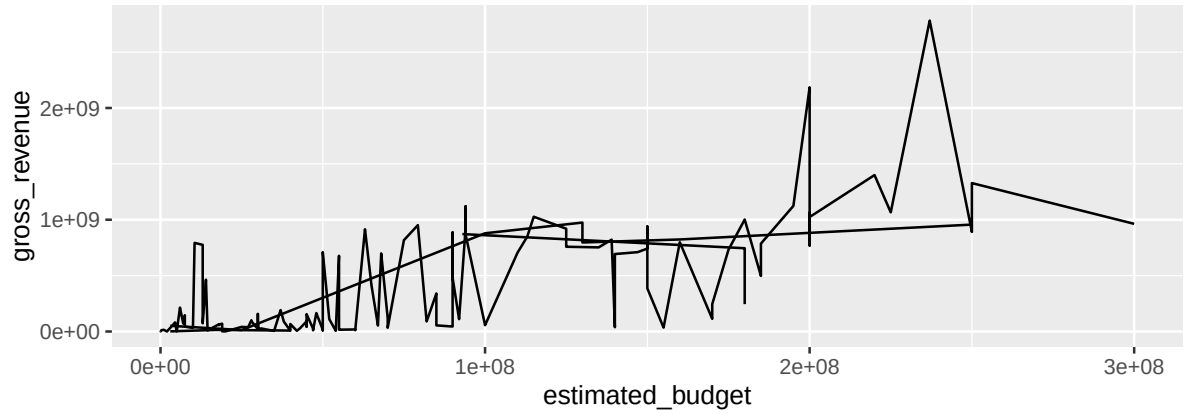
Pour tracer des lignes entre les points, il faut utiliser la fonction `geom_line()`.

```
ggplot(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue, col = country)) +
  geom_line()
```



Le fait d'ajouter le paramètre `colour` et de lui donner en valeur une variable de type facteur crée automatiquement des groupes. Les points de chaque groupe seront reliés entre eux.

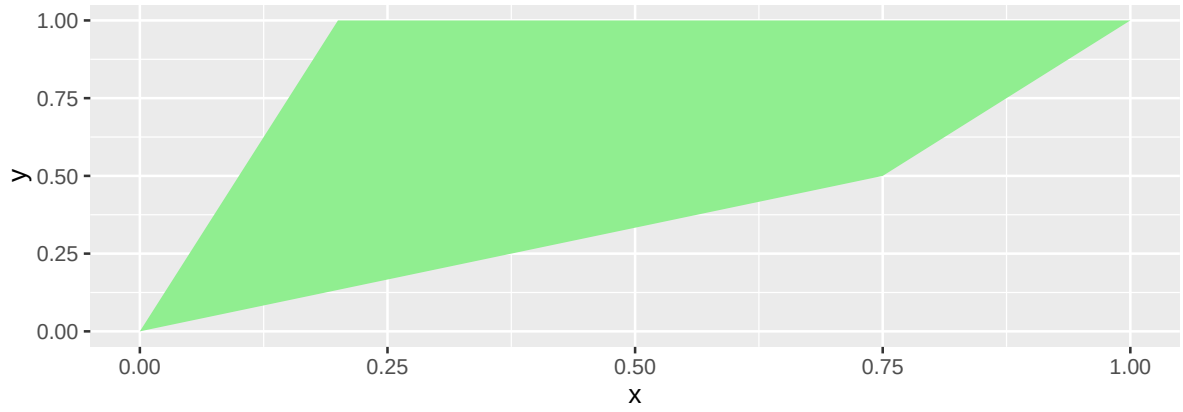
```
ggplot(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue, group = country)) +
  geom_line()
```



5.2.2.3 Polygones

La fonction `geom_polygon()` permet de tracer des polygones. Les coordonnées doivent être ordonnées dans le sens direct. Tandis que de nombreuses fonctions utilisant des polygones en `R` nécessitent que la première et la dernière observation soient identiques, et ce afin de pouvoir fermer le polygone, ce n'est pas obligatoire avec `ggplot2`.

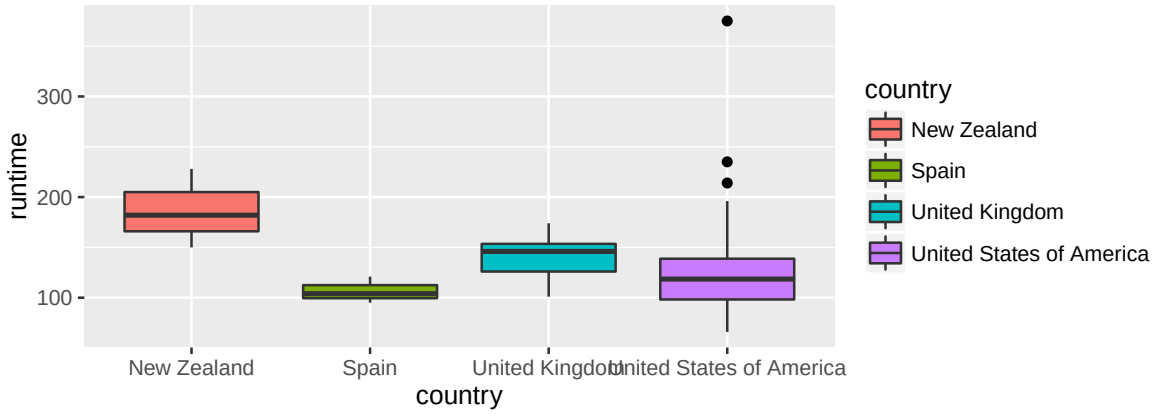
```
df <- data.frame(x = c(0, 0.2, 1, 0.75), y = c(0, 1, 1, 0.5))
ggplot() + geom_polygon(data = df, aes(x = x, y = y), fill = "light green")
```



5.2.2.4 Boxplot

Pour réaliser une boîte à moustache, il faut utiliser la fonction `geom_boxplot()`.

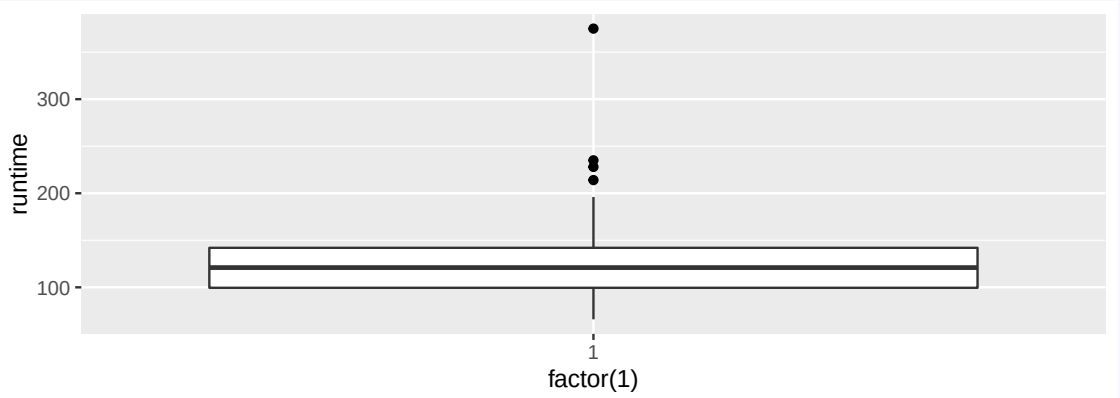
```
ggplot(data = films_reduit,
       aes(x = country, y = runtime, fill = country)) +
  geom_boxplot()
```



Remarque 5.2.4

La fonction `geom_boxplot()` ne permet pas de tracer de manière élégante (au sens du code) la boîte à moustache pour une seule variable. Il faut pour cela, avoir recours à une petite astuce : fournir une variable factorielle à une seule modalité à l'esthétique `x`.

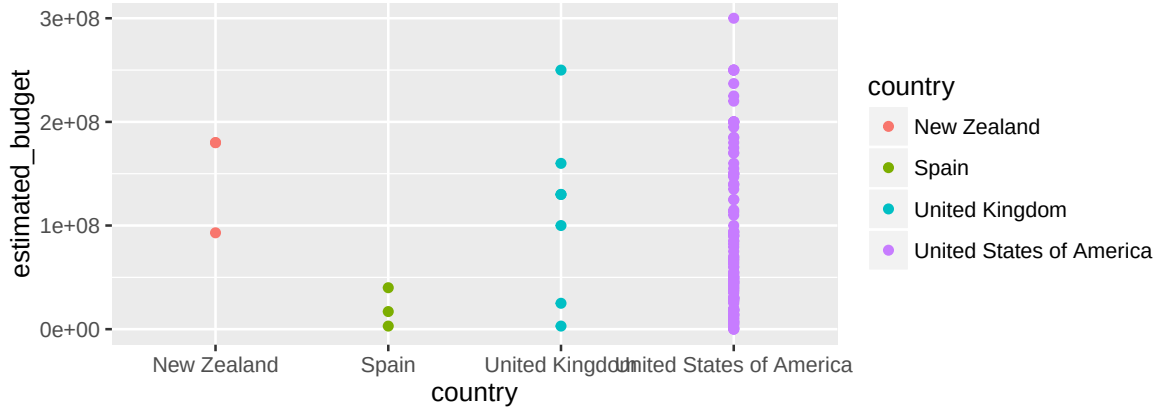
```
ggplot(data = films_reduit,
       aes(x = factor(1), y = runtime)) + geom_boxplot()
```



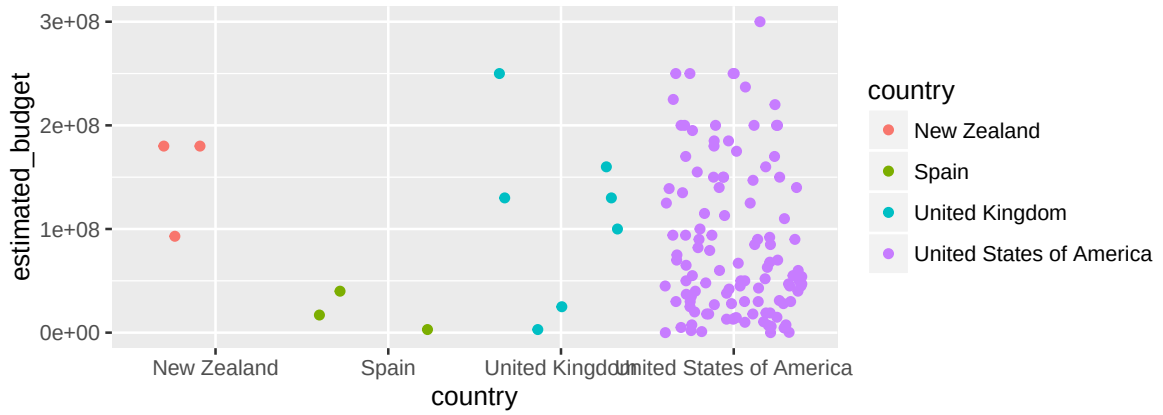
5.2.2.5 Gigue

Une forme géométrique qui peut être pratique est `jitter`. Elle permet de placer les points côte à côte au lieu de les entasser. Dans le cas de variables catégorielles, cela peut être intéressant à utiliser.

```
ggplot(data = films_reduit,
       aes(x = country, y = estimated_budget, col = country)) +
  geom_point()
```



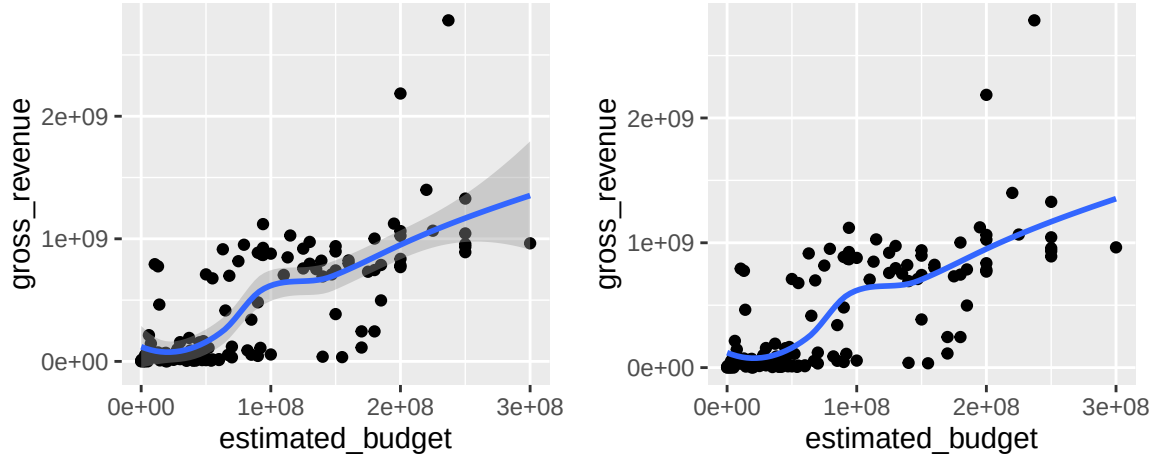

```
ggplot(data = films_reduit,
       aes(x = country, y = estimated_budget, col = country)) +
  geom_jitter()
```



5.2.2.6 Courbe de tendance

On peut facilement ajouter une courbe de tendance à un nuage de points, en ajoutant une couche avec la fonction `geom_smooth`. Des intervalles de confiance associés aux valeurs obtenues par le lissage sont automatiquement dessinés. Pour les retirer, il faut ajouter `se=FALSE` à la liste des paramètres.

```
ggplot(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue)) +
  geom_point() +
  geom_smooth()

ggplot(data = films_reduit,
       aes(x = estimated_budget, y = gross_revenue)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

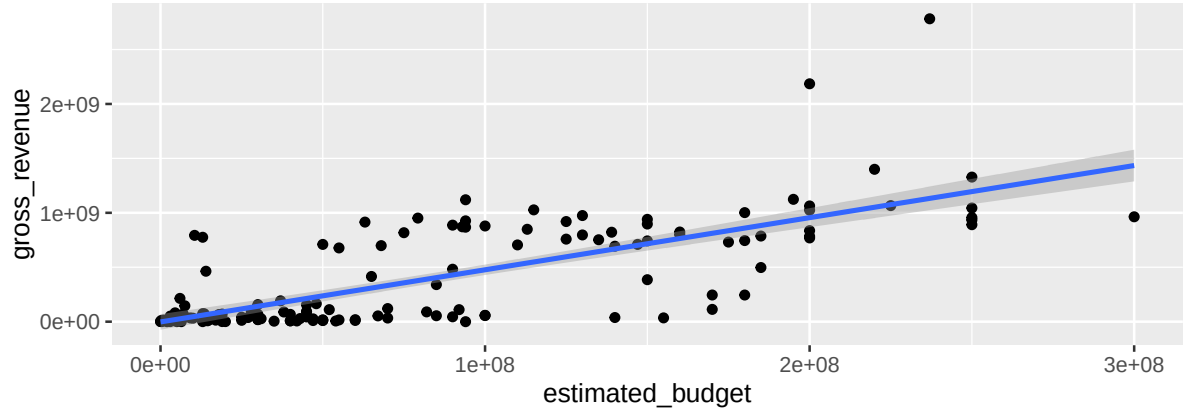


Le paramètre `method` contrôle le type de lissage. Par défaut, la fonction `loess()` est appelée pour le réaliser, et on a `method = "loess"`. Si on regarde la page d'aide de cette fonction, on voit que cette dernière possède plusieurs paramètres modifiables. Comme le paramètre `...` est présent dans la fonction `ggplot()`, on peut transmettre à la fonction de lissage les paramètres que l'on souhaite modifier.

Parmi les autres méthodes qu'il est possible d'utiliser pour le lissage, on retrouve `lm`, pour un modèle linéaire, `rlm` (*package* `MASS`) pour un modèle linéaire robuste, ou encore `gam` (*package* `mgcv`) pour un modèle additif généralisé.

Voici un exemple, avec un lissage par régression linéaire, avec un intervalle de confiance pour la prévision à 90% :

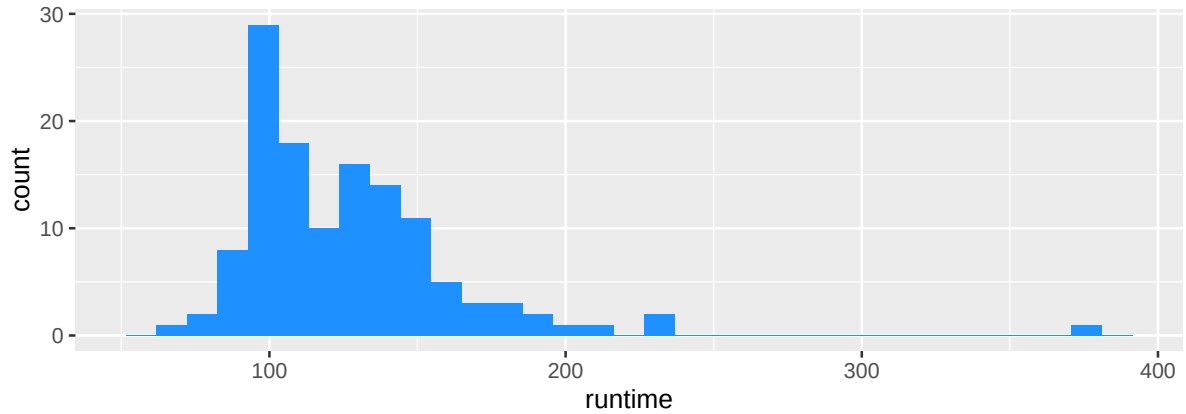
```
ggplot(data = films, aes(x = estimated_budget, y = gross_revenue)) +
  geom_point() +
  stat_smooth(method = "lm", level = 0.9)
```



5.2.2.7 Histogramme

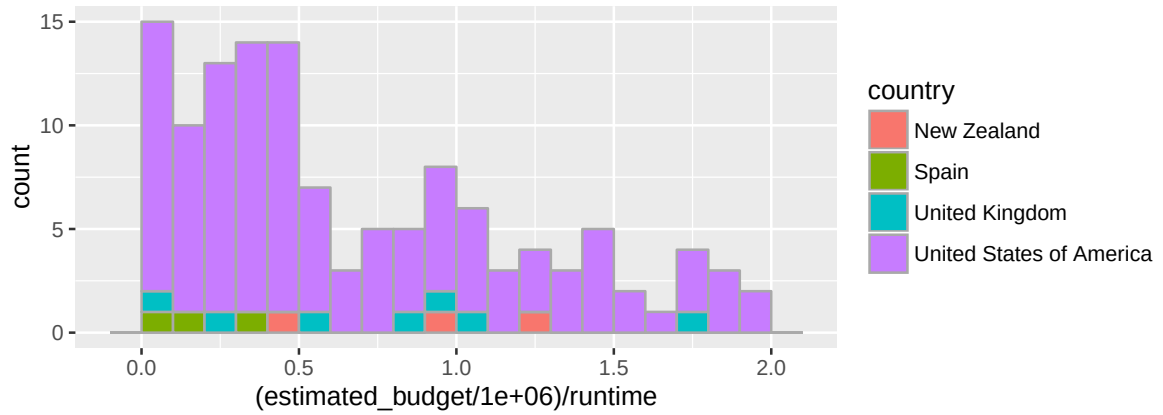
Pour réaliser un histogramme, `ggplot2` propose la fonction `geom_histogram()`. La fenêtre (*bindwidth*) par défaut est l'étendue divisée par 30.

```
ggplot(data = films_reduit,
       aes(x = runtime)) +
  geom_histogram(fill = "dodger blue")
```



Si on ajoute au moins un des paramètres esthétiques `colour` ou `fill` en fonction d'une variable, les histogrammes feront état des sous-groupes.

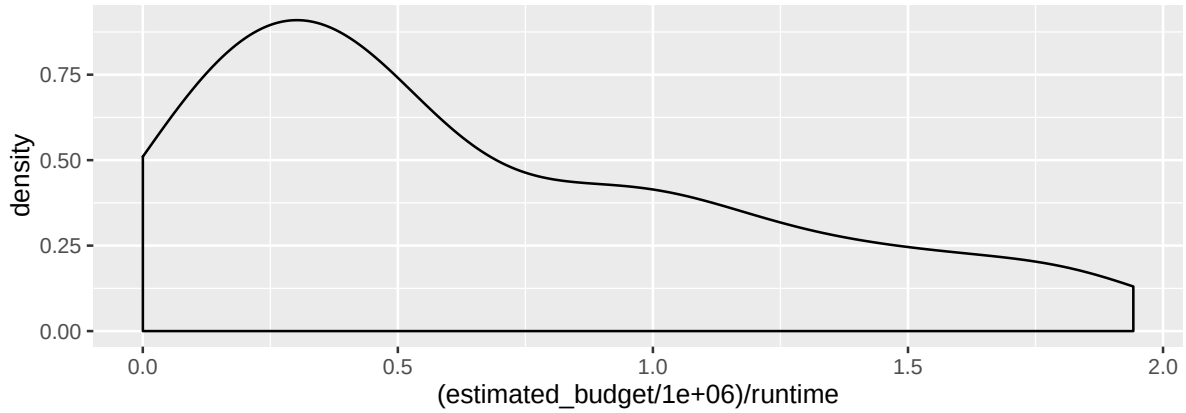
```
ggplot(data = films_reduit,
       aes(x = (estimated_budget/1000000)/runtime,
           fill = country))+
  geom_histogram(binwidth = 0.1, colour = "dark grey")
```



5.2.2.8 Densité

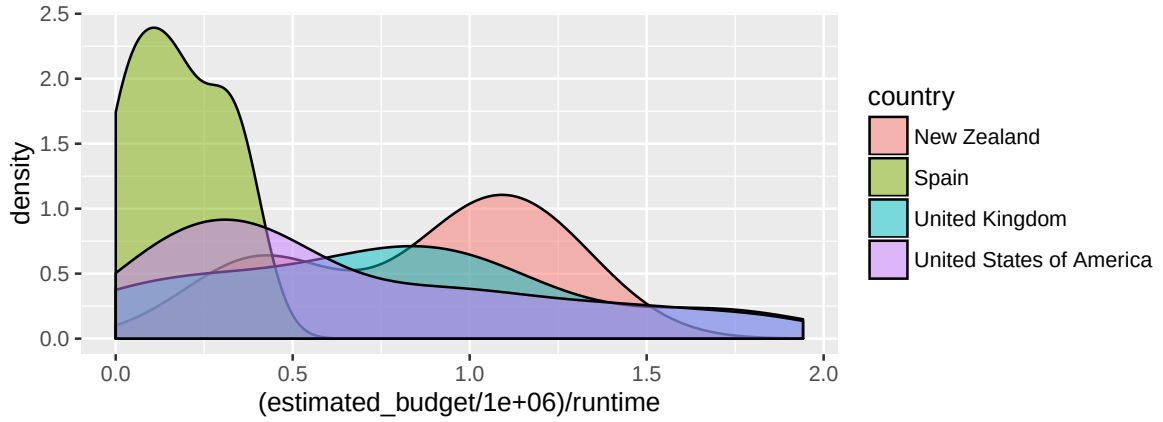
Pour afficher des estimations de la densité, il faut utiliser la fonction `geom_density()`. Le noyau peut être changé *via* le paramètre `kernel` (c.f. la page d'aide `?stat_density` pour de plus amples détails).

```
ggplot(data = films_reduit,
       aes(x = (estimated_budget/1000000)/runtime)) +
  geom_density(colour = "black")
```



Si on ajoute au moins un des paramètres esthétiques `colour` ou `fill` en fonction d'une variable, les densités seront estimées pour chaque sous groupe.

```
ggplot(data = films_reduit,
       aes(x = (estimated_budget/1000000)/runtime,
           fill = country)) +
  geom_density(colour = "black", alpha = .5)
```

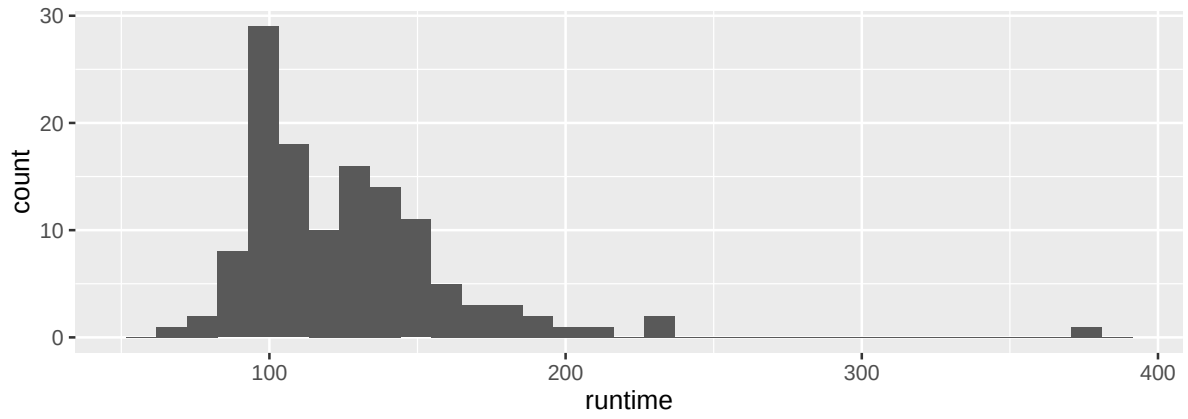


5.2.3 Paramètres statistiques (fonctions `stat_*()`)

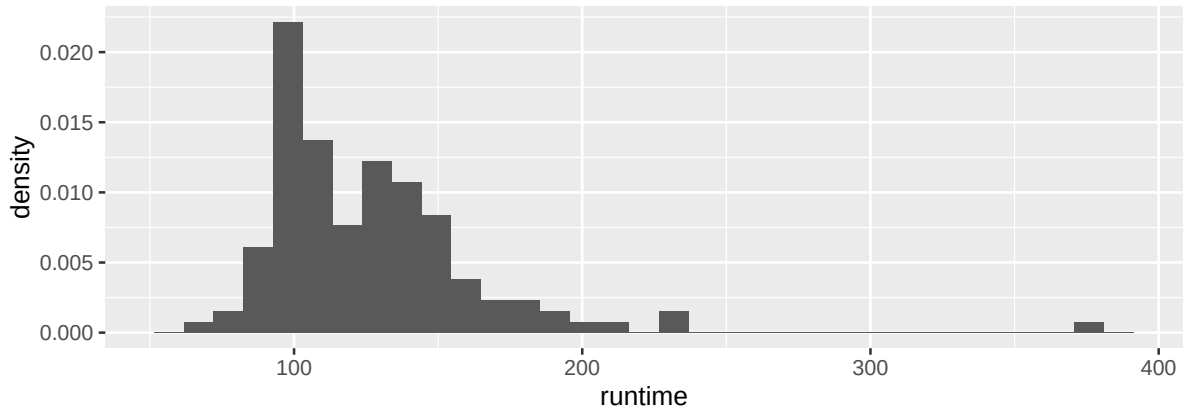
Pour effectuer des opérations statistiques sur les données avant de les représenter, `ggplot2` propose de nombreuses fonctions. Leur nom suit la syntaxe suivante : `stat_*` avec `*` le nom d'une opération à réaliser, comme par exemple `density` pour une estimation de la densité par la méthode du noyau. Les fonctions de type `geom_*` s'appuient sur celles de type `stat_*`. Une ou plusieurs variables sont créés automatiquement lors de l'appel à ces fonctions.

Il faut savoir qu'il est possible d'accéder à ces variables, à condition de connaître leur nom, en les entourant par deux points de chaque côté. Par exemple, lorsque l'on fait appel à la fonction `geom_histogram()`, un appel interne à la fonction `stat_bin()` est fait par défaut (voir le fichier d'aide de la fonction `geom_histogram()`). Or, la fonction `stat_bin()` retourne entre autre les variables `count`, qui indique le nombre d'observations dans chaque classe créée (*bin*) ; `density` pour la densité des points dans chaque classe, mise à l'échelle pour que l'intégrale vaille 1 (pourcentage du total sur la largeur de la barre).

```
# Par défaut, stat_bin() utilise ..count.. en ordonnées
ggplot(data = films_reduit, aes(x = runtime)) + geom_histogram()
```

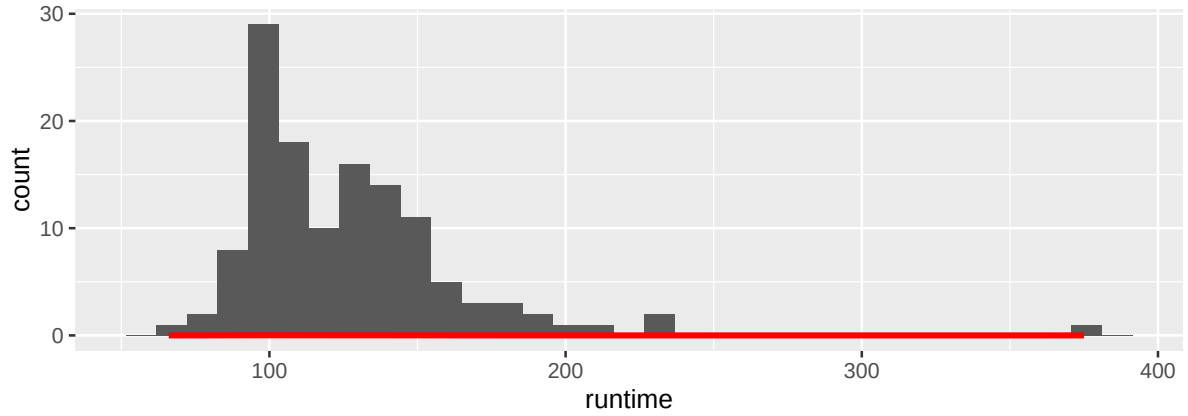



```
# Produit le même résultat
# ggplot(data = films_reduit, aes(x = runtime)) + stat_bin()

# Pour afficher la densité estimée à la place
ggplot(data = films_reduit, aes(x = runtime)) +
  geom_histogram(aes(y = ..density..))
```



Il n'est pas rare de vouloir tracer un histogramme et d'y ajouter une courbe de densité. Pour cela, avec `ggplot2`, on peut faire appel à la fonction `stat_density()`, soit directement, soit par l'intermédiaire de la fonction `geom_line()`, en fournissant la valeur `"density"` au paramètre `stat`.

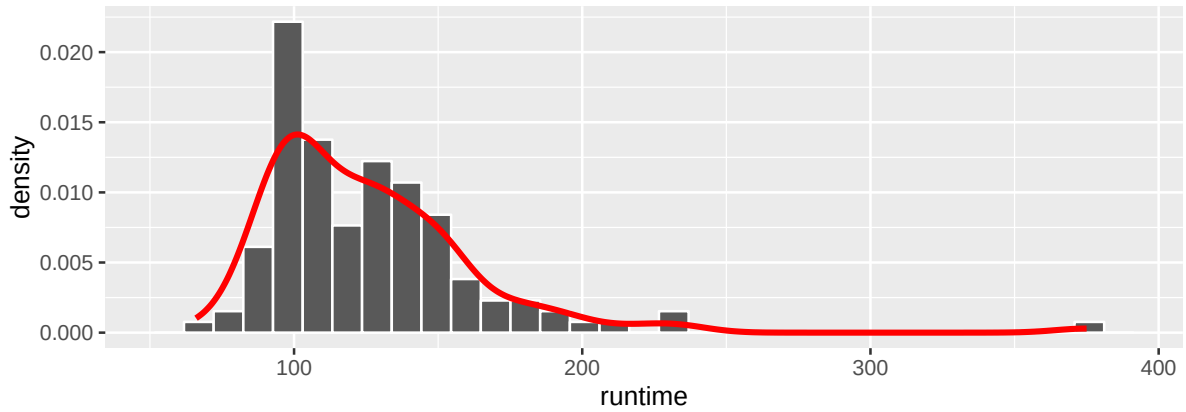
```
p <- ggplot(data = films_reduit, aes(x = runtime))
p + geom_histogram() + geom_line(stat="density", col = "red", size = 1.2)
```



On peut voir sur le précédent graphique que les valeurs pour l'axe des ordonnées sont basées sur ce qu'utilise par défaut la fonction `geom_histogram()` [2], à savoir le nombre de données par classe. En revanche, l'axe des ordonnées pour la densité estimée par `stat_density()` est la valeur de la densité. Comme les valeurs de la densité sont moins élevées que celles du nombre de points par classe, on n'aperçoit qu'une forme applatie de cette première. Aussi, il faut indiquer à la fonction `ggplot()` le *mapping* suivant : `y = ..density...` De fait, lors de l'appel de la fonction `stat_bin()`, la valeur utilisée pour `y` sera la densité plutôt que la variable de comptage.

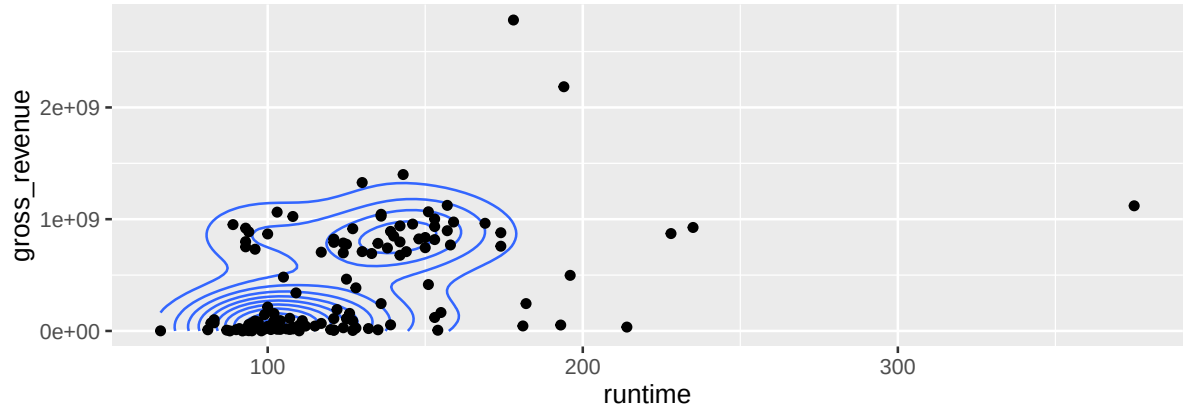
```
p <- ggplot(data = films_reduit, aes(x = runtime, y = ..density..))
p + geom_histogram(colour = "white") +
  geom_line(stat="density", col = "red", size = 1.2)
```

2. On peut noter que si on fait appel à la fonction `geom_line()` avant `geom_histogram()`, l'étiquette de l'axe des ordonnées sera `density`, mais les valeurs seront bien celles du nombre de données par classe !

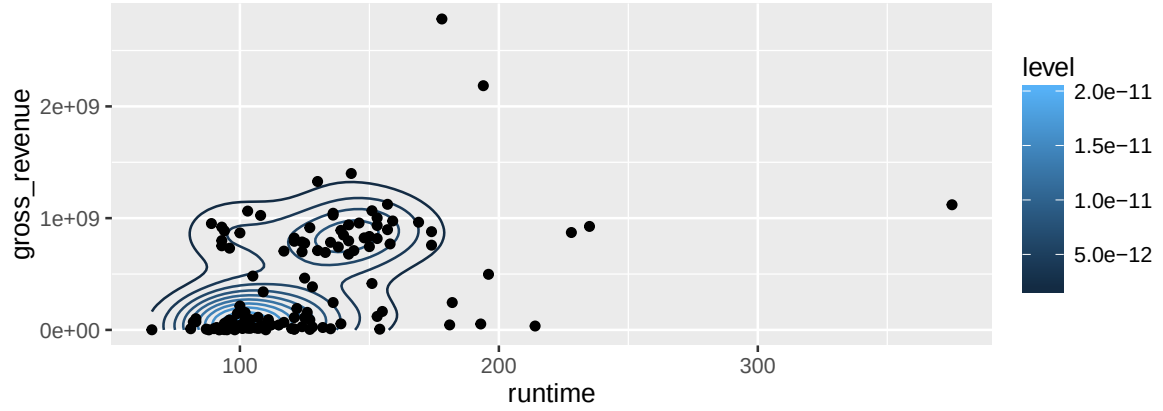


Un dernier exemple des fonctions `stat_*()` (le lecteur étant invité à consulter les références mentionnées au sujet de `ggplot2` pour plus de détails sur ces fonctions) : les estimations de densité en 2d.

```
p <- ggplot(data = films_reduit, aes(x = runtime, y = gross_revenue))

p + stat_density2d() + geom_point()
```




```
# En faisant dépendre la couleur des lignes de contour de la hauteur
p + stat_density2d(aes(colour = ..level..)) + geom_point()
```



Les principales fonctions `stat_*()`, qui peuvent donc être soit appelées directement, soit en fournissant le suffixe au paramètre `stat` d'une fonction `geom_*()`, sont les suivantes :

- `stat_bin()` : répartition des données en classes ;
- `stat_contour()` : calculer les contours des données en 3d ;

- `stat_density()` : estimation de densité 1d par la méthode du noyau ;
- `stat_density2d()` : estimation de densité 2d ;
- `stat_identity()` : ne transforme pas les données ;
- `stat_qq()` : *qqplot* (droite de Henry) ;
- `stat_quantile()` : quantiles continus ;
- `stat_smooth()` : lissage ;
- `stat_sum()` : somme les valeurs uniques ;
- `stat_summary()` : appliquer une fonction pour faire des *summaries* sur les valeurs de `y` ;
- `stat_unique()` : retire les valeurs dupliquées ;

5.2.4 Échelles (fonctions `scale_*()`)

Les fonctions `scale_*()` du *package* `ggplot2` permettent de définir et contrôler le *mapping* entre les données et les attributs esthétiques. Chaque paramètre esthétique possède son échelle, et sa fonction `scale_*()`. Encore une fois, le nom des fonctions est bien pensé, et tout est uniformisé. De fait, l'utilisateur peut, une fois qu'il a compris le fonctionnement d'une fonction, utiliser toutes les autres aisément.

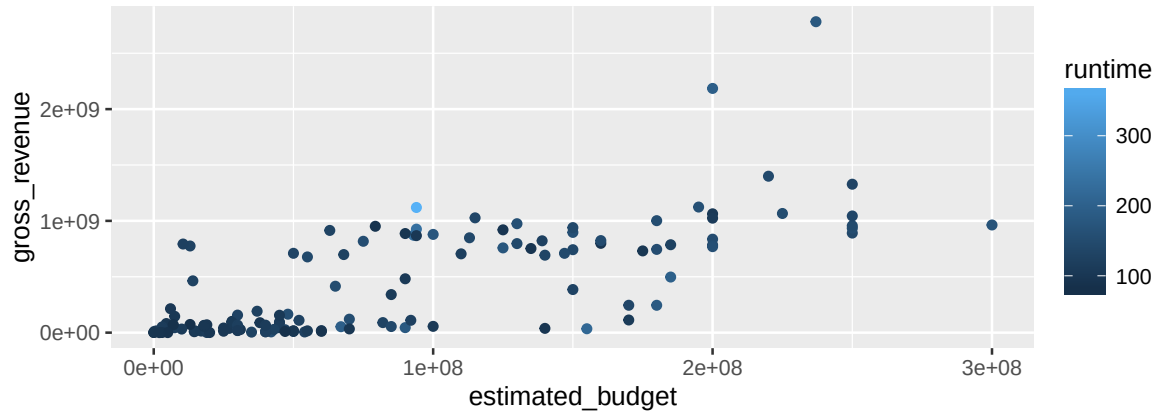
On peut diviser en quatre catégories les échelles :

1. de positions ;
2. de couleurs ;
3. manuelles discrètes ;
4. *identity* : pas de mise à l'échelle.

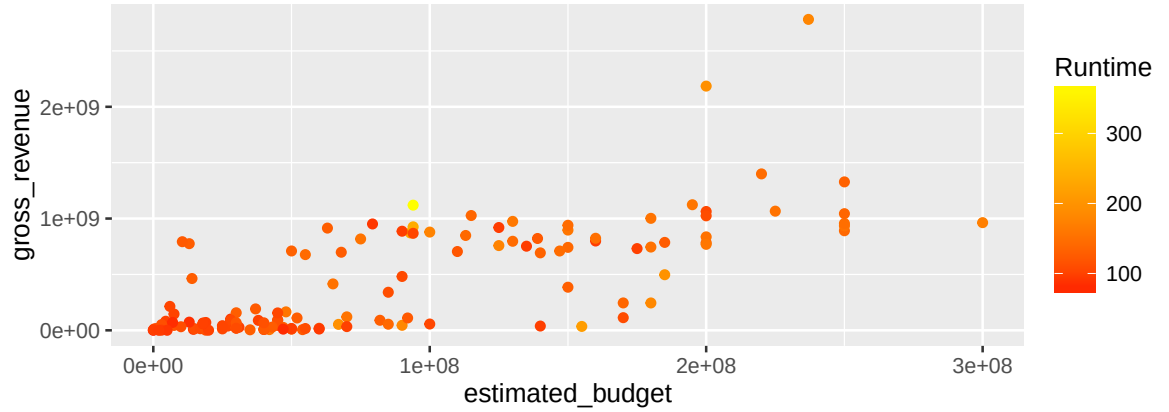
Ces échelles font donc le lien entre :
— le domaine (*domain*) : il s'agit de l'espace des données. Cet espace est caractérisé de manière différente selon la nature des données. En effet, si ces dernières sont discrètes, ce qui est le cas pour les facteurs, les logiques ou encore les chaînes de caractères, alors le domaine sera une énumération des valeurs possibles. En revanche, si les données sont continues, le domaine sera être défini par un intervalle ;
— la gamme (*range*) : il s'agit de l'espace des esthétiques. La gamme sera discrète lorsque le domaine l'est, et sera constituée des valeurs esthétiques correspondantes aux valeurs des données d'*input*. En revanche, lorsque le domaine est continue, la gamme est alors un chemin, pour indiquer comment passer d'une valeur à une autre.

Les échelles sont ajoutées automatiquement aux graphiques et l'utilisateur n'a rien à faire (c'est ce qui a été fait dans les sections précédentes). Cependant, la possibilité de changer ces échelles est offerte à l'utilisateur, par l'intermédiaire des fonction `scale_*()`. Les échelles dépendent du type de variable en *input*, à savoir continues (`numeric`) ou discrètes (`factor`, `character` ou `logical`). Pour ajouter ou modifier une échelle, il faut en construire une nouvelle, et l'ajouter au graphique, à l'aide, encore une fois, de l'opérateur `+`. Le nom des échelles est composé de deux ou trois parties : le préfixe `scale_`, puis le nom de l'esthétique auquel on ajoute un trait de soulignement, et enfin le nom de l'échelle à utiliser. Par exemple, pour modifier l'échelle qui gère la couleur, pour des données d'*input* continues, pour changer les couleurs du gradient, il faut utiliser la fonction `scale_colour_gradient()`.

```
p <- ggplot(data = films_reduit, aes(x = estimated_budget,
                                     y = gross_revenue, colour = runtime)) +
  geom_point()
p
```




```
# Changer l'échelle des couleurs pour que les films les plus courts
# soient en rouge (#FF0000), et les plus longs en jaune (#FFFF00),
# en créant un dégradé.
# On change également le titre de la légende
p + scale_colour_gradient(name = "Runtime", low = "#FF0000", high ="#FFFF00")
```



Le tableau ci-après répertorie les fonctions `scale_*()` en fonction du type d'esthétique et de la nature discrète ou continue des données.

Esthétique	Variable discrète	Variable continue
Transparence (`alpha`)	`scale_alpha_discrete()` `scale_alpha_manual()`	`scale_alpha_continuous()`
	`scale_alpha_identity()`	

Couleur (`colour`)	`scale_colour_discrete()`
	`scale_colour_brewer()`
	`scale_colour_grey()`
	`scale_colour_hue()`
	`scale_colour_manual()`
		`scale_colour_continuous()`
		`scale_colour_distiller()`
		`scale_colour_gradient()`
		`scale_colour_gradient2()`
		`scale_colour_gradientn()`

	`scale_colour_identity()`	

Remplissage (`fill`)	`scale_fill_discrete()`
	`scale_fill_brewer()`
	`scale_fill_grey()`
	`scale_fill_hue()`
	`scale_fill_manual()`
		`scale_fill_continuous()`
		`scale_fill_distiller()`
		`scale_fill_gradient()`
		`scale_fill_gradient2()`
		`scale_fill_gradientn()`

	`scale_fill_identity()`	
Type de ligne (`linetype`)	`scale_linetype_discrete()` `scale_linetype_manual()`	`scale_linetype_continuous()`
	`scale_linetype_identity()`	

continue sur la page suivante

suite du tableau

Esthétique	Variable discrète	Variable continue

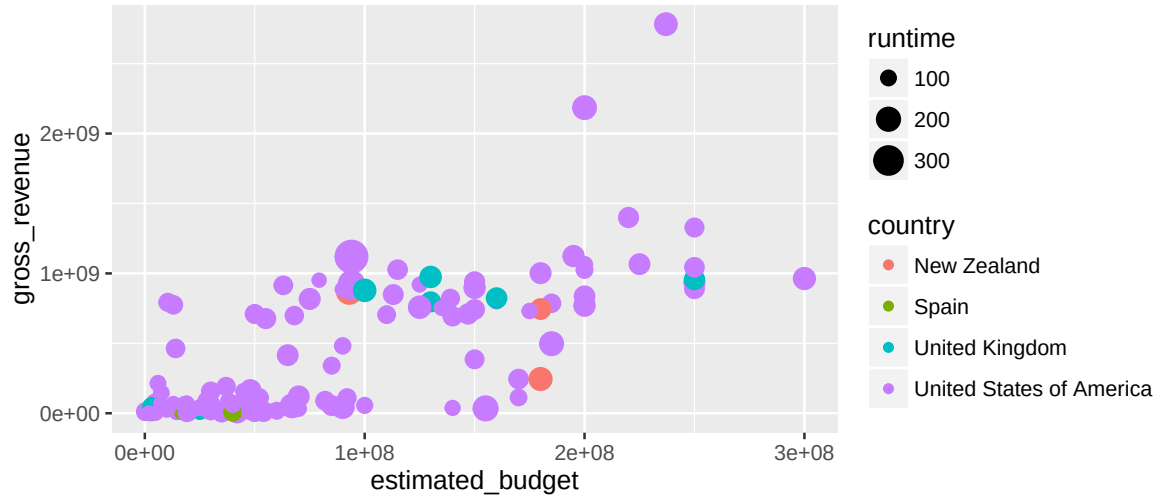
Forme	`scale_shape_discrete()`
	`scale_shape_manual()`
	`scale_shape_identity()`
		`scale_shape_continuous()`

Taille (`size`)	`scale_size_discrete()`
	`scale_size_manual()`
	`scale_size_identity()`
		`scale_size_continuous()`
		`scale_size_area()`

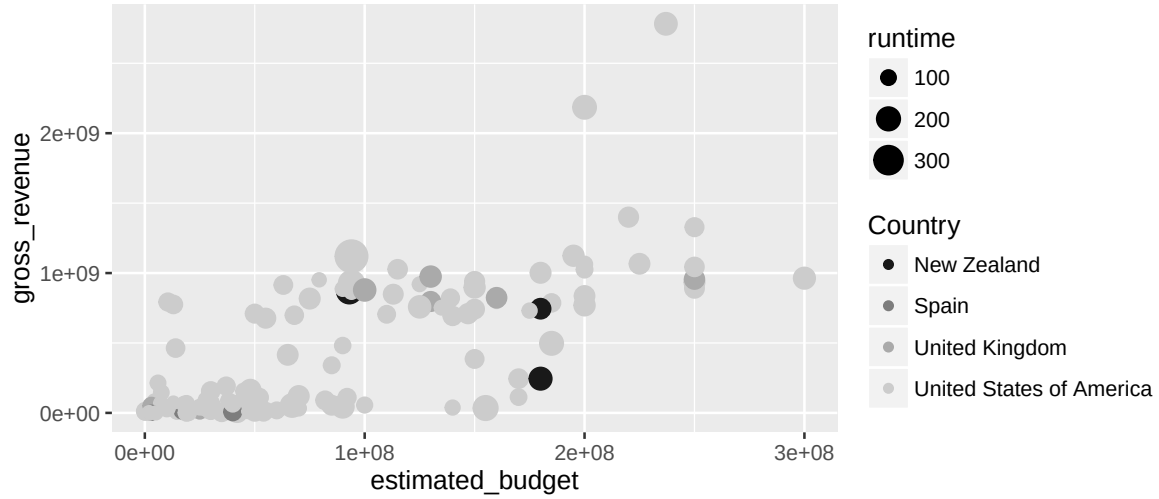
Position (`x`, `y`)	`scale_x_discrete()`,
	`scale_y_discrete()`
		`scale_x_continuous()`,
		`scale_y_continuous()`
		`scale_x_date()`,
		`scale_y_date()`
		`scale_x_datetime()`,
		`scale_y_datetime()`
		`scale_x_log10()`,
		`scale_y_log10()`
		`scale_x_reverse()`,
		`scale_y_reverse()`
		`scale_x_sqrt()`,
		`scale_y_sqrt()`

L'emploi de ces fonctions est simple, et les fichiers d'aide lèvent les éventuels doutes. Prenons cependent le temps de regarder un second exemple. Traçons un nuage de points représentant le revenu brut des films en fonction de leur budget ; faisons dépendre la couleur et la taille des points du pays d'origine, et de la longueur du film respectivement. La couleur dépend donc d'une variable discrète, et la taille d'une variable continue.

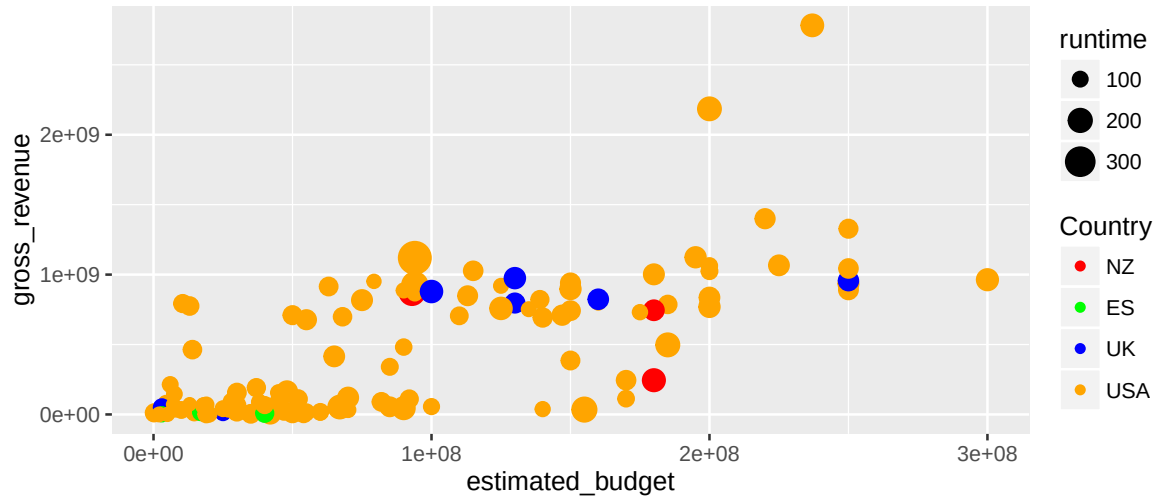
```
p <- ggplot(data = films_reduit, aes(x = estimated_budget,
                                     y = gross_revenue,
                                     colour = country,
                                     size = runtime)) +
  geom_point()

# Le graphique par défaut
p
```



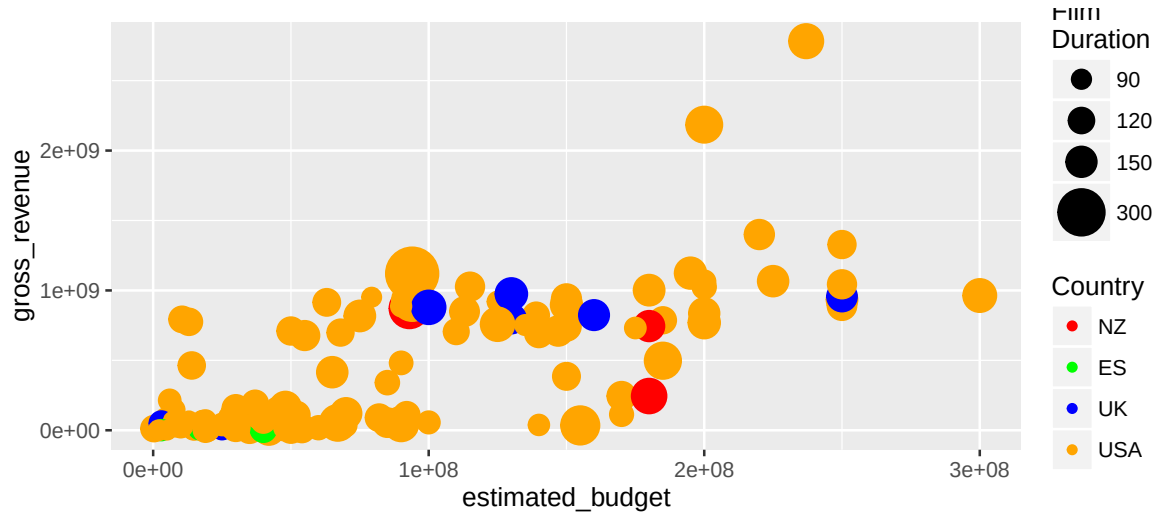
```
# Dégradé de gris pour la couleur, les valeurs manquantes en orange
# Le titre de la légende est aussi modifié
p + scale_colour_grey(name = "Country", start = .1, end = .8,
                      na.value = "orange")
```



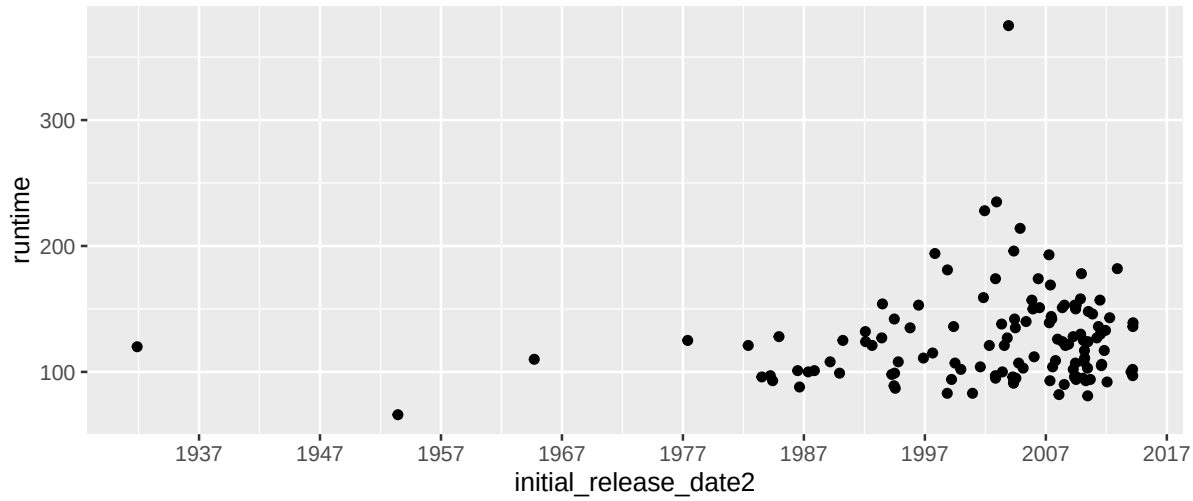

```
# Définissons nous-même la couleur pour chaque pays
# Si on souhaite en plus afficher un nom différent,
# il faut faire attention à bien effectuer le matching...
# La variable "country" est transformée en facteur,
# l'ordre est alphabétique (mais peut être changé avec la fonction order())
levels(factor(films_reduit$country))

## [1] "New Zealand"              "Spain"
## [3] "United Kingdom"           "United States of America"

# Une première solution
p + scale_colour_manual(name = "Country",
                        values = c("red", "green", "blue", "orange"),
                        labels = c("NZ", "ES", "UK", "USA"))
```



```
# Une seconde solution, plus longue, moins esthétique, mais plus sûre
p <- p + scale_colour_manual(name = "Country",
                             values = c("Spain" = "green", "New Zealand" = "red",
                                        "United States of America" = "orange",
                                        "United Kingdom" = "blue"),
                             labels = c("Spain" = "ES", "New Zealand" = "NZ",
                                        "United States of America" = "USA",
                                        "United Kingdom" = "UK"))


# Changeons également la taille
# En choisissant nous même les seuils
range(films_reduit$runtime)

## [1]  66 375

p + scale_size_continuous(name = "Film\nDuration",
                          breaks = c(0, 60, 90, 120, 150, 300, Inf),
                          range = c(1,10))
```

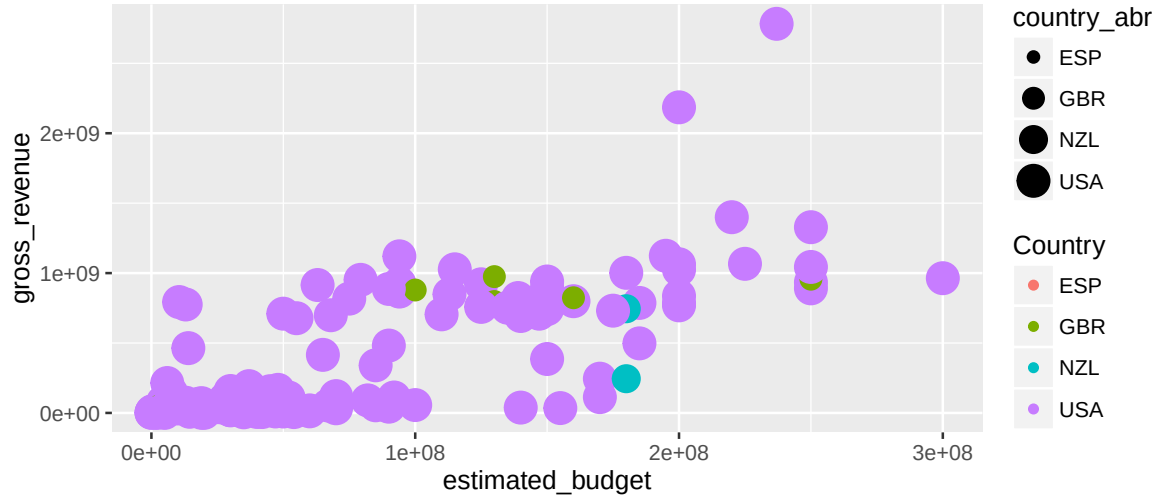



```
# Un autre graphique, avec des dates en abscisses.
# Le package scales offre de merveilleuses fonctions !
films_reduit$initial_release_date2 <- as.Date(films_reduit$initial_release_date)
p_2 <- ggplot(data = films_reduit,
              aes(x = initial_release_date2, y = runtime)) +
  geom_point()

# En affichant les dates tous les dix ans uniquement
library(scales)
p_2 + scale_x_date(breaks = date_breaks("10 year"), labels = date_format("%Y"))
```

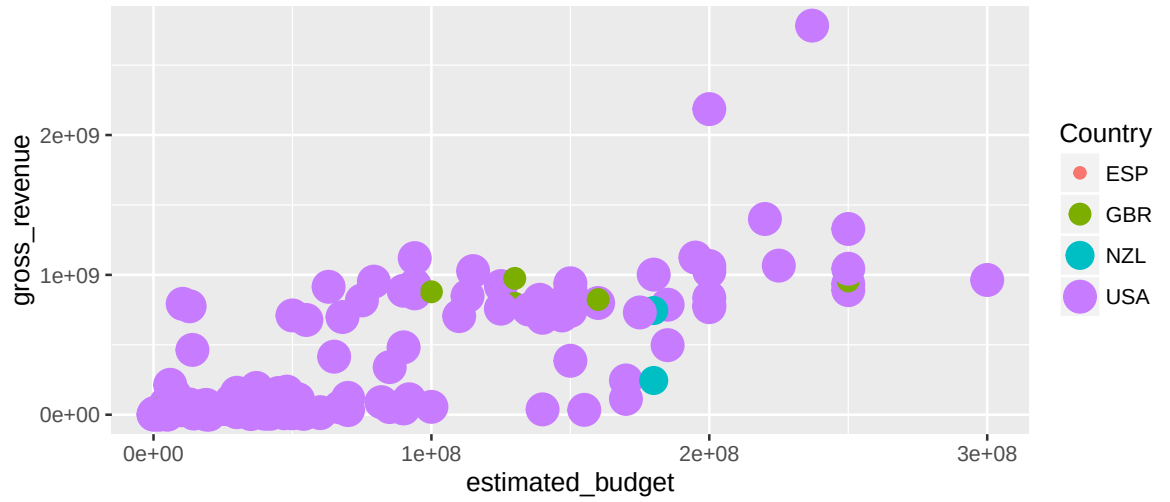


Les légendes affichées sont fonction du *mapping* réalisé, et `ggplot2` essaie de combiner les légendes autant que possible. Par exemple, si le paramétrage de la couleur et de la forme concernent la même variable, au lieu d'afficher deux légendes distinctes (une pour la couleur, et une pour la forme), une seule combinant les deux paramètres esthétiques sera affichée. Par ailleurs, il est nécessaire que les légendes qui sont combinées possèdent le même nom. Il est important d'avoir conscience de cela : en effet, pour changer le titre d'une légende, si elle est le résultat d'une combinaison, il est nécessaire de le changer pour toutes les légendes qui composent la combinaison.

```
p <- ggplot(data = films_reduit,
            aes(x = estimated_budget, y = gross_revenue,
                colour = country_abr, size = country_abr)) +
              geom_point()

p + scale_colour_discrete(name = "Country")
```



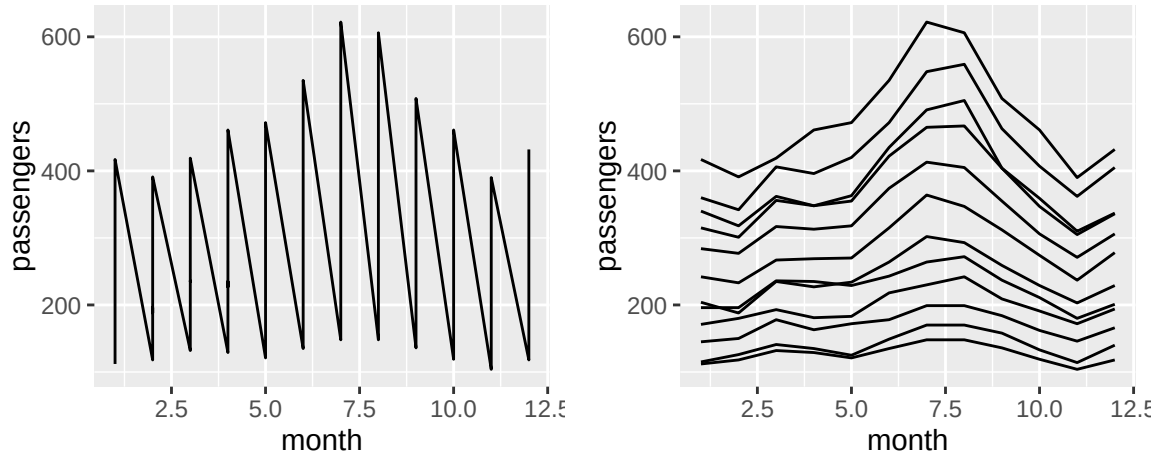

```
p + scale_colour_discrete(name = "Country") +
  scale_size_discrete(name = "Country")
```



5.2.5 Groupes

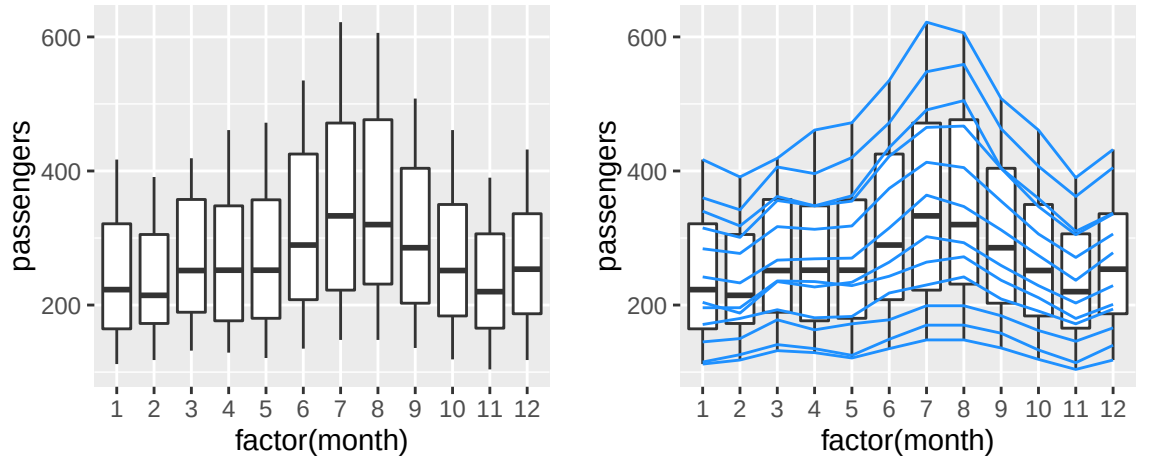
Dans de nombreux cas, `ggplot2` effectue des regroupements automatiquement, lorsqu'il y a besoin de tracer des objets géométriques. Par exemple, lorsque l'on souhaite tracer des points, chaque groupe est constitué d'un seul point. Pour tracer des polygones, chaque groupe est constitué de plusieurs points qu'il faut relier. Le groupement s'effectue en fonction des interactions de toutes les variables qualitatives d'un graphique. Parfois, il est nécessaire de définir à nouveau les groupes (par exemple, si aucune variable discrète n'est appelée dans le graphique). Pour ce faire, il faut préciser quelle variable doit permettre le regroupement, en l'indiquant au paramètre `group`. L'exemple qui suit s'appuie sur les données `AirPassengers`, qui renseignent le nombre de passagers aériens annuels entre 1949 et 1960. Si on désire afficher le nombre de passagers en fonction des mois de l'année, en superposant les courbes par année, le paramètre `group` nous le permet.

```
library(reshape2)

df <- data.frame(year = rep(1949:1960, each = 12),
           month = rep(1:12, 12),
           passengers = c(AirPassengers))

head(df)

##   year month passengers
## 1 1949     1        112
## 2 1949     2        118
## 3 1949     3        132
## 4 1949     4        129
## 5 1949     5        121
## 6 1949     6        135

# Sans préciser de groupe
# Comme il n'y a pas de variable discrète, on obtient
ggplot(data = df, aes(x = month, y = passengers)) + geom_line()

# En précisant que l'on souhaite regrouper par années
ggplot(data = df, aes(x = month, y = passengers, group = year)) + geom_line()
```



Si on veut tracer des lignes qui relient des points au sein d'un même groupe, alors que l'échelle est discrète, il est tout à fait possible d'indiquer un changement du groupement par défaut. Il suffit alors de préciser la valeur du groupe au paramètre `group` dans la fonction `aes()` de la fonction `geom_line()`.

```
# Boxplot du nombre de passager pour chaque mois
ggplot(data = df, aes(x = factor(month), y = passengers)) + geom_boxplot()

# En ajoutant les lignes de chaque année
ggplot(data = df, aes(x = factor(month), y = passengers)) +
  geom_boxplot() +
  geom_line(aes(group = year), colour = "dodger blue")
```



5.2.6 Annotations

L'annotation sur un graphique peut passer par du texte, ou bien par l'ajout d'objets géométriques (e.g. des lignes ou des rectangles).

Les exemples qui suivent vont s'appuyer sur le graphique suivant.

```
p <- ggplot(data = films_reduit,
            aes(x = estimated_budget, y = gross_revenue)) +
  geom_point()
```

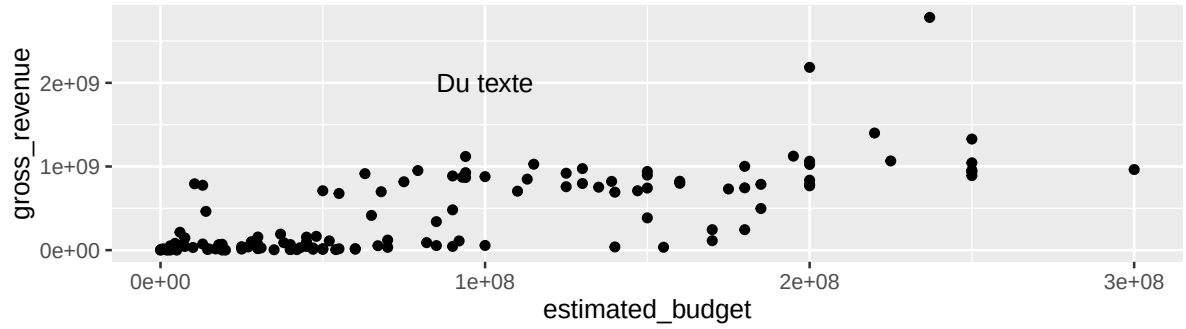
5.2.6.1 Texte

Pour ajouter du texte sur les graphiques, il existe deux fonctions dans le *package* `ggplot` : `geom_text()` et `annotate()`, bien que la dernière soit plus générique (elle permet d'ajouter d'autres éléments que du texte). Nous ne présentons que la manière d'utiliser `annotate()`, dans la mesure où elle offre des performances accrues comparativement à `geom_text()`.

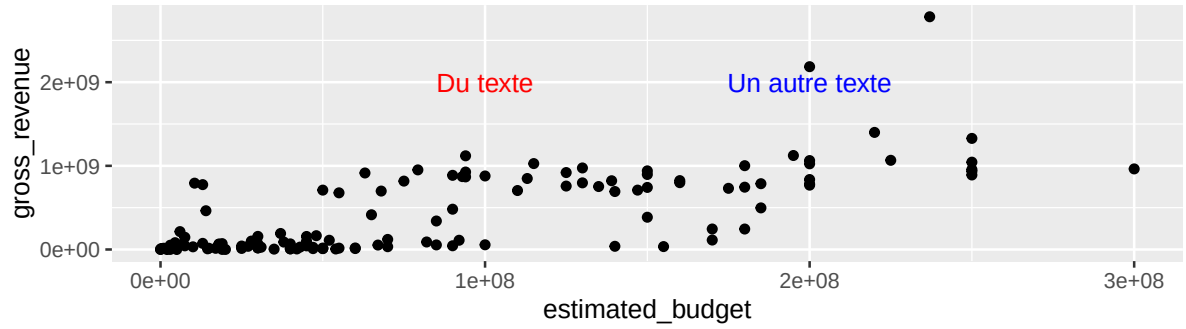
La fonction `annotate()`, dans le cas général, créé une nouvelle couche qui contient les annotations. Les données pour faire le lien avec les paramètres esthétiques ne sont pas celles du *data.frame*, mais contenues dans un vecteur qu'il est nécessaire de renseigner. Par ailleurs, la forme géométrique est à définir *via* le paramètre `geom` (`"text"` pour du texte, ou encore `"segment"` pour un segment, etc.). Enfin, comme pour les couches de type `geom_*()`, on peut rajouter des paramètres esthétiques.

Pour ajouter du texte, on précise la position `x` et `y` (ou les positions, si on désire afficher plusieurs textes), le paramètre `geom` doit recevoir la valeur `"text"` et le paramètre `legend` se voit attribuer la chaîne à afficher (ou les chaînes en cas d'affichage de plusieurs textes) [3]. Si on souhaite afficher le texte d'une certaine couleur, on peut ajouter le paramètre `colour` dans la fonction `annotate()`.

```
# Ajout d'un texte simple
p + annotate("text", x = 1e8, y = 2e9, label = "Du texte")
```




```
# Ajoute de deux textes, le premier en rouge, le second en bleu
p + annotate("text", x = c(1e8, 2e8), y = 2e9,
             label = c("Du texte", "Un autre texte"),
             colour = c("red", "blue"))
```

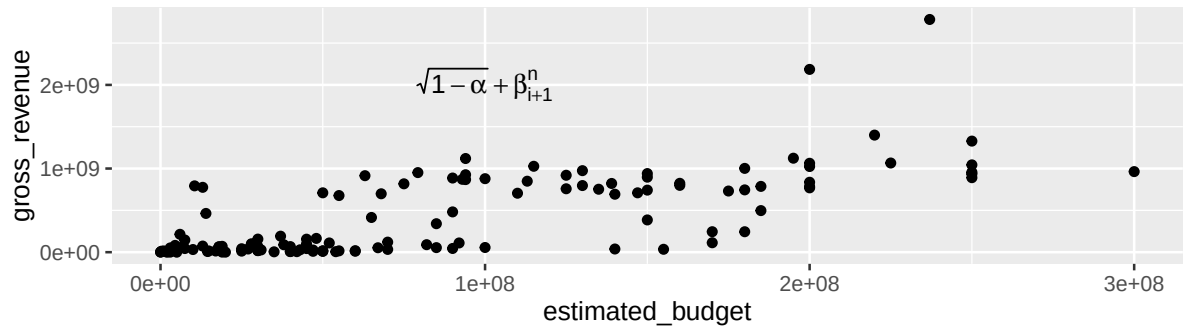


Dès lors que des lettres grecques doivent apparaître, ou bien des écritures en indice ou exposant, ou encore des symboles mathématiques, on peut rajouter le paramètre `parse` évalué à `TRUE`. Pour faire apparaître une lettre grecque, il suffit d'écrire son nom. Pour écrire en indice, on

3. Les règles de recyclage s'appliquent !

utilise les crochets [] et pour écrire en exposant, l'accent circonflèxe ^. Attention, pour ajouter un terme en indice et un en exposant, il est nécessaire de le faire dans l'odre suivant : d'abord en indice, puis en exposant.

```
p + annotate("text", x = 1e8, y = 2e9,
             label = "sqrt(1-alpha) + beta[i+1]^n", parse = TRUE)
```



Remarque 5.2.5

Il existe un *package* pour convertir des expressions LaTeXen expressions lisibles par R : latex2exp [a].

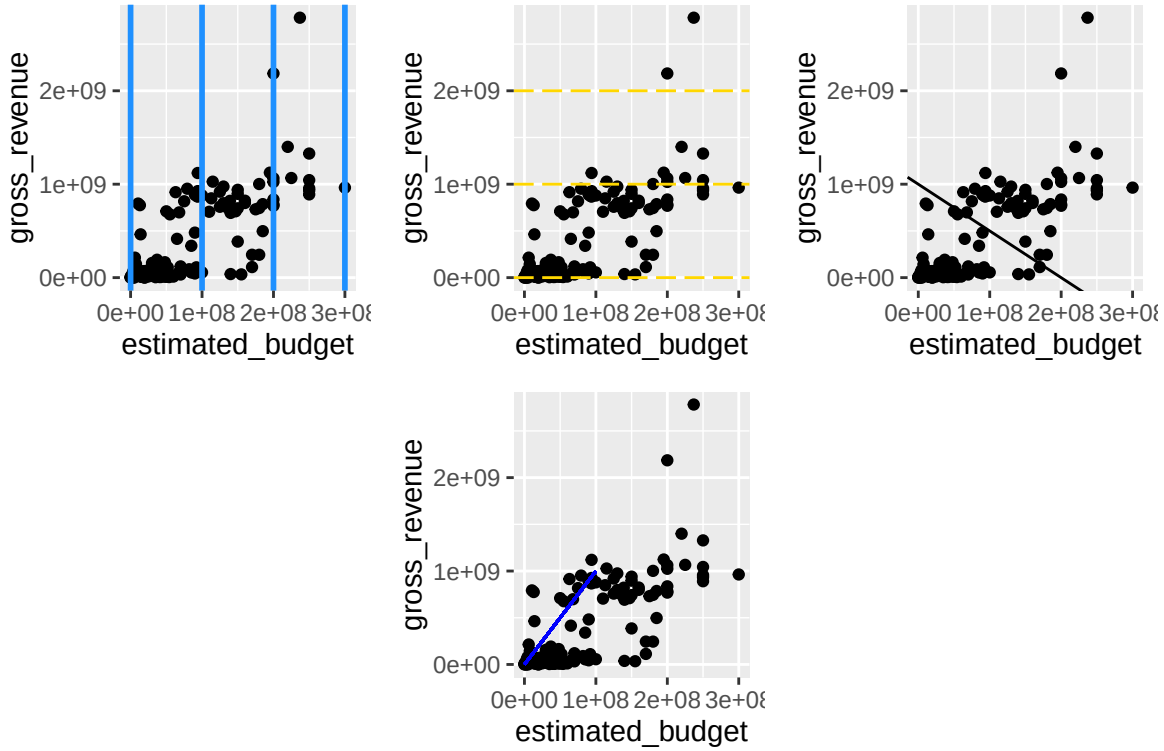
a. https://github.com/stefano-meschiari/latex2exp

5.2.6.2 Lignes

Le *package* ggplot2 contient quatre fonctions très utile pour ajouter des lignes sur un graphique, en plus des fonctions geom_line() et geom_path() :

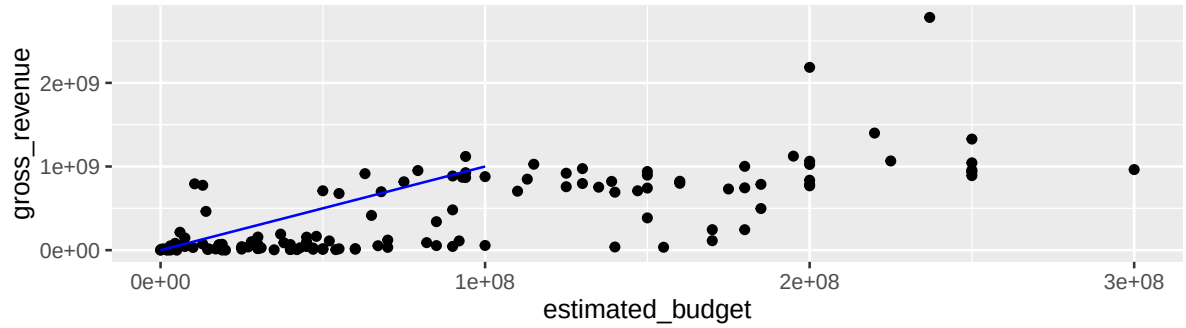
- geom_vline() : ligne verticale ;
- geom_hline() : ligne horizontale ;
- geom_abline() : ligne spécifiée par sa pente et son ordonnée à l'origine ;
- geom_segment() : segment ou flèche (en utilisant arrow()).

```
# Ligne verticale
p + geom_vline(xintercept = seq(0, 3e8, by = 1e8),
               size = 1, col = "dodger blue")

# Ligne horizontale
p + geom_hline(yintercept = seq(0, 2e9, by = 1e9),
               col = "gold", linetype = "longdash")

# Droite
p + geom_abline(intercept = 1e9, slope = -5)

# Segment
p + geom_segment(aes(x = 0, xend = 1e8, y = 0, yend = 1e9), col = "blue")
```



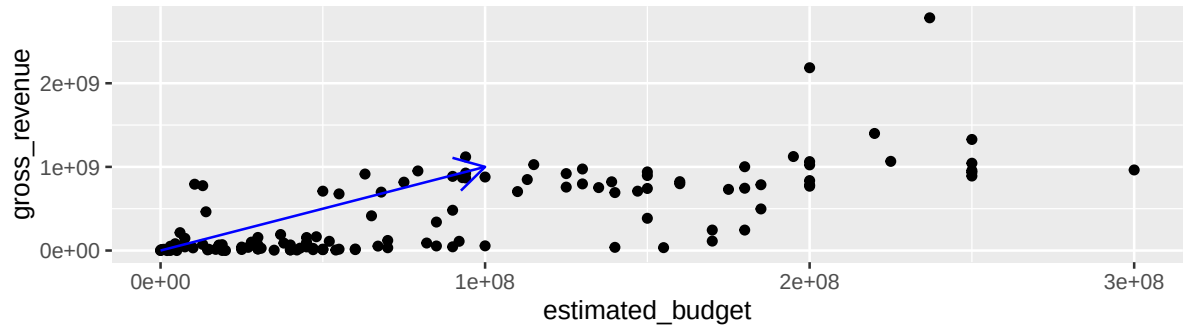
Pour les segments, il peut être plus intéressant d'utiliser la fonction `annotate()`, afin de s'affranchir de tout héritage des paramètres de la fonction `ggplot()`.

```
p + annotate(geom = "segment", x = 0, xend = 1e8,
             y = 0, yend = 1e9, col = "blue")
```



Pour tracer une flèche, il faut avoir recours à la fonction `arrow()` du *package* `grid`.

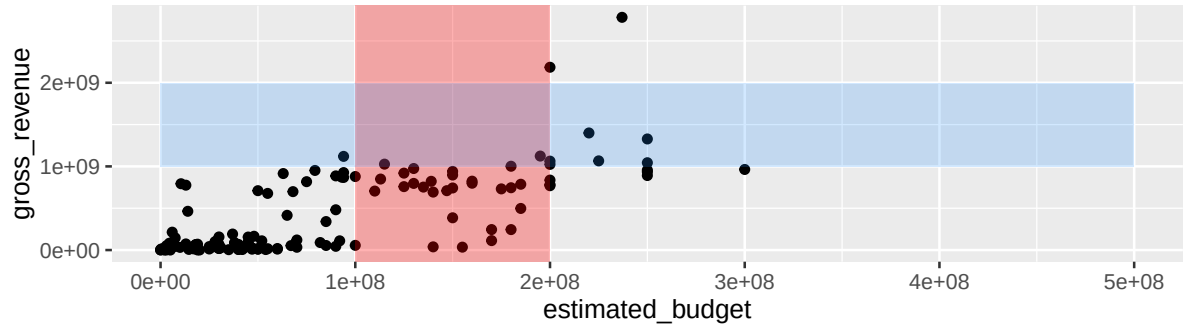
```
library(grid)
p + annotate(geom = "segment", x = 0, xend = 1e8,
             y = 0, yend = 1e9, col = "blue",
             arrow = arrow(length = unit(0.5, "cm")))
```



5.2.6.3 Rectangles

Pour ajouter un rectangle à un graphique, ce qui peut être pratique avec des données temporelles par exemple (e.g. pour faire figurer les périodes de croissance ou de récession), on peut utiliser soit la fonction `geom_rect()`, soit à nouveau la fonction `annotate()`. Les deux méthodes fonctionnent à peu près de la même façon, mais encore une fois, `annotate()` permet de ne pas subir les héritages des paramètres esthétiques.

```
p + annotate(geom = "rect", xmin = 1e8, xmax = 2e8, ymin = -Inf, ymax = Inf,
             alpha = .3, fill = "red") +
  annotate(geom  ="rect", xmin = 0, xmax = 5e8, ymin = 1e9, ymax = 2e9,
           alpha = .2, fill = "dodger blue")
```



5.2.7 Positions

Pour modifier le positionnement de certains éléments dans les graphiques, `ggplot2` propose cinq fonctions :

- `position_dodge()` : évite les chevauchements, place les éléments côte à côte ;
- `position_fill()` : empile les éléments qui se chevauchent, en normalisant pour avoir une hauteur égale ;
- `position_identity()` : n'ajuste pas la position ;
- `position_jitter()` : place les éméments côte à côte en essyant d'optimiser l'espace ;
- `position_stack()` : empile les éléments qui se chevauchent.

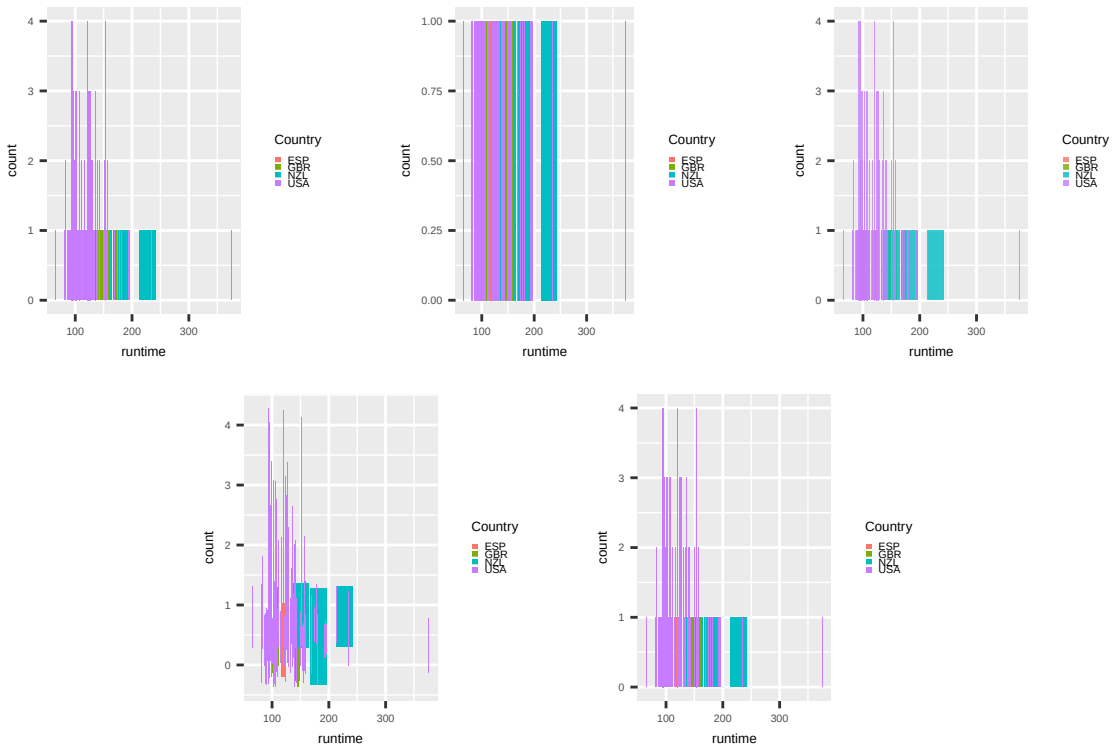
La manière la plus simple d'utiliser ces fonctions est de les appeler à travers le paramètre `position` d'une fonction `geom_*()`, en précisant uniquement le suffixe. Par exemple, avec un diagramme en bâton, pour que les barres soient côte à côte, on indique `position = "dodge"` à la fonction `geom_bar()`.

Encore une fois, même s'il est possible de réaliser une action avec une fonction de `ggplot2`, cela ne veut pas forcément dire que cette action est adaptée à une lecture facilitée. Ainsi, lorsque l'on fournit la valeur `"identity"` au paramètre `position` de la fonction `geom_bar()`, les barres

s'entre-mêlent, et certaines peuvent devenir invisibles, cachées par une autre.

Les exemples ci-après montrent toutes les possibilités de placement pour un diagramme en bâton.

```
p <- ggplot(data = films_reduit, aes(x = runtime, fill = country_abr))

# Dodge
p + geom_bar(position = "dodge")

# Fill
p + geom_bar(position = "fill")

# Identity (pas pratique avec un barchart)
p + geom_bar(position = "identity", alpha = .8)

# Jitter
p + geom_bar(position = "jitter")

# Stack
p + geom_bar(position = "stack")
```



5.2.8 Facettes

Tout comme pour `qplot()`, on peut créer des facettes, c'est-à-dire produire des graphique de même type pour différentes sous-divisions du *data.frame*, et les aligner sur une grille. `ggplot2` propose deux fonctions pour faire du *fateting* : `facet_grid` (produisant une grille à deux dimensions, pour laquelle les variables définissent les lignes et colonnes) et `facet_wrap` (produisant les graphiques pour chaque sous-division, qui sont ensuite placés sur une grille à deux dimension,

les uns après les autres).

Les deux paramètres principaux de ces fonctions sont : les variables servant à faire le *faceting*, sour forme de formule, et un logique indiquant si les échelles de positions doivent être globales ou bien locales à chaque facette.
Pour être plus précis avec les exemples, nous avons besoin d'une seconde variable qualitative avec peu de modalités. Prenons arbitrairement une variable indicatrice, valant `nouveau` pour les films produits après 2000, et `ancien` sinon.

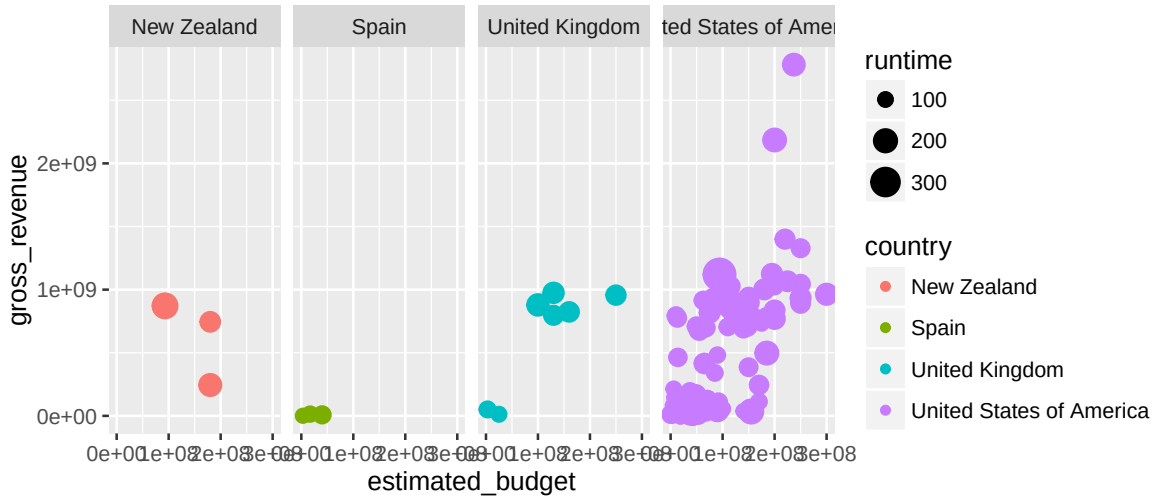
```
films_reduit$old <- ifelse(films_reduit$year <= 2000, "ancien", "nouveau")
```

5.2.8.1 La fonction `facet_grid()`

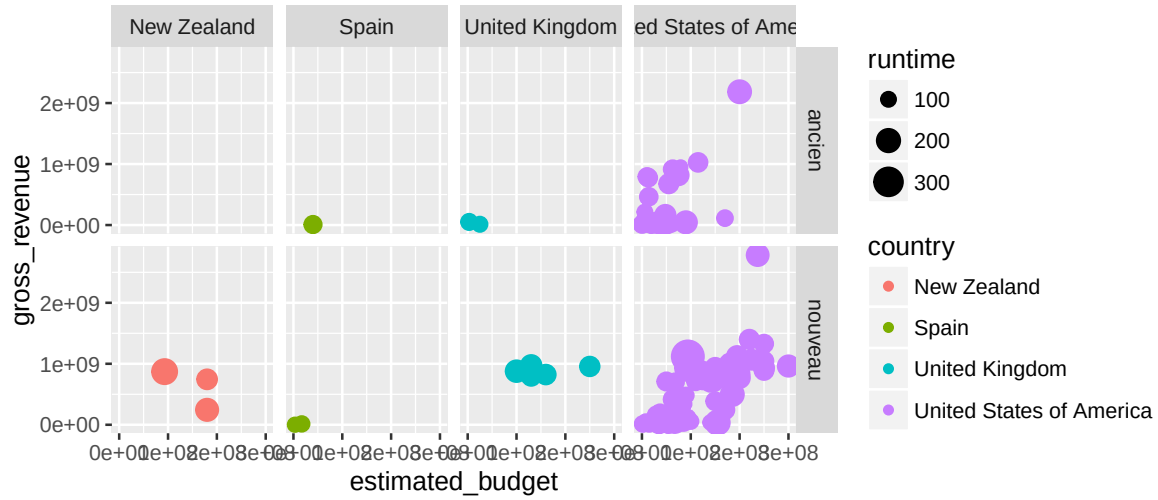
Pour créer une grille sur laquelle les colonnes et les lignes servent à différentier différentes modalités de variables, on utilise la fonction `facet_grid`. Le paramètre `facets` attend une formule, dont la syntaxe est la suivante : `variable_ligne ~ variable_colonne`, avec `variable_ligne` le nom de la variable facteur en ligne, et `variable_colonne` le nom de la variable facteur en colonne. Le symbole `.` indique que l'on ne souhaite pas créer de facette sur la dimension :

- `. ~ .` (la valeur par défaut) : ne pas faire de *faceting* ;
- `. ~ variable_colonne` : une ligne, autant de colonnes que de valeurs possibles pour `variable_colonne` ;
- `variable_ligne ~ .` : autant de lignes que de valeurs possibles pour `variable_ligne`, une seule colonne ;
- `variable_ligne ~ variable_colonne` : autant de lignes que de valeurs possibles pour `variable_ligne`, autant de colonnes que de valeurs possibles pour `variable_colonne`.

```
p <- ggplot(data = films_reduit, aes(x = estimated_budget,
                                     y = gross_revenue,
                                     colour = country,
                                     size = runtime)) +
  geom_point()

# Avec une colonne pour chaque pays
p + facet_grid(. ~ country)
```

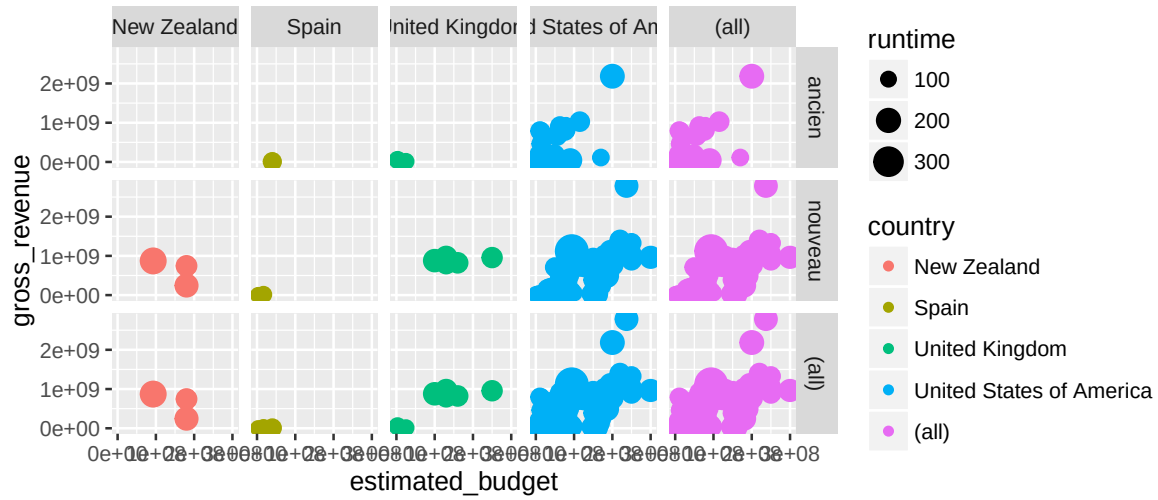


```
# Avec en ligne la récence du film, et en colonne le pays
p + facet_grid(old ~ country)
```

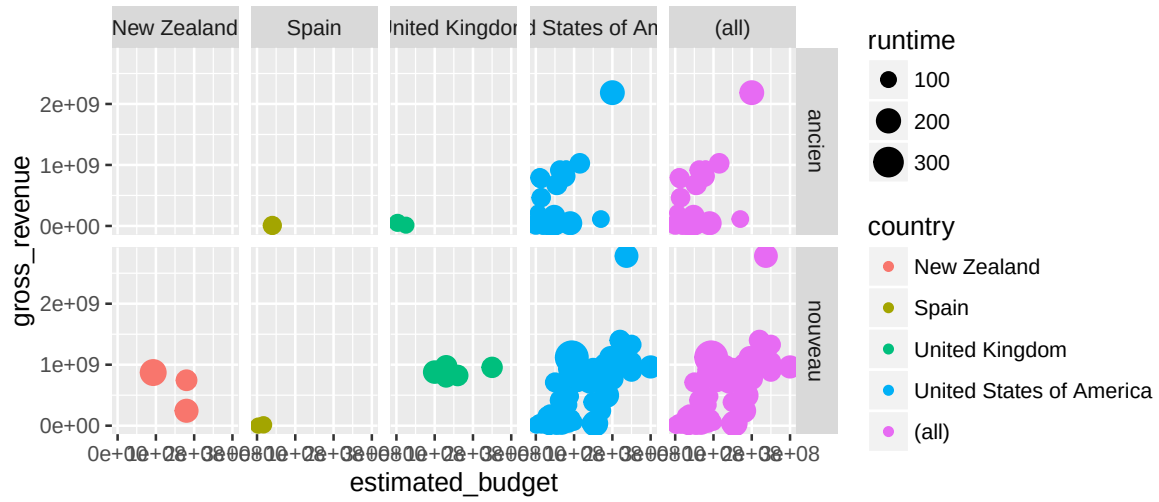


On peut ajouter une option pour avoir les situations marginales (similairement au tableau obtenu, dans le cas de deux variables aléatoires discrètes, faisant figurer les lois marginales en bout de ligne et de colonne). Pour ce faire, il faut attribuer la valeur `TRUE` au paramètre `margins`. Si on souhaite afficher uniquement les facettes supplémentaires pour une variable mais pas les deux, il faut fournir le nom de cette variable au paramètre `margins`.

```
p + facet_grid(old ~ country, margins = TRUE)
```



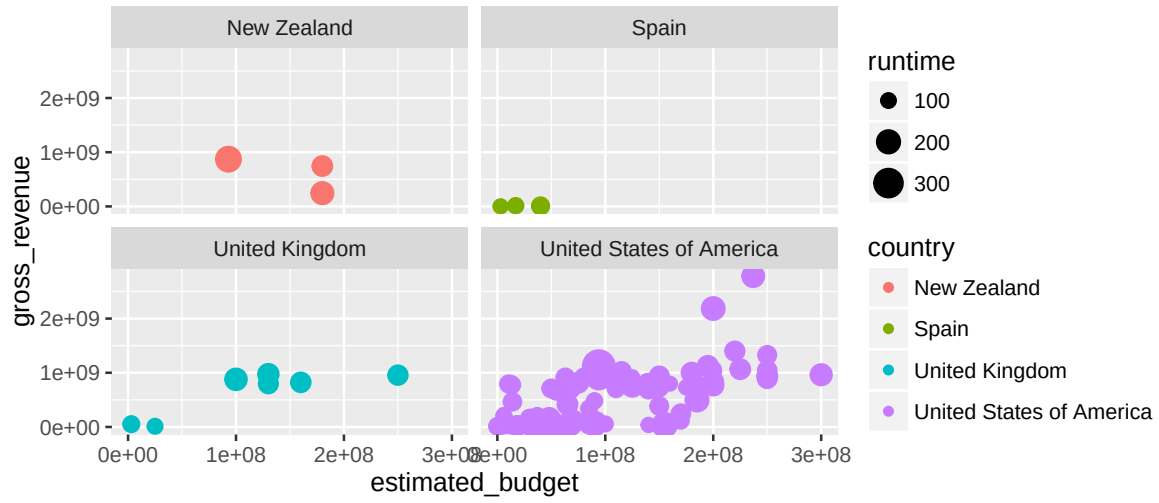

```
# Ajout d'une facette avec tous les pays, en différenciant par ligne (récence)
p + facet_grid(old ~ country, margins = "country")
```



5.2.8.2 La fonction `facet_wrap()`

La fonction `facet_wrap` fonctionne un peu différemment de `facet_grid`. En effet, au lieu de travailler en fonction de variables par lignes et par colonnes, elle crée une succession de graphiques qui sont **par la suite** placés sur une grille à deux dimensions. Pour créer les facette, il faut renseigner une formule au paramètre `facets`. La syntaxe pour la formule est la suivante : `~variable_1 + variable_2 + ... variable_n`. La grille finale sera ce qui se rapproche le plus d'un carré. Dans les cas où ce n'est pas possible, la grille sera plutôt large que longue (nos écrans sont généralement configurés ainsi, ce qui explique ce choix).

```
# Avec une seule variable
p + facet_wrap(facets = ~ country)
```



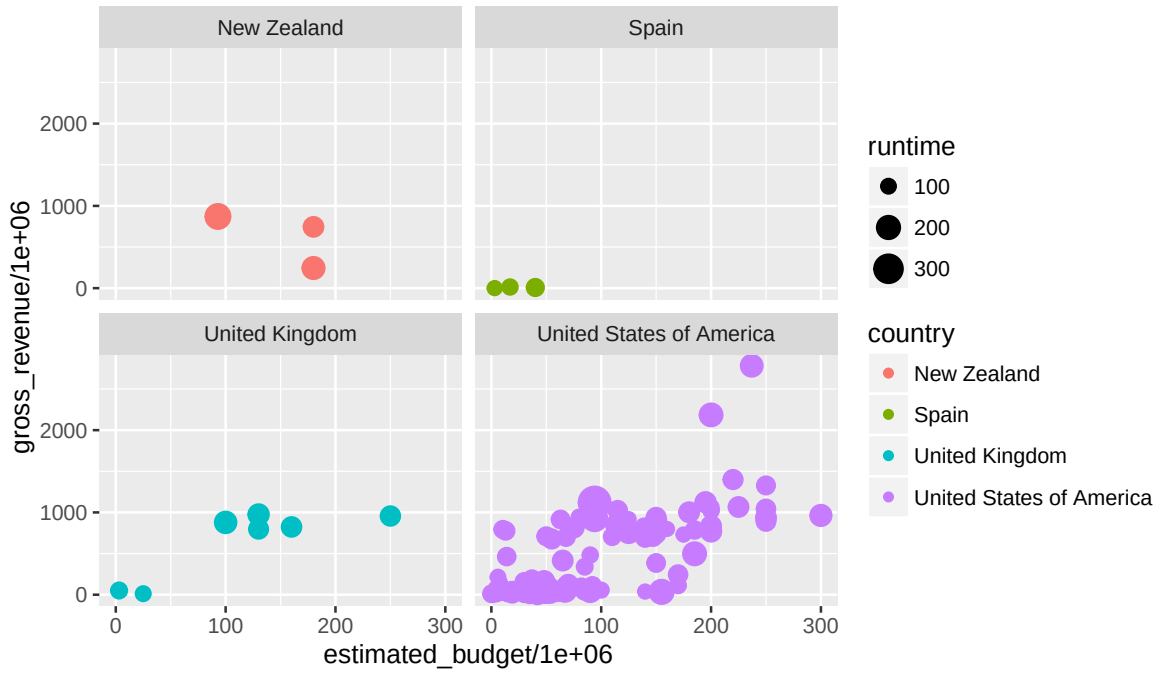

```
# Avec deux variables
p + facet_wrap(facets = ~ country + old)
```

5.2.8.3 Échelles et facettes

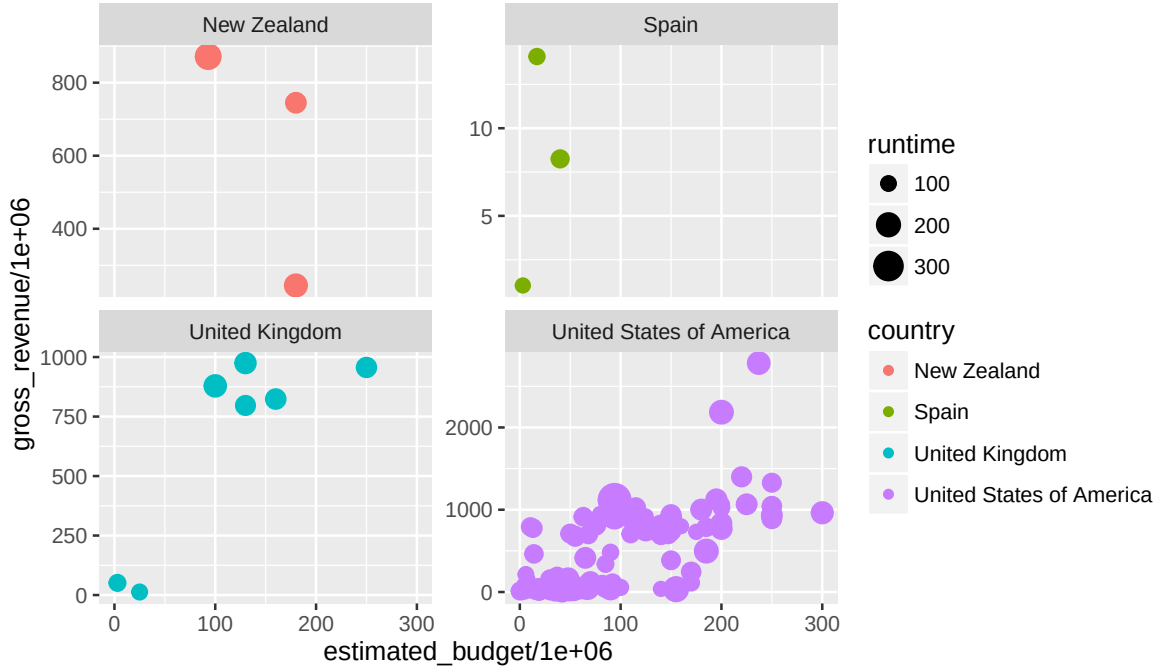
Que ce soit pour `facet_grid()` ou `facet_wrap()`, il est possible de préciser si les échelles de positions doivent s'appliquer à tous les graphiques de la grille, ou bien doivent varier. Cela se fait *via* le paramètre `scales`. On distingue quatre options :

- `fixed` : les échelles les fixes, identiques pour chaque graphique ;
- `free` : les échelles peuvent varier en fonction de chaque graphique de la grille ;
- `free_x` : seule l'échelle pour les `x` peut varier, l'échelle pour les `y` est fixe, commune à chaque graphique de la grille ;
- `free_y` : seule l'échelle pour les `y` peut varier, l'échelle pour les `x` est fixe, commune à chaque graphique de la grille.

```
p_m <- ggplot(data = films_reduit,
       aes(estimated_budget/1000000,
           gross_revenue/1000000,
           colour = country,
           size = runtime)) +
  geom_point()

# Toutes les échelles identiques
p_m + facet_wrap( ~ country, scales = "fixed")
```




```
# Échelles variant pour chaque graphique de la grille
p_m + facet_wrap( ~ country, scales = "free_y")
```



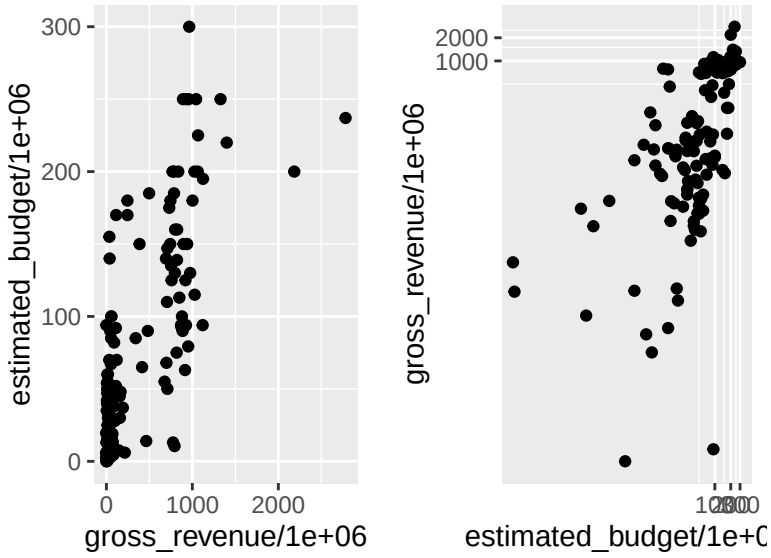
5.2.9 Coordonnées

Le *package* `ggplot2` gère plusieurs systèmes de coordonnées différents, le système cartésien étant celui utilisé par défaut. Pour définir le système souhaité, il faut faire appel aux fonctions commençant par `coord_` :

- `coord_cartesian()` : coordonnées cartésiennes ;
- `coord_fixed()` : coordonnées cartésiennes avec la même échelle pour les deux axes ;
- `coord_flip()` : coordonnées cartésiennes avec les axes renversés ;

- `coord_map()` : projections pour les cartes ;
- `coord_polar()` : coordonnées polaires ;
- `coord_trans()` : coordonnées cartésiennes transformées.

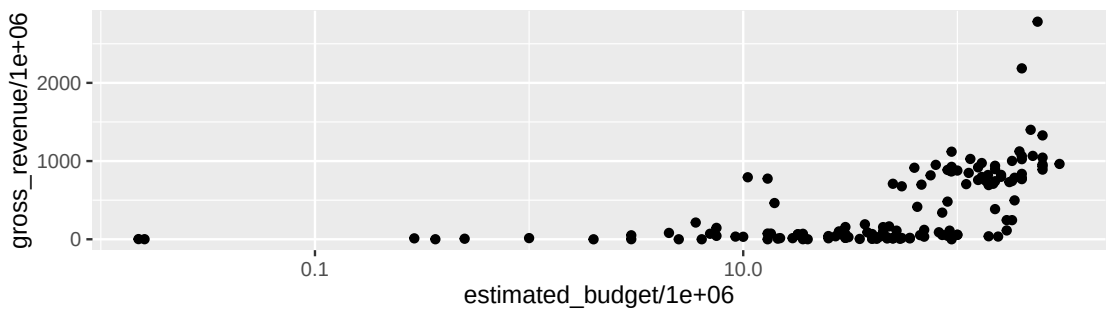
```
p <- ggplot(data = films,
            aes(x = estimated_budget/1e6, y = gross_revenue/1e6)) +
  geom_point()

# En renversant les axes
p + coord_flip()

# En effectuant une transformation du système de coordonnées
p + coord_trans(x = "log10", y = "log10")
```



Remarque 5.2.6

Nous avons vu que la fonction `scale_x_log10()` (section 5.2.4) permet aussi de changer les échelles en échelles logarithmiques. Cependant, il y a une petite différence : avec `coord_trans`, la transformation est réalisée après la réalisation de statistiques. Cela a un impact, par exemple, sur les lignes du quadrillage

```
p + scale_x_log10()
```



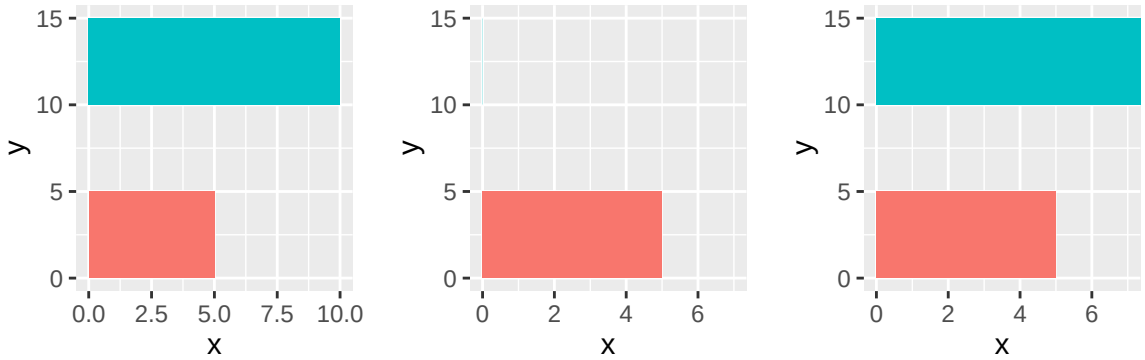
5.2.10 Titres, axes, légendes

Jusqu'ici, les graphiques proposés en exemple souffrent pour la plupart de l'absence de titre, de légendes illisibles, etc. Heureusement, tout cela est paramétrable !
Pour ajouter un titre, rien de plus simple : faire appel à la fonction `ggtitle()`, et fournir le titre au paramètre `label`. Pour changer les étiquettes des axes, on peut faire appel aux fonctions `xlab()` et `ylab()` pour l'axe des `x` et des `y` respectivement, et renseigner le paramètre `label`.

```
p <- ggplot(data = films,
            aes(x = estimated_budget/1e6, y = gross_revenue/1e6)) +
  geom_point() + ggtitle("Titre") +
  xlab("Étiquette axe des x") + ylab("Étiquette axe des y")
```

Pour définir manuellement les limites des axes, on peut utiliser les fonctions `xlim()` et `ylim()`. Attention toutefois, les valeurs en dehors des intervalles définis seront complètement jetées, les autres couches hériteront des données appartenant aux intervalles définis. Pour changer les limites des axes sans jeter de données, il faut faire appel à la fonction `coord_cartesian()`.

```
df <- data.frame(x = c(0, 0, 5, 5, 0, 0, 10, 10),
           y = c(0,5, 5, 0, 10, 15, 15, 10),
           g = factor(rep(1:2, each = 4)))

p_2 <- ggplot(data = df, aes(x = x, y = y, group = g, fill = g)) +
  geom_polygon()

# Le graphique par défaut
p_2

# En jouant avec la limite des x avec la fonction xlim()
p_2 + xlim(0, 7)

# Avec la fonction coord_cartesian()
p_2 + coord_cartesian(xlim = c(0,7))
```

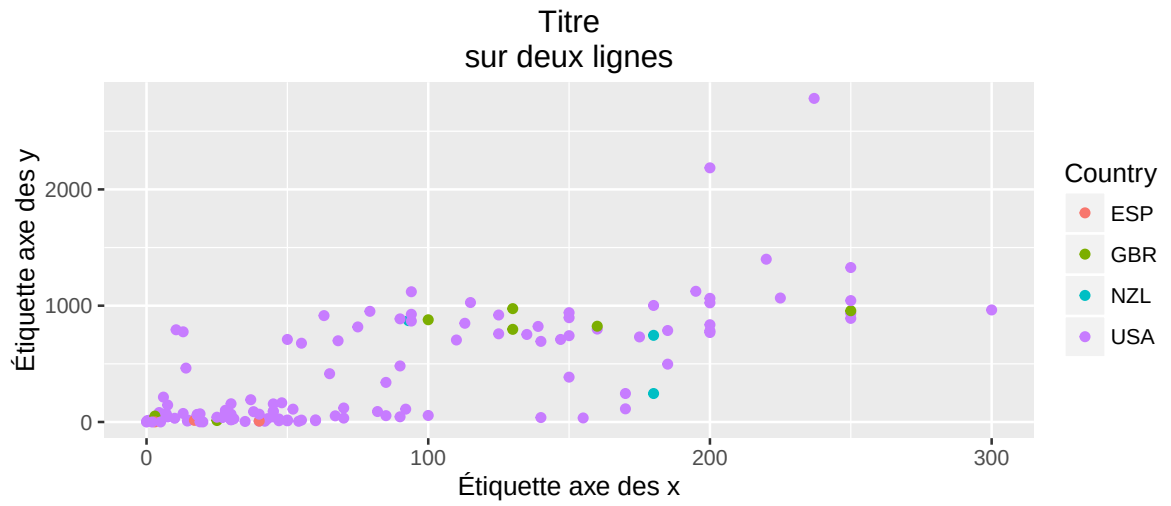


Le changement des paramètres des légendes est un peu moins simple. Il faut passer par la fonction `theme()`. Le lecteur est invité à consulter la page d'aide de la fonction à l'adresse suivante : http://docs.ggplot2.org/current/theme.html. Dès que l'on souhaite modifier l'aspect de la légende, des marques des axes, des étiquettes des axes, des lignes principales et secondaires formant le quadrillage, etc. il faut utiliser la fonction `theme()`.

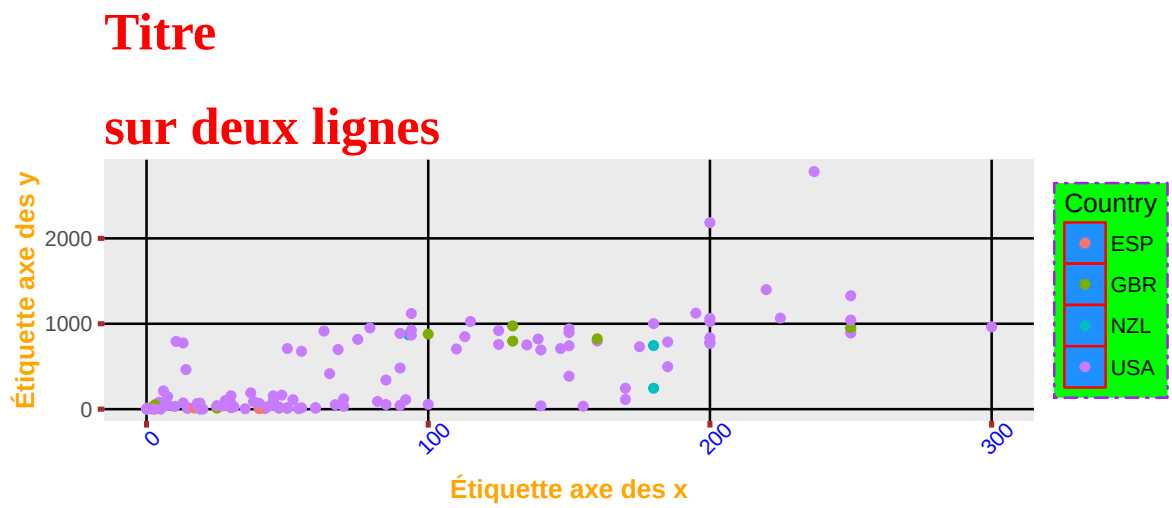
Les éléments pouvant être changés sont nombreux. Il y a quatre types de fonctions pour modifier les éléments :

- `element_text()` : pour toutes les étiquettes, ce qui est au format texte. On peut modifier :
  - ▷ `family` : la famille de la fonte,
  - ▷ `face` : graisse (`"plain"`, `"italic"`, `"bold"`, `"bold.italic"`),
  - ▷ `colour` : couleur,
  - ▷ `size` : taille en pts,
  - ▷ `hjust` : justification horizontale, dans $[0, 1]$,
  - ▷ `vjust` : justification verticale, dans $[0, 1]$,
  - ▷ `angle` : angle, dans $[0, 360]$,
  - ▷ `lineheight` : hauteur de ligne (pour l'espacement entre les lignes) ;
- `element_line()` : pour toutes les lignes tracées. On peut modifier :
  - ▷ `colour` : la couleur de ligne,
  - ▷ `size` : la taille,
  - ▷ `linetype` : le type de ligne (`"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, `"twodash"`),
  - ▷ `lineend` : le type de fin de ligne (`"round"`, `"butt"` ou `"square"`) ;
- `element_rect()` : pour les *backgrounds* et les cadres. On peut modifier :
  - ▷ `fill` : la couleur de remplissage,
  - ▷ `colour` : la couleur de la bordure,
  - ▷ `size` : la taille de la bordure,
  - ▷ `linetype` : le type de ligne (`"blank"`, `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, `"twodash"`) ;
- `element_blank()` permet de ne rien dessiner ;

```
library(grid)
p <- ggplot(data = films_reduit,
            aes(x = estimated_budget/1e6, y = gross_revenue/1e6,
                colour = country_abr)) +
  # Tracer des points
  geom_point() +
  # Ajout d'un titre
  ggtitle("Titre\nsur deux lignes") +
  # Changement des étiquettes pour les axes
  xlab("Étiquette axe des x") + ylab("Étiquette axe des y") +
  # Changement du titre de la légende
  scale_colour_discrete(name = "Country")

# Le graphique par défaut
p
```



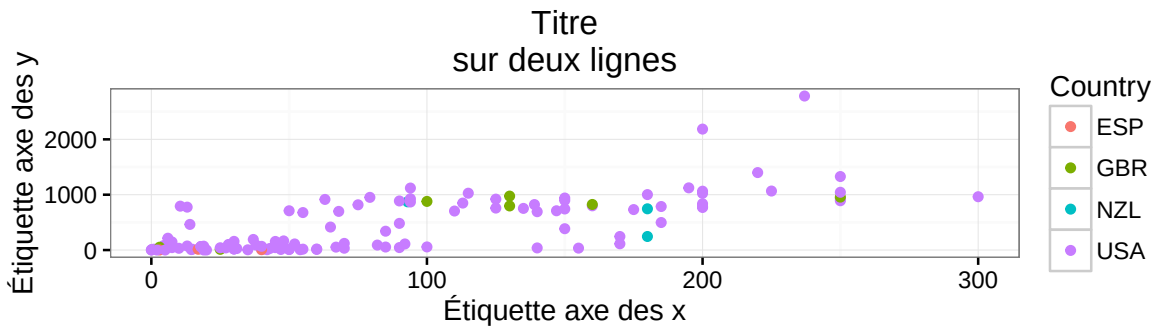

```
# En changeant certains éléments
p + theme(plot.title = element_text(family = "Times", face = "bold",
                        colour = "red",
                        size = rel(2),
                        hjust = 0,
                        lineheight = 1.5),
          axis.title = element_text(face = "bold", colour = "orange"),
          axis.text.x = element_text(colour = "blue", angle = 45),
          axis.ticks = element_line(colour = "brown", size = rel(2)),
          legend.key = element_rect(fill = "dodger blue", colour = "red"),
          legend.background = element_rect(fill = "green",
                                           colour = "purple",
                                           linetype = "twodash"),
          panel.grid.minor = element_blank(),
          panel.grid.major = element_line(colour = "black")
          )
```



Comme on peut le voir, changer les paramètres graphiques peut facilement produire un graphique difficile à lire. Le thème par défaut dans `ggplot2` a été étudié, il convient dans de nombreux cas. Mais on est parfois amené à effectuer quelques changements, d'où l'importance deprésenter la méthode pour réaliser ces changements. Par ailleurs, il existe un autre thèmes

réfléchi dans le *package* `ggplot2` : `theme_bw()` (*background* blanc et grilles noires) [4].

```
p + theme_bw()
```



Certains développent des thèmes supplémentaires. Voir par exemple sur ce *github* : https://github.com/jrnold/ggthemes.

5.3 Enregistrement des graphiques

Pour enregistrer un graphique créé avec la fonction `qplot()` ou `ggplot()`, rien de plus simple : la fonction `ggsave()`. Ses paramètres sont les suivants :

- `filename` : nom du fichier, ou chemin et nom du fichier ;
- `plot` : graphique à sauvegarder (par défaut, le dernier graphique, en faisant appel à la fonction `last_plot()`) ;
- `device` : dispositif à utiliser (automatiquement extrait de l'extension du fichier indiqué au paramètre `filename`) ;
- `path` : chemin vers le fichier ;
- `scale` : facteur d'échelle ;
- `width` : largeur (par défaut, celle de la fenêtre de graphique actuelle) ;
- `height` : hauteur (par défaut, celle de la fenêtre de graphique actuelle) ;
- `units` : unité pour la largeur et la longueur (`"in"`, `"cm"` ou `"mm"`) ;
- `dpi` : nombre de points par pouce, uniquement pour les images matricielles ;
- `limitsize` : quand `TRUE` (la valeur par défaut), l'image sauvegardée ne dépassera pas les 50×50 in.

La fonction `ggsave()` est capable de reconnaître les extensions de fichiers suivants : `eps/ps`, `tex`, `pdf`, `jpeg`, `tiff`, `png`, `bmp`, `svg` et `wmf` (uniquement pour Windows).

```
p <- ggplot(data = films_reduit, aes(x = estimated_budget,
                                     y = gross_revenue,
                                     colour = country)) +
  geom_point() + xlab("Estimated budget") + ylab("Gross Revenue") +
  scale_colour_discrete(name = "Country") +
  ggtitle("A small sample of movies")

p

# Sauvegarde dans le répertoire de travail,
# au format pdf, le dernier graphique affiché créé via ggplot2,
# aux dimensions égales à la fenêtre des graphiques
ggsave("estim_bud.pdf")
```

4. `theme_grey()` (*background* gris et grilles blanches) correspond au thème par défaut

```
# En précisant plus de paramètres
ggsave(p, file = "estim_bud.pdf", width = 15, height = 8,
       unit = "cm", scale = 2)
```

5.4 Cartes

Avec `ggplot2`, il est très facile de dessiner des cartes. Il faut cependant charger quelques autres *packages*. Attention, le but de cette section est de montrer des méthodes pour dessiner des cartes. Le lecteur intéressé par les traitements des données spatiales ne trouvera pas satisfaction avec cette introduction à l'utilisation de `R` et devra plutôt consulter d'autres références (e.g. (Bivand et al., s. d.))

5.4.1 Récupérer des cartes toutes faites

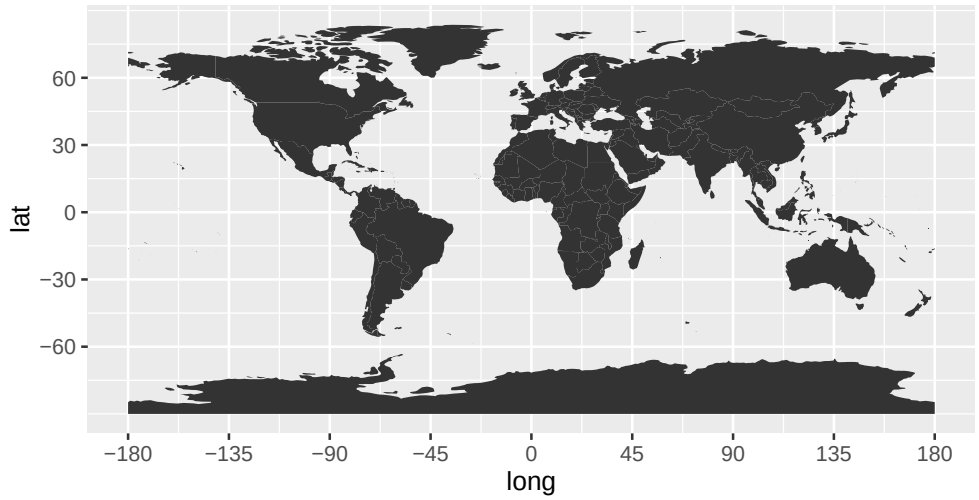
La première chose nécessaire à la création d'une carte, est de disposer des données pour pouvoir tracer les frontières. Dans les cas les plus simples, on peut récupérer des jeux de données présents dans des *packages*.

5.4.1.1 *Package* `rworldmap`

La fonction `getMap()` permet d'accéder à une carte stockée dans le *package*. Ensuite, on applique la fonction `fortify()`, pour transformer le *SpatialPolygonsDataFrame* qui a été retourné par la fonction `getMap()` en `data frame`, afin d'avoir un format lisible par la fonction `ggplot()`.

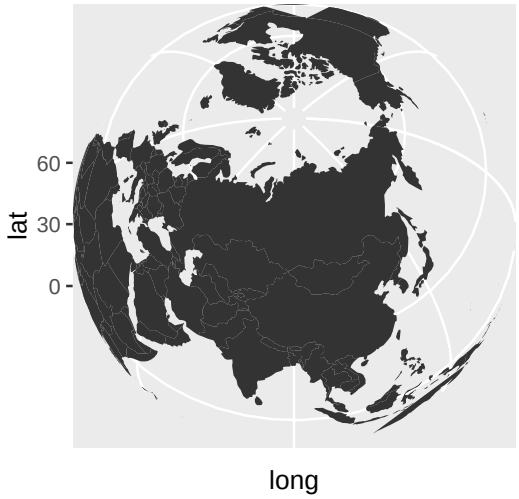
```
library(ggplot2)
library(rworldmap)

## Loading required package: sp
## ### Welcome to rworldmap ###
## For a short introduction type : vignette('rworldmap')

# Carte du monde
worldMap <- getMap()
# Format lisible pour ggplot()
world_df <- fortify(worldMap)

## Regions defined for each Polygons

head(world_df)

##       long      lat order  hole piece          id         group
## 1 61.21082 35.65007     1 FALSE     1 Afghanistan Afghanistan.1
## 2 62.23065 35.27066     2 FALSE     1 Afghanistan Afghanistan.1
## 3 62.98466 35.40404     3 FALSE     1 Afghanistan Afghanistan.1
## 4 63.19354 35.85717     4 FALSE     1 Afghanistan Afghanistan.1
## 5 63.98290 36.00796     5 FALSE     1 Afghanistan Afghanistan.1
## 6 64.54648 36.31207     6 FALSE     1 Afghanistan Afghanistan.1
```

```
worldmap <- ggplot() +
  geom_polygon(data = world_df, aes(x = long, y = lat, group = group)) +
  scale_y_continuous(breaks = (-2:2) * 30) +
  scale_x_continuous(breaks = (-4:4) * 45) +
  coord_equal()

worldmap
```



On peut tout à fait s'amuser avec la fonction `coord_map()`, pour changer le système de coordonnées.

```
worldmap <- ggplot() +
  geom_polygon(data = world_df, aes(x = long, y = lat, group = group)) +
  scale_y_continuous(breaks = (-2:2) * 30) +
  scale_x_continuous(breaks = (-4:4) * 45) +
  coord_map("ortho", orientation=c(61, 90, 0))
worldmap
```



5.4.1.2 *Package* `maps`

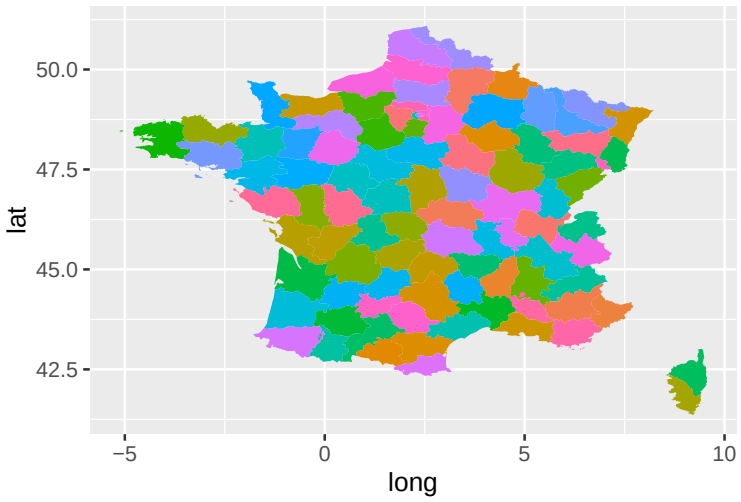
Le *package* `rworldmap` est pratique pour obtenir les frontières des pays, mais si on désire obtenir les *counties* américains, ou les départements français, il est préférable d'aller voir du côté du

package `maps`. La fonction `map_data()` du *package* `ggplot2` s'appuie sur les cartes du *package* `maps` pour extraire un *data frame* prêt à être employé avec la fonction `ggplot()`. Il faut lui fournir le nom de la carte voulu, parmi les noms suivants :

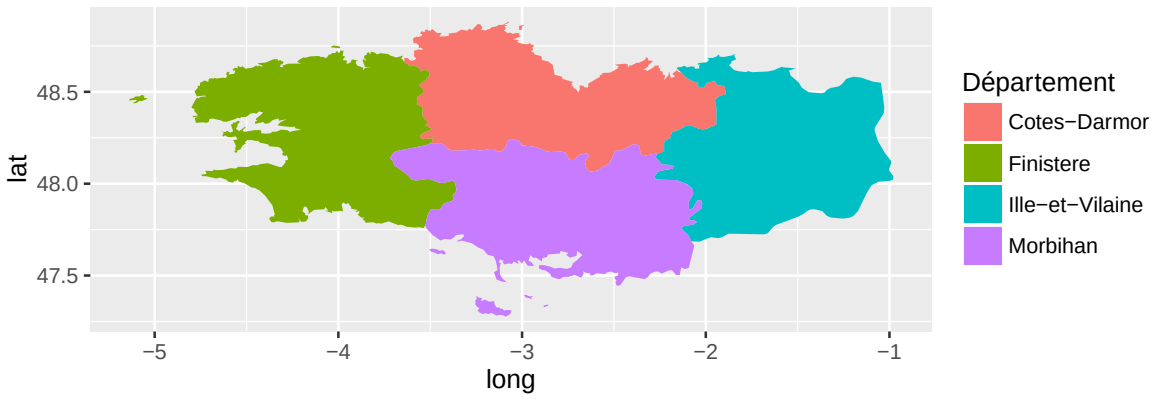
- `county` : carte des *counties* américains ;
- `france` : carte de la France ;
- `italy` : carte de l'Italie ;
- `nz` : carte de la Nouvelle-Zélande ;
- `state` : carte des États-Unis avec chaque état ;
- `usa` : carte des États-Unis avec uniquement les frontières ;
- `world` : carte du monde ;
- `world2` : carte du monde centrée sur le Pacifique.

Si on désire extraire uniquement certains sous-régions, il faut préciser leur nom au paramètre `region`.

```
map_fr <- map_data("france")

##
##  # ATTENTION: maps v3.0 has an updated 'world' map.        #
##  # Many country borders and names have changed since 1990. #
##  # Type '?world' or 'news(package="maps")'. See README_v3. #
##
##
##
## Attaching package: 'maps'
##
## The following object is masked from 'package:plyr':
##
##    ozone

# Le nom des régions
head(unique(map_fr$region))

## [1] "Nord"           "Pas-de-Calais"  "Somme"          "Ardennes"
## [5] "Seine-Maritime" "Aisne"

# Carte de la France
p_map_fr <- ggplot(data = map_fr,
                   aes(x = long, y = lat, group = group, fill = region)) +
  geom_polygon() + coord_equal() + scale_fill_discrete(guide = "none")

p_map_fr
```




```r
# Extrayons les données uniquement pour la Bretagne
ind_bzh <- grep("armor|finis|vilaine|morb",
                unique(map_fr$region), ignore.case = TRUE)

# Voici les noms des départements de la Bretagne, tels qu'ils sont
# stockés dans le package maps
(dep_bzh <- unique(map_fr$region)[ind_bzh])

## [1] "Cotes-Darmor"    "Finistere"       "Ille-et-Vilaine" "Morbihan"

map_fr_bzh <- map_data("france", region = dep_bzh)

# Carte de la Bretagne
p_map_fr_bzh <- ggplot(data = map_fr_bzh,
                       aes(x = long, y = lat, group = group, fill = region)) +
  geom_polygon() + coord_equal() + scale_fill_discrete(name = "Département")

p_map_fr_bzh
```



5.4.1.3 Fichier `shapefile`

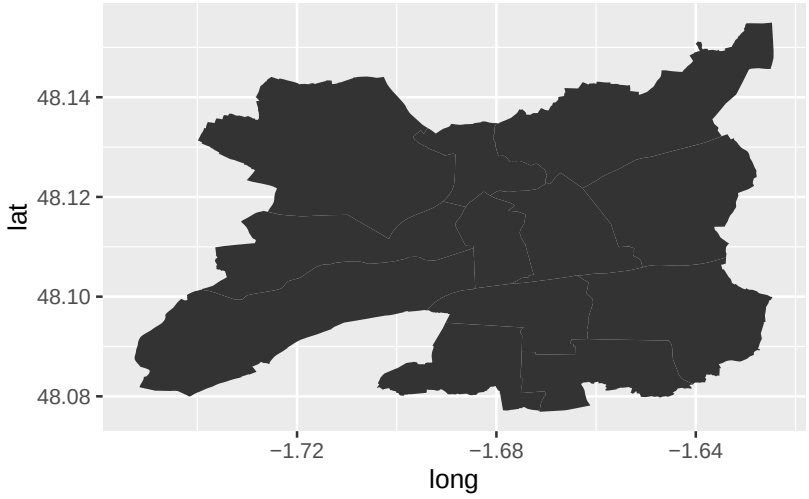
Il est également possible d'importer un fichier `shp` et de tracer une carte basée sur son contenu. Il est cependant nécessaire de charger quelques *packages* supplémentaires. Par exemple, pour tracer les quartiers de la ville de Rennes, on peut télécharger et extraire le contenu du fichier `shapefile` zippé disponible sur le site : http://www.data.rennes-metropole.fr. L'importation dans `R` se fait comme suit, en ayant placé le répertoire `"quartiers_shp_lamb93"` que l'on vient d'extraire dans le dossier pointé par la fonction `getwd()` [5].

```
library("rgdal")
library("maptools")
library("ggplot2")
library("plyr")

# Importer les polygones
rennes <- readOGR(dsn="./quartiers_shp_lamb93", layer="quartiers")

# Étape pour changer la projection de la carte
rennes <- spTransform(rennes, CRS("+proj=longlat +ellps=GRS80"))

# Pour permettre la jointure des objets géométriques
rennes@data$id <- rownames(rennes@data)

# Transformer en data frame pour fournir à ggplot()
rennes_points <- fortify(rennes, region="id")

# Permet d'éviter des trous éventuels
rennes_df <- join(rennes_points, rennes@data, by="id")
```

Il ne reste plus qu'à tracer la carte.

```
p_map_rennes <- ggplot(data = rennes_df,
                       aes(x = long, y = lat, group = group)) +
  geom_polygon() +
  coord_equal()

p_map_rennes
```

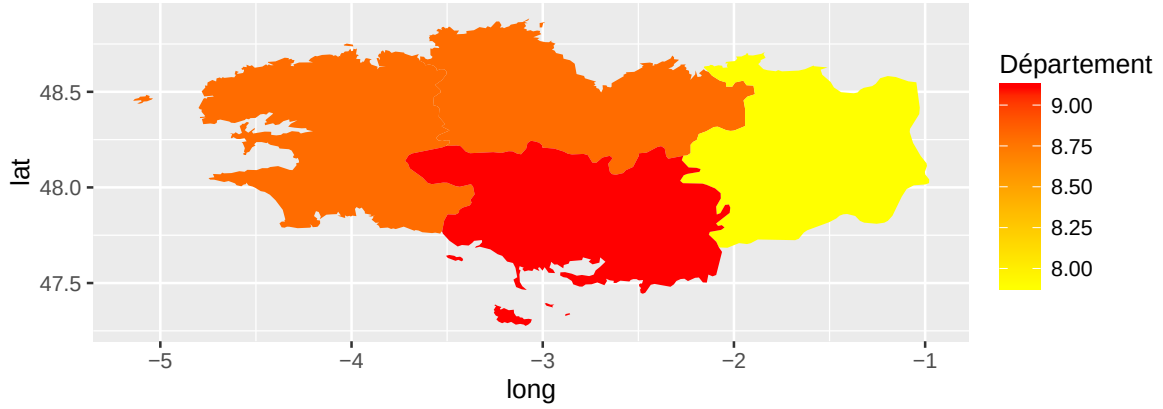
5. pour plus de renseignements, voir ce *github* : https://github.com/hadley/ggplot2/wiki/plotting-polygon-shapefiles



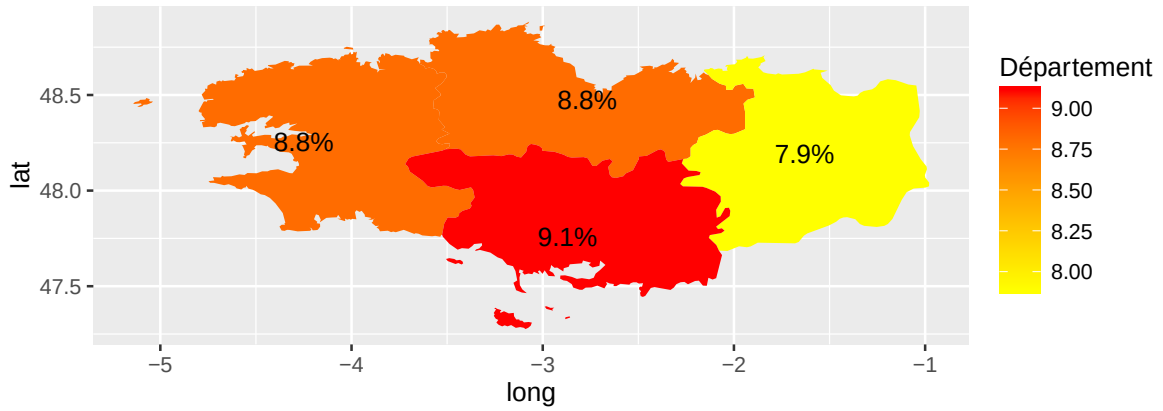
5.4.2 Carte choroplèthe

Pour réaliser une carte choroplète, c'est-à-dire une carte où les régions sont remplies par une couleur en fonction d'une statistique, il suffit juste d'ajouter une colonne avec la valeur de la statistique dans le *data frame* permettant de dessiner la carte. Appuyons-nous sur la carte de la Bretagne définie en section 5.4.1.2.

```
tx_chomage_2014_T1 <- data.frame(
  region = c("Cotes-Darmor","Finistere",
             "Ille-et-Vilaine", "Morbihan"),
  tx_chomage_2014_T1 = c(8.8, 8.8,7.9, 9.1))

# Ajout des valeurs pour chaque région
ind_match <- match(map_fr_bzh$region, tx_chomage_2014_T1$region)
map_fr_bzh$tx_chomage_2014_T1 <- tx_chomage_2014_T1[ind_match,
                                                    "tx_chomage_2014_T1"]

# Il suffit de faire dépendre le remplissage de tx_chomage_2014_T1
p_map_fr_bzh <- ggplot(data = map_fr_bzh,
                   aes(x = long, y = lat, group = group,
                       fill = tx_chomage_2014_T1)) +
  geom_polygon() + coord_equal() +
  scale_fill_gradient(name = "Département", low ="#FFFF00", high = "#FF0000")

p_map_fr_bzh
```



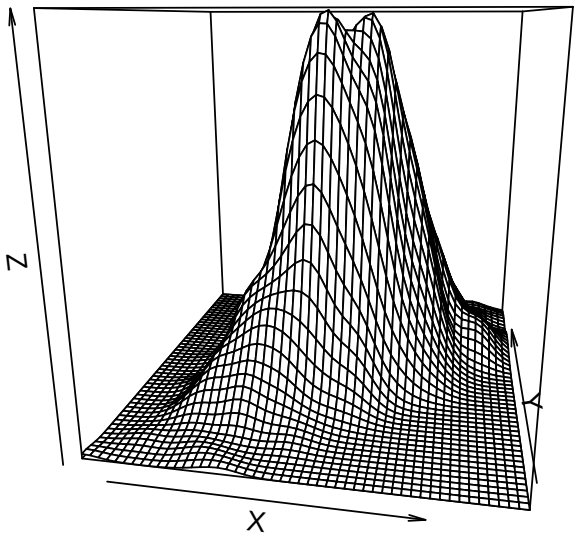

```
# Fonction pour trouver le point central du polygone
mid_range <- function(x) mean(range(x, na.rm = TRUE))
centres <- ddply(map_fr_bzh, .(region), colwise(mid_range, .(lat, long)))

# Rajout des taux de chômage
ind_match <- match(centres$region, tx_chomage_2014_T1$region)
centres$tx_chomage_2014_T1 <- tx_chomage_2014_T1$tx_chomage_2014_T1[ind_match]

label_chomage <- paste0(centres$tx_chomage_2014_T1, "%")

p_map_fr_bzh + annotate("text", x = centres$long,
                        y = centres$lat, label = label_chomage)
```



5.5 Graphiques en 3D

Le *package* `ggplot2` ne permet malheureusement pas (encore) de réaliser des graphiques en 3D. Il faut aller regarder du côté des *packages* `base`, `lattice` ou encore `rgl` pour cela. Le lecteur intéressé par ces options offertes par `R` est invité à consulter les pages d'aide des fonctions, à parcourir les vignettes et les démos.

```
library(MASS)

##
## Attaching package: 'MASS'
##
## The following object is masked from 'package:dplyr':
##
##     select

set.seed(1)
# Normale bivariée
Sigma <- matrix(c(10,3,3,2),2,2)
biv_n <- mvrnorm(n=1000, rep(0, 2), Sigma)

# Estimation par la méthode du noyau de la densité
biv_n_kde <- kde2d(biv_n[,1], biv_n[,2], n = 50)

persp(biv_n_kde, theta = 10, phi = 15, xlab = "X")
```



Une autre fonction, peut-être plus pratique pour visualiser des graphiques en 3d, puisqu'on peut se servir de la souris pour changer d'angle, est `plot3d()`, du *package* `rgl`.

```
library(rgl)
set.seed(1)
n <- 10000
x <- rnorm(n, mean = 38)
y <- rnorm(n, mean = 42)
```

```
biv_kde <- kde2d(x, y, n = 50)
den_z <- biv_kde$z

surface3d(biv_kde$x,biv_kde$y,den_z*20,color="#FF2222",alpha=0.5)
```

Pour les personnes intéressées, l'exécution des lignes ci-après donne un aperçu plus large de ce qu'il est possible de faire avec le *package* `rgl`.

```
demo(rgl)
example(rgl)
```

5.6 Exercices

Exercice 5.1 : créer un graphique simple, modifier son aspect

5.1.1 Charger le *package* `ggplot2`, et utiliser la fonction `data()` pour charger en mémoire le jeu de données `economics`. Consulter la page d'aide de ce jeu de données pour prendre connaissance de son contenu ;

5.1.2 À l'aide de la fonction `ggplot()`, représenter les dépenses personnelles de consommation (`pce`) en fonction de la date (`date`). Les observations doivent être connectées par une ligne.

5.1.3 Modifier le graphique de la question précédente de manière à ce que la couleur de la ligne soit `dodger blue` et définir la taille de la ligne à `0.5`. Stocker le résultat dans un objet que l'on appellera `p_1` ;

5.1.4 Ajouter une couche au graphique `p_1` pour modifier les titres des axes (les retirer), et définir le titre suivant : *"Personal Consumption Expenditures (billions dollars)"*.

5.1.5 Utiliser la fonction `date_breaks()` du *package* `scales` pour modifier l'échelle des abscisses de `p_1`, afin que les étiquettes des marques soient affichées tous les 5 ans ; à l'aide de la fonction `date_format()`, modifier le format de ces étiquettes pour que seule l'année des dates s'affiche ;

Exercice 5.2 : créer un graphique avec plusieurs courbes, modifier son aspect

5.2.1 Charger le *package* `ggplot2`, et utiliser la fonction `data()` pour charger en mémoire le jeu de données `economics`. Consulter la page d'aide de ce jeu de données pour prendre connaissance de son contenu ;

5.2.2 Charger les *packages* `dplyr` et `tidyr`. Sélectionner les variables `date`, `psavert` et `uempmed` dans le tableau de données `economics` et utiliser la fonction `gather()` sur le résultat pour obtenir un tableau dans lequel chaque ligne indiquera la valeur (`value`) pour une variable donnée (`key`) à une date donnée (`date`). Stocker le résultat dans un objet que l'on appellera `df` ;

5.2.3 Sur un même graphique, représenter à l'aide de lignes, l'évolution dans le temps du taux d'épargne personnelle (`psavert`) et de la durée médiane en semaines du chômage (`uempmed`). Stocker le graphique dans un objet que l'on appellera `p_2` ;

5.2.4 Modifier le code ayant servi à construire le graphique `p_2` pour que le type de ligne soit différent pour chacune des deux séries représentées. Les deux lignes doivent être tracées en bleu. Stocker le graphique dans un objet que l'on appellera `p_3` ;

5.2.5 À présent, modifier le code ayant servi à construire `p_3` pour qu'à la fois la couleur et le type de ligne servent à différencier les deux séries. Stocker le graphique dans un objet que l'on appellera `p_4` ;

5.2.6 Modifier le graphique `p_4` en ajoutant une couche d'échelle de couleur pour que le taux d'épargne personnelle (`psavert`) soit représenté en `dodger blue`, et que la durée de chômage (`uempmed`) soit représentée en rouge. Par ailleurs, retirer le titre de la légende ;

5.2.7 Modifier le graphique `p_4` en ajoutant une couche d'échelle de type de ligne pour que le taux d'épargne personnelle (`psavert`) soit représenté par des tirets, et que la durée de chômage (`uempmed`) soit représentée par une ligne pleine. Par ailleurs, retirer le titre de la légende des types de lignes, afin que les légendes de couleur et de type de ligne soient fusionnées ;

5.2.8 Créer le tableaux de données `df_2`, une copie de `df`, dans lequel la variable `key` doit être un facteur dont les niveaux sont `uempmed` et `psavert` ;

5.2.9 Créer le vecteur `etiq` suivant :

```
etiq <- c("psavert" = "Pers. Saving Rate",
          "uempmed" = "Median Duration of Unemployment (weeks)")
```

Ce vecteur contient des valeurs d'étiquettes pour la légende du graphique qu'il va falloir créer.

Représenter sur un même graphique lévolution dans le temps du taux d'épargne personnelle et de la durée médiane du chômage en semaines, en s'appuyant sur les données contenues dans le tableau `df_2`. La courbe du taux d'épargne personnelle doit être composée de tirets et être de couleur `dodger blue` ; la courbe de la durée médiane du taux de chômage doit être une ligne rouge. La légende ne doit pas comporter de titre, et ses étiquettes doivent être modifiées pour que *"Pers. Saving Rate"* s'affiche 'a la place de *"psavert"*, et pour que *"Median Duration of Unemployment (weeks)"* s'affiche à la place de *"uempmed"*. Stocker le graphique dans un objet que l'on appellera `p_5` ;

Note : il s'agit de reprendre le code ayant servi à créer le graphique `p_4`, en modifiant légèrement les échelles de couleur et de ligne pour prendre en compte les étiquettes proposées dans le vecteur `etiq`.

5.2.10 Modifier `p_5` pour lui ajouter une couche permettant de déplacer la légende en bas du graphique (utiliser la fonction `theme()`) ;

5.2.11 Ajouter une couche au graphique `p_5` qui permet de définir un thème. Utiliser le thème minimal (`theme_minimal()`). Que se passe-t-il pour la légende ? Repositionner la légende en dessous, et retirer les titres des axes ;

5.2.12 Sauvegarder le graphique `p_5` au format PDF en précisant une largeur de 12 et une hauteur de 8.

Exercice 5.3 : différentes représentations graphiques

5.3.1 Charger le jeu de données `mpg` contenu dans le *package* `ggplot2` en mémoire, puis consulter la page d'aide du jeu de données pour en prendre connaissance ;

5.3.2 Représenter à l'aide d'un nuage de points la relation entre la consommation sur autoroute des véhicules de l'échantillon (`hwy`) et la cylindrée de leur moteur (`displ`)

5.3.3 Reprendre le code du graphique précédent et modifier la forme des points pour les changer en symbole `+` ; modifier la couleur des `+` de manière à la faire dépendre du nombre de cylindres (`cyl`) ;

5.3.4 À présent, représenter par des boîtes à moustaches la distribution de la consommation sur autoroute des véhicules (`hwy`) pour chacune des valeurs possibles du nombre de cylindres (`cyl`) ;

5.3.5 Charger le jeu de données `economics` contenu dans le *package* `ggplot2` en mémoire, puis consulter la page d'aide du jeu de données pour en prendre connaissance. Ensuite, ajouter au tableau (les créer) les variables `u_rate` et `e_rate`, respectivement le taux de chômage et le taux d'emploi (on définira le taux de chômage de manière très grossière ici : nombre de personnes non employées sur la population totale) ;

5.3.6 Représenter à l'aide de barres l'évolution dans le temps du taux de chômage, et remplir les barres avec la couleur rouge ;

5.3.7 Reprendre le code du graphique précédent et ajouter une couche permettant de modifier les limites de l'axe des abscisses pour afficher les valeurs uniquement sur la période "2012-01-01" à "2015-01-01" (utiliser la fonction `coord_cartesian()`). Stocker le graphique dans un objet que l'on appellera `p` ;

5.3.8 Dans le tableau de données `economics`, sélectionner les variables `date`, `u_rate` et `e_rate`, puis utiliser la fonction `gather()` pour obtenir un tableau dans lequel chaque ligne correspond à la valeur d'une des variables (taux de chômage ou taux d'emploi) à une date donnée. Stocker le résultat dans un objet que l'on appellera `df_3` ;

5.3.9 Utiliser le tableau de données `df_3` pour représenter graphiquement à l'aide de barres les taux de chômage et taux d'emploi par mois sur la période "2012-01-01" à "2015-01-01". Sur le graphique, les barres représentant le taux de chômage et celles représentant le taux d'emploi devront être superposées.

Note : il s'agit de modifier légèrement le code ayant permis de réaliser le graphique `p`*.*

Exercice 5.4 : facettes

5.4.1 Charger le *package* `WDI` (l'installer si nécessaire), puis en utilisant la fonction `WDI()`, récupérer les données de PIB par tête (`NY.GDP.PCAP.PP.KD`, PPP, constant 2005 international $) et de taux de chômage (`SL.UEM.TOTL.ZS`, total, % of total labor force) pour la France, l'Allemagne et le Royaume Uni, pour la période allant de 1990 à 2015. Ces données doivent être stockées dans un tableau que l'on appellera `df` ;

5.4.2 Transformer le tableau `df` afin que chaque ligne indique : l'état (`country`), l'année (`year`), le nom de la variable (`variable`) et la valeur (`valeur`) (utiliser la fonction `gather()`). Puis, modifier la colonne `variable` afin qu'elle soit de type `factor`, et que les étiquettes des niveaux `NY.GDP.PCAP.PP.KD` et `SL.UEM.TOTL.ZS` deviennent `GDP` et `Unemployment` respectivement ;

5.4.3 Représenter graphiquement l'évolution du PIB et du taux de chômage pour les trois pays. Utiliser la fonction `facet_wrap()` afin de regrouper les variables par type : les observations des valeurs du PIB d'un côté du "tableau" de graphiques, et celles du taux de chômage de l'autre. Utiliser une représentation en ligne, en faisant dépendre la couleur du pays ;

5.4.4 Reprendre le code du graphique précédent en le modifiant légèrement afin de libérer les axes des ordonnées ;

5.4.5 Modifier les paramètres esthétiques du graphique afin de faire dépendre le type de ligne des pays de la manière suivante : des points pour la France, des tirets pour l'Allemagne, des tirets longs pour le Royaume Uni. Définir l'épaisseur des lignes à 1.5 ;

5.4.6 Modifier légèrement le code ayant permis de réaliser le graphique de la question précédente pour que la direction ne soit non plus horizontale (par défaut), mais verticale (paramètre `dir`, ou à défaut, `ncol` dans ce cas précis) ;

5.4.7 En utilisant la fonction `facet_wrap()`, créer une grille de graphiques, de sorte que chaque pannel représente l'évolution d'une seule série pour un pays donné ;

5.4.8 À présent, utiliser la fonction `facet_grid()` pour créer une grille de graphiques dans laquelle les lignes correspondent aux pays et les colonnes aux variables. Prendre soin de libérer les échelles ;

5.4.9 Reprendre la question précédente en faisant cette fois une girlle dans laquelle les lignes correspondent aux variables et les colonnes aux pays ;

Exercice 5.5 : annotations

5.5.1 En utilisant la fonction `WDI()` du *package* `WDI`, récupérer les séries de 2010 du PIB par tête (`NY.GDP.PCAP.PP.KD`, PPP, constant 2005 international $) et de l'espérance de vie à la naissance (`SP.DYN.LE00.IN`, total, years) pour tous les pays. Les données seront stockées dans un tableau que l'on nommera `df` ;

5.5.2 Représenter par un nuage de points l'espérance de vie à la naissance en fonction du PIB par tête. Retirer les titres des axes, et ajouter le titre suivant (sur deux lignes) :
"Life Expectancy at birth (years) VS.
Real GDP per Capita (PPP, contant 2005 international $)" ;

5.5.3 Ajouter une courbe de tendance obtenue par lissage Loess (en utilisant une fonction du *package* `ggplot2`) ;

5.5.4 Modifier le tableau `df` pour lui ajouter la variable `drapeau` qui prendra la valeur `TRUE` si l'état de l'observation est soit `France`, soit `Luxembourg`, et `FAUX` sinon ;

5.5.5 Modifier légèrement le code ayant permis de réaliser le précédent graphique, pour faire dépendre la couleur des points de la variable drapeau (rouge pour `TRUE` et noir pour `FALSE`) ;

5.5.6 Créer un tableau de données que l'on appellera `df_fleche` qui contient les observations du tableau `df` pour la France et le Luxembourg uniquement ;

5.5.7 Reprendre le code du graphique précédent pour le modifier de façon à ajouter deux flèches : l'une montrant le points de la France et l'autre du Luxembourg. Le nom du pays devra être inscrit à l'origine des deux flèches (utiliser la fonction `annotate()` et se servir du tableau de données `df_fleche`) ;

Exercice 5.6 : annotations

5.6.1 À l'aide de la fonction `WDI` du *package* `WDI`, récupérer la série de l'inflation (`FP.CPI.TOTL.ZG`, consumer prices, annual %) en France sur la période 1960–2015, et stocker ces données dans un tableau que l'on appellera `inflation_fr` ;

5.6.2 Représenter par une ligne l'évolution du taux annuel d'inflation en France sur la période 1960–2015. Retirer les titres d'axes et ajouter le titre suivant : *"Inflation in France (annual %)"* ;

5.6.3 Soient les deux vecteurs suivants, qui définissent les dates de début et fin de la période inflationniste et celle de grande modération :

```
per_1 = c(1970, 1976) # Periode inflationniste
per_2 = c(1976, 1991) # Periode de grande moderation
```

Reprendre le code du graphique précédent et ajouter deux rectangles : un premier, rouge, pour mettre en valeur la période inflationniste et un second, bleu, mettant en valeur la période de grande modération. Modifier le paramètre de transparence pour le fixer à 0.2 ;

5.6.4 Ajouter au graphique précédent des lignes grises verticales en tirets pour les années 1973, 1979 et 2008. Modifier également les *breaks* de l'échelle des abscisses à l'aide de la fonction `pretty_breaks()` du *package* scales ;

Exercice 5.7 : carte simple

5.7.1 À l'aide de la fonction `WDI` du *package* `WDI`, récupérer la série fournie par la Banque Mondiale du PIB par tête (`NY.GDP.PCAP.PP.KD`, PPP, constant 2005 international $) pour tous les pays disponibles pour l'année 2010, et stocker ces données dans un tableau que l'on appellera `gdp_capita` ;

5.7.2 Dans le tableau `gdp_capita`, modifier la valeur de la variable `country` pour l'observation de la Slovaquie, pour qu'elle vaille `Slovakia` au lieu de `Slovak Republic` ;

5.7.3 Filtrer les observations du tableau `gdp_capita` pour ne conserver que les observations des pays membres de l'Union Européenne dont les noms sont contenus dans le vecteur `membres_ue`. Stocker le résultat dans un tableau que l'on nommera `gdp_capita_eu` ;

```
membres_ue <-c("Austria", "Belgium", "Bulgaria", "Cyprus", "Croatia",
               "Czech Republic", "Denmark", "Estonia", "Finland", "France",
               "Germany", "Greece", "Hungary", "Ireland", "Italy", "Latvia",
               "Lithuania", "Luxembourg", "Malta", "Netherlands", "Poland",
               "Portugal", "Romania", "Slovakia", "Slovenia", "Spain",
               "Sweden", "United Kingdom")
```

5.7.4 Utiliser le *package* `rworldmap` pour récupérer les données nécessaires à la réalisation d'une carte du monde ;

5.7.5 Afficher une carte du monde à l'aide des fonctions contenues dans le *package* `ggplot2` ;

5.7.6 Modifier les échelles des axes pour faire figurer les méridiens de -60 à 60 par pas de 30 et les parallèles de -180 à 180 par pas de 45. Modifier également la projection cartographique pour choisir la projection orthographique, à l'aide de la fonction `coord_map()` ;

5.7.7 Joindre les informations contenues dans le tableau `gdp_capita_eu` au tableau contenant les données permettant la réalisation des cartes ;

5.7.8 Réaliser une carte choroplèthe reflétant pour chaque pays membre de l'Union Européenne la valeur du PIB par tête de 2012 ;

5.7.9 Modifier les couleurs de l'échelle continue de la carte précédente, pour que les faibles valeurs du PIB par tête soient représentées en jaune, et les valeurs les plus hautes en rouge ;

5.7.10 Modifier les ruptures de l'échelle de couleur pour qu'elles aillent de 10000 à 100000 ; modifier également l'étiquette de ces ruptures de sorte que 35000 soit affiché comme 35k, 60000 comme 60k, etc. Enfin, ajouter un titre au graphique et retirer les titres d'axes.

6

Régressions linéaires avec R

6.1 Rappels

On souhaite étudier la liaison entre une variable y et une ou plusieurs variables $x_1, x_2, \ldots, x_m$. La variable y est appelée *variable à expliquer*, ou *réponse*, et les variables x_j, $j = 1, 2, \ldots, m$ sont appelées variables explicatives. On suppose que la relation entre la variable à expliquer et les variables explicatives est de la forme $y = f(x_1, x_2, \ldots, x_m)$, avec m le nombre de variables explicatives. On émet l'hypothèse que la réponse est linéairement indépendante des variables x_j, avec $j = 1, \ldots, m$.
Il s'agit d'estimer les coefficients β_j de l'équation à m variables explicatives x_j, avec $j = 1, 2, \cdots, m$, β_0 étant la constante, et ε un terme d'erreur supposé normal :

$$\boldsymbol{y} = \beta_0 + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \cdots + \beta_j \boldsymbol{x}_j + \cdots + \beta_m \boldsymbol{x}_m + \boldsymbol{\varepsilon}. \tag{6.1.1}$$

Soit, en termes matriciels

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}, \tag{6.1.2}$$

$$\text{où } \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}, \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} \text{ et } \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_m \end{bmatrix}.$$

Les coefficients β_j sont inconnus et estimés par $\hat{\beta}_j$ tels que :

$$\begin{cases} \hat{y_1} & = \hat{\beta}_0 + \hat{\beta}_1 x_{11} + \hat{\beta}_2 x_{12} + \cdots + \hat{\beta}_j x_{1j} + \hat{\beta}_m x_{1m} \\ \hat{y_2} & = \hat{\beta}_0 + \hat{\beta}_1 x_{21} + \hat{\beta}_2 x_{22} + \cdots + \hat{\beta}_j x_{2j} + \hat{\beta}_m x_{2m} \\ \vdots & = \vdots \\ \hat{y_n} & = \hat{\beta}_0 + \hat{\beta}_1 x_{n1} + \hat{\beta}_2 x_{n2} + \cdots + \hat{\beta}_j x_{nj} + \hat{\beta}_m x_{nm} \end{cases}.$$

En termes matriciels, cela donne :

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{\beta}}, \tag{6.1.3}$$

$$\text{où } \hat{\boldsymbol{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix}, \boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}, \text{ et } \hat{\boldsymbol{\beta}} = \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_m \end{bmatrix}.$$

Avec la méthode des moindres carrés, l'objectif est de trouver $\hat{\boldsymbol{\beta}}$ tels que la somme des carrés des résidus soit minimale. La somme des carrés des résidus est définie par :

$$|| \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta} ||^2 = \sum_{i=1}^{n} (y_i - x_i\beta)^2.$$

La condition du premier ordre donne [1] :

$$\begin{aligned} & \boldsymbol{X}^t\boldsymbol{X}\hat{\boldsymbol{\beta}} - 2\boldsymbol{X}^t\boldsymbol{X}\hat{\boldsymbol{\beta}} - 2\boldsymbol{X}^t\boldsymbol{y} = 0 \\ \Leftrightarrow \quad & \boldsymbol{X}^t\boldsymbol{X}\hat{\boldsymbol{\beta}} = \boldsymbol{X}^t\boldsymbol{y} \\ \Leftrightarrow \quad & \hat{\boldsymbol{\beta}} = (\boldsymbol{X}^t\boldsymbol{X})^{-1}\boldsymbol{X}^t\boldsymbol{y}. \end{aligned} \tag{6.1.4}$$

6.2 Données de l'exemple

Nous allons nous appuyer sur des données de naissances à Philadelphie, en 1990 (Elo et al., 2001). L'échantillon concerne 5% des naissances ayant eu lieu dans cette ville en 1990, ce qui représente 1115 observations. Chaque enregistrement renseigne sur :

- `grams` : masse à la naissance (grammes) ;
- `gestate` : temps de gestation (semaines) ;
- `educ` : nombre d'années d'éducation de la mère (0–17) ;
- `black` : variable indicatrice de la couleur de peau de la mère (1 si oui, 0 sinon) ;
- `smoke` : indique si la mère a fumé pendant la grossesse (1 si oui, 0 sinon).

Attention, ce qui est proposé dans cette section ne constitue pas un fil d'Ariane pour réaliser une analyse sur des données. L'idée est de montrer au lecteur les fonctions principales, pour que ce premier puisse réaliser par lui-même ses analyses. D'excellentes références proposent d'aller plus loin au sujet de la régression (Matzner-Løber, 2007; Lafaye de Micheaux et al., 2011; Dalgaard, 2008).

Une très rapide visualisation des données est l'occasion d'introduire ici la fonction `grid.arrange()` du *package* `gridExtra`, qui permet de positionner des graphiques réalisés avec `ggplot2` sur une grille.

```
url <- "http://data.princeton.edu/wws509/datasets/phbirths.dat"
births <- read.table(url, header = TRUE)
head(births)

##   black educ smoke gestate grams
## 1 FALSE    0  TRUE      40  2898
## 2  TRUE    0  TRUE      26   994
## 3 FALSE    2 FALSE      38  3977
## 4 FALSE    2  TRUE      37  3040
## 5 FALSE    2 FALSE      38  3523
## 6 FALSE    5  TRUE      40  3100
```

1. On utilise les propriétés suivantes : $\frac{\partial \boldsymbol{x}^t\boldsymbol{A}}{\partial \boldsymbol{x}} = \boldsymbol{A}^t$, $\frac{\partial \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{A}$ et $\frac{\partial a\boldsymbol{\varepsilon}}{\partial \boldsymbol{x}} = a\frac{\partial u}{\partial \boldsymbol{x}}$, avec $u = u(\boldsymbol{x})$.

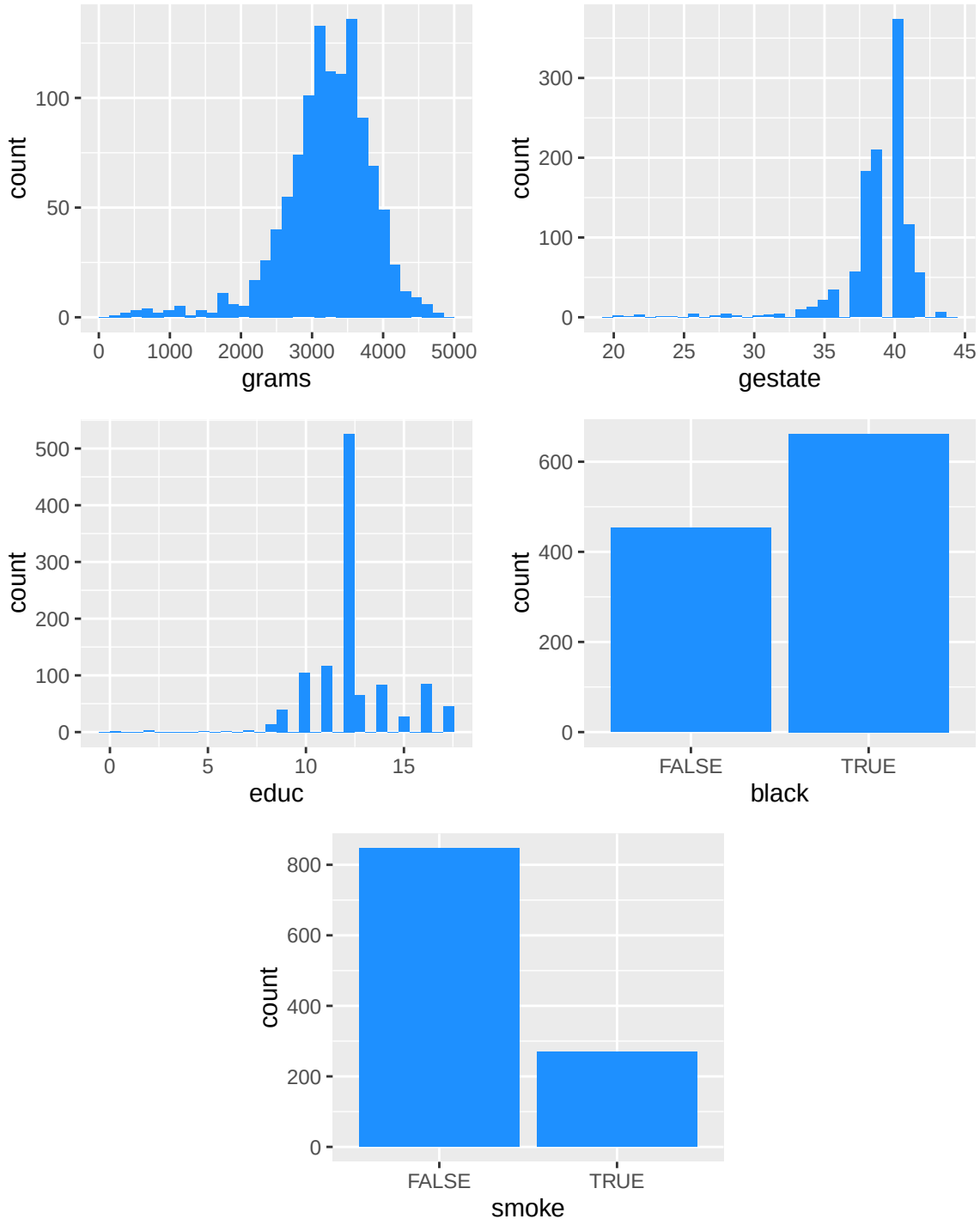
```r
# Un aperçu des données
summary(births)
```

```
##    black              educ          smoke            gestate
##  Mode :logical   Min.   : 0.00   Mode :logical   Min.   :20.00
##  FALSE:453       1st Qu.:11.00   FALSE:846       1st Qu.:38.00
##  TRUE :662       Median :12.00   TRUE :269       Median :39.00
##  NA's :0         Mean   :12.27   NA's :0         Mean   :38.84
##                  3rd Qu.:13.00                   3rd Qu.:40.00
##                  Max.   :17.00                   Max.   :43.00
##      grams
##  Min.   : 284
##  1st Qu.:2900
##  Median :3267
##  Mean   :3220
##  3rd Qu.:3630
##  Max.   :4830
```

```r
# Les corrélations
round(cor(births), 2)
```

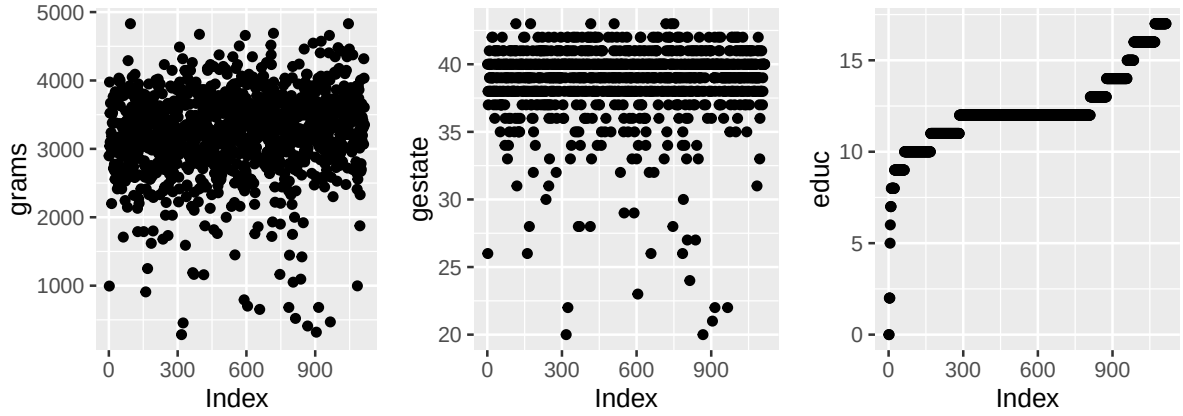
```
##         black  educ smoke gestate grams
## black    1.00 -0.15  0.05   -0.17 -0.26
## educ    -0.15  1.00 -0.23    0.06  0.12
## smoke    0.05 -0.23  1.00   -0.15 -0.23
## gestate -0.17  0.06 -0.15    1.00  0.70
## grams   -0.26  0.12 -0.23    0.70  1.00
```

```r
# Un aperçu graphique des données
library(ggplot2)
qplot(data = births, grams, fill = I("dodger blue"))
qplot(data = births, gestate, fill = I("dodger blue"))
qplot(data = births, educ, fill = I("dodger blue"))
qplot(data = births, black, fill = I("dodger blue"))
qplot(data = births, smoke, fill = I("dodger blue"))
```



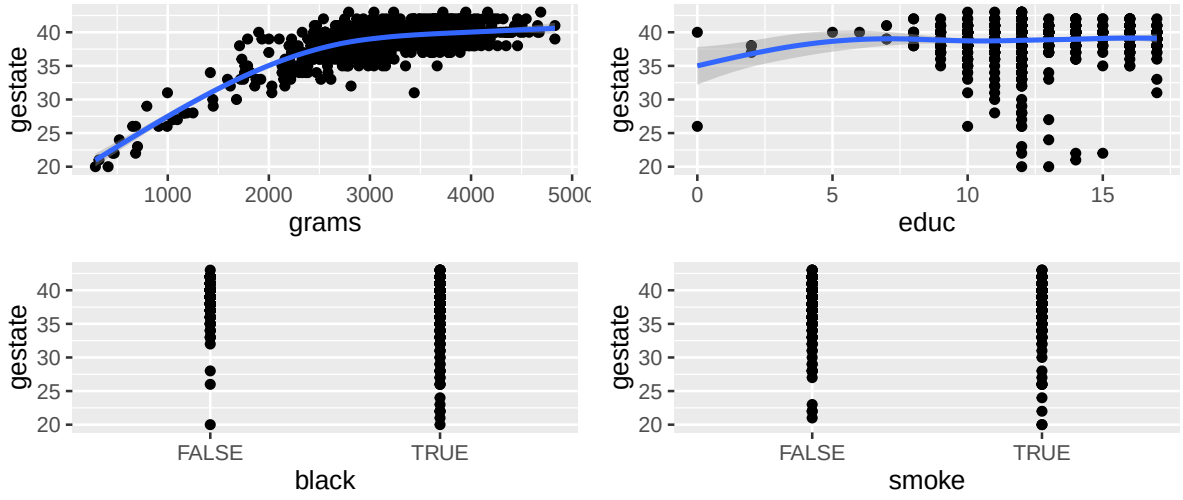
Si on souhaite regarder les différentes valeurs que prennent chaque variable pour chaque observation, on peut utiliser la petite astuce consistant à utiliser la fonction `seq_along()`.

```
# Pour avoir des nuages de points
p_1 <- qplot(data = births, seq_along(grams), grams) + xlab("Index")
p_2 <- qplot(data = births, seq_along(gestate), gestate) + xlab("Index")
p_3 <- qplot(data = births, seq_along(educ), educ) + xlab("Index")

library(gridExtra)
grid.arrange(p_1, p_2, p_3, ncol=3)
```



Pour avoir une idée de la relation possible entre la réponse et chaque variable explicative, il est intéressant de tracer les graphiques suivants.

```
p_1 <- qplot(data = births, grams, gestate, geom = c("point", "smooth"))
p_2 <- qplot(data = births, educ, gestate, geom = c("point", "smooth"))
p_3 <- qplot(data = births, black, gestate, position = "jitter")

## Warning: `position` is deprecated

p_4 <- qplot(data = births, smoke, gestate, position = "jitter")

## Warning: `position` is deprecated

grid.arrange(p_1, p_2, p_3, p_4, ncol=2)
```



6.3 Estimation des paramètres

La fonction permettant de réaliser une régression linéaire avec `R` se nomme `lm()`. Il est nécessaire de fournir une formule au paramètre `formula`. Le paramètre `data` indique le *data frame* (la liste, ou un objet convertible en `data.frame`) dans lequel les variables mentionnées dans la formule se trouvent.

```
reg <- lm(grams ~ gestate, data = births)
reg
```

```
##
## Call:
## lm(formula = grams ~ gestate, data = births)
##
## Coefficients:
## (Intercept)      gestate
##     -3245.4        166.4
```

On lit dans la sortie que le coefficient de la constante vaut -3245.446394 et que le coefficient associé à la variable `gestate` vaut 166.4462854.

Comme on peut le voir sur les graphiques proposés précédemment, la relation entre la masse du nouveau né et le temps de gestation semble plus quadratique que linéaire. Pour introduire le carré de la durée de gestation, on utilise la fonction `I()`.

```
reg_2 <- lm(grams ~ gestate + I(gestate^2), data = births)
reg_2
```

```
##
## Call:
## lm(formula = grams ~ gestate + I(gestate^2), data = births)
##
## Coefficients:
##  (Intercept)       gestate  I(gestate^2)
##    -4545.933       243.159        -1.108
```

6.4 Lecture des sorties

Une des fonctions les plus utiles autour de la régression linéaire avec `R` est `summary()`. Elle affiche plusieurs éléments :

- `Call` : la fomule du modèle ;
- `Residuals` : des statistiques descriptives des résidus ;
- `Coefficients` : un tableau à deux entrées où les lignes correspondent aux coefficients associés aux variables explicatives, et les colonnes, dans l'ordre, à l'estimation du coefficient, l'écart-type estimé, la valeur du test de Student de nullité statistique du coefficient et enfin la *p-value* associé à ce test, suivie d'un symbole pour lire rapidement la significativité ;
- `Signif. codes` : les significations des symboles de niveau de significativité ;
- `Residual standard error` : estimation de l'écart-type de l'aléa et degré de liberté ;
- `Multiple R-squared` : coefficient de détermination ;
- `Adjusted R-squared` : coefficient de détermination ajusté ;
- `F-statistic` : valeur de la statistique de Fisher du test de significativité globale, ainsi que les degrés de liberté et la *p-value* associée au test.

```
summary(reg)
```

```
##
## Call:
## lm(formula = grams ~ gestate, data = births)
##
```

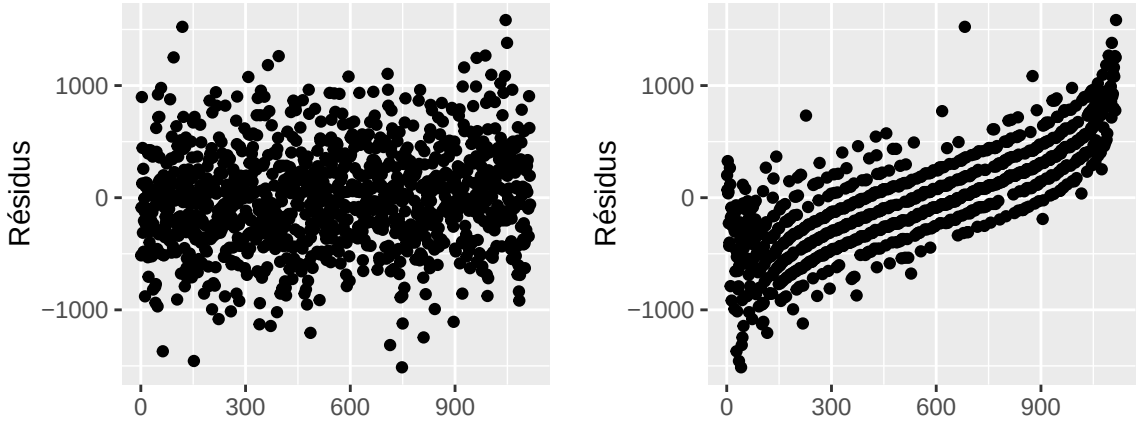
```
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1512.41  -302.17   -12.41   285.15  1584.04
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -3245.45     197.01  -16.47   <2e-16 ***
## gestate       166.45       5.06   32.89   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 451.3 on 1113 degrees of freedom
## Multiple R-squared:  0.4929,Adjusted R-squared:  0.4925
## F-statistic:  1082 on 1 and 1113 DF,  p-value: < 2.2e-16
```

6.5 Extractions

L'objet retourné par la régression contient plusieurs éléments auxquels il est possible d'accéder. Voici les principaux

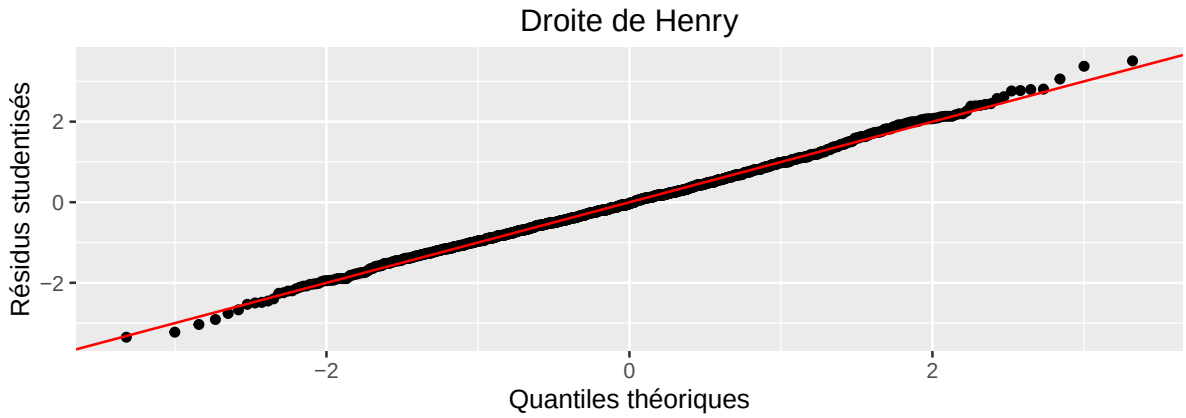
- `coefficients` : un vecteur de coefficients (nommé) ;
- `residuals` : les résidus ;
- `fitted.values` : les valeurs estimées ;
- `df.residual` : nombre de degrés de liberté.

```
names(reg)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"

reg$coefficients

## (Intercept)     gestate
##  -3245.4464    166.4463

# Résidus
qplot(seq_along(reg$residuals), reg$residuals) +
  xlab("") + ylab("Résidus")

# Ordonnons les résidus en fonction de la masse des nouveaux-nés
ind <- order(births$grams)
qplot(seq_along(reg$residuals[ind]), reg$residuals[ind]) +
  xlab("") + ylab("Résidus")
```



Certaines fonctions permettent également d'accéder aux éléments de la régression, comme `residuals()` (ou `resid()`), `fitted()` ou encore `coefficients()` (ou `coef()`) qui retournent les résidus, les valeurs estimées et les coefficients de la régression respectivement. Ces fonctions prennent en paramètre l'objet retourné par `lm()`. Voici un exemple d'utilisation de `residuals()`, pour tracer la droite de Henry (QQ-plot).

```
qqplot <- function(y, distribution=qnorm, title = "Droite de Henry",
                   xlab = "Quantiles théoriques",
                   ylab = "Résidus studentisés") {
  if(class(y) == "lm"){
    # Résidus
    r <- residuals(y)
    # Résidus studentisés
    y <- r / sqrt(deviance(y) / df.residual(y))
  }
  x <- distribution(ppoints(y))
  df <- data.frame(x = x, y = sort(y))
  ggplot(df, aes(x = x, y = y)) +
    geom_point() +
    geom_abline(intercept = 0, slope = 1, col = "red") +
    ggtitle(title) +
    xlab(xlab) + ylab(ylab)
}

qqplot(reg)
```



6.6 Variables catégorielles

En R, les variables catégorielles sont de mode `factor`. Si on souhaite intégrer une variable catégorielle à un modèle de régression linéaire, il y a deux méthodes. La première, est de définir le type de la variable dans le *data.frame* qui contient les données. La seconde est d'utiliser la variable `factor()` dans la formule, lors de l'appel de la régression. La première méthode possède l'avantage de la lisibilité, surtout lorsque l'on souhaite définir la valeur de référence.

Lorsque la variable est de type `logical` ou `character`, la conversion est faite automatiquement par R. Le choix de la classe de référence est aussi effectué automatiquement.

```
class(births$smoke)

## [1] "logical"

summary(reg_3 <- lm(grams ~ gestate + smoke + black, data = births))

##
## Call:
## lm(formula = grams ~ gestate + smoke + black, data = births)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1464.13  -295.56     1.86   287.70  1611.83
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2713.653    199.723 -13.587  < 2e-16 ***
## gestate       156.570      5.016  31.213  < 2e-16 ***
## smokeTRUE    -185.015     30.883  -5.991 2.82e-09 ***
## blackTRUE    -174.402     27.027  -6.453 1.64e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 436.3 on 1111 degrees of freedom
## Multiple R-squared:  0.5269,Adjusted R-squared:  0.5256
## F-statistic: 412.4 on 3 and 1111 DF,  p-value: < 2.2e-16

# Equivalent de
# summary(reg_3 <- lm(grams ~ gestate + factor(smoke) + factor(black),
#                     data = births))
```

Pour changer la valeur de référence, on utilise la fonction `relevel()`, ou alors, on précise les niveaux lors de la création du facteur, le premier énoncé devenant la modalité de référence.

```
births$smoke <- factor(births$smoke)
levels(births$smoke)

## [1] "FALSE" "TRUE"

births$smoke <- relevel(births$smoke, ref = "TRUE")
# Au moment de la création
```

```
births$black <- factor(births$black, levels = c("TRUE", "FALSE"),
                       labels = c("Black", "Not Black"))
summary(reg_3 <- lm(grams ~ gestate + smoke + black, data = births))

##
## Call:
## lm(formula = grams ~ gestate + smoke + black, data = births)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1464.13  -295.56     1.86   287.70  1611.83
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -3073.071    191.836 -16.019  < 2e-16 ***
## gestate          156.570      5.016  31.213  < 2e-16 ***
## smokeFALSE       185.015     30.883   5.991 2.82e-09 ***
## blackNot Black   174.402     27.027   6.453 1.64e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 436.3 on 1111 degrees of freedom
## Multiple R-squared:  0.5269,Adjusted R-squared:  0.5256
## F-statistic: 412.4 on 3 and 1111 DF,  p-value: < 2.2e-16
```

6.7 Tests de nullité des coefficients et intervalles de confiance

Soit le problème de test :

$$\begin{cases} H_0 : \beta_i = 0 \\ H_1 : \beta_i \neq 0 \end{cases} , i = 1, 2, \ldots, m.$$

La statistique de test est la suivante :

$$T = \frac{\hat{\beta}_i - \beta_{i,H_0}}{\hat{\sigma}_{\hat{\beta}_i}} \sim \mathcal{S}t(n - m - 1,)$$

avec β_{i,H_0} la valeur de β_j sous l'hypothèse nulle, $\hat{\sigma}_{\hat{\beta}_i}$ l'estimation de l'écart-type de l'estimation du paramètre β_i.

Pour effectuer ce test bilatéral, on peut lire dans la table de la loi de Student deux fractiles tels que :

$$\mathbb{P}\left(-t_{1-\alpha/2} < \frac{\hat{\beta}_i - \alpha_{i,H_0}}{\hat{\sigma}_{\hat{\beta}_i}} < t_{1-\alpha/2}\right) = 1 - \alpha.$$

avec α le risque de première espèce.
À partir des observations, il est possible de calculer :

$$t_{i,\text{obs.}} = \frac{\hat{\beta}_i}{\hat{\sigma}_{\hat{\beta}_i}}.$$

La règle de décision est la suivante :

- si $t_{i,\text{obs.}} \in [-t_{1-\alpha/2}, t_{1-\alpha/2}]$, nous somme dans la région d'acceptation, on ne rejette donc pas H_0 au seuil de α, et on considère alors que α_i n'est pas statistiquement différent de zéro ;
- si en revanche $t_{i,\text{obs.}} \notin [-t_{1-\alpha/2}, t_{1-\alpha/2}]$, nous sommes dans la région critique et cette fois on rejette l'hypothèse nulle en faveur de l'hypothèse alternative. On considère alors qu'avec un risque de première espèce de α, on a $\alpha_i \neq 0$.

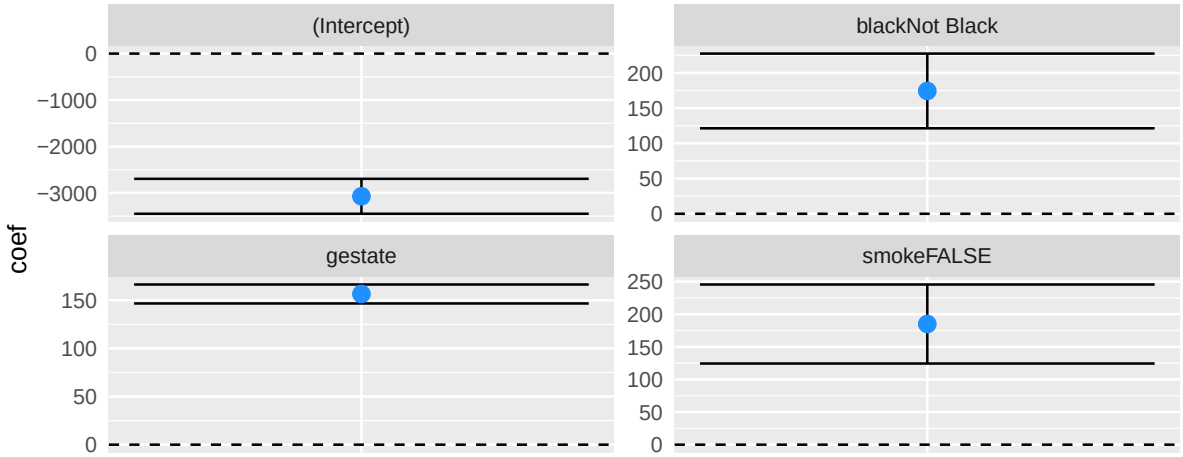
Sous R, comme vu dans la section 6.4, les tests de nullité de chaque coefficients sont effectués lors de l'appel de la fonction `summary()` sur l'objet retourné par la fonction `lm()`. Pour obtenir les intervalles de confiance, on peut s'amuser à extraire soi-même les coefficients et les écarts-types associés, pour faire le calcul à la main, ou bien avoir recours à la fonction `confint()`. Il suffit de lui fournir l'objet retourné par la fonction `lm()`, et de préciser éventuellement un niveau (le niveau par défaut étant 95%).

```
# Intervalles de confiance à 95% pour les paramètres
confint(reg_3)

##                      2.5 %     97.5 %
## (Intercept)    -3449.4726 -2696.6686
## gestate          146.7278   166.4120
## smokeFALSE       124.4204   245.6101
## blackNot Black   121.3724   227.4324

# Intervalles de confiance à 90% pour les paramètres
confint(reg_3, level = 0.95)

##                      2.5 %     97.5 %
## (Intercept)    -3449.4726 -2696.6686
## gestate          146.7278   166.4120
## smokeFALSE       124.4204   245.6101
## blackNot Black   121.3724   227.4324
```

Voici une fonction qui permet d'afficher les intervalles de confiance pour chaque coefficient.

```
# @x : objet issu de lm()
confint_bar <- function(x){
  df <- data.frame(confint(x))
  colnames(df) <- list("b_inf", "b_sup")
  df$variable <- rownames(df)
  rownames(df) <- NULL
  df$coef <- coef(x)

  ggplot(df, aes(x = factor(1), y = coef)) +
    geom_errorbar(aes(ymin = b_inf, ymax = b_sup)) +
    geom_point(col = "dodger blue", size = 3) +
    geom_hline(yintercept = 0, linetype = "dashed") +
    facet_wrap(~variable, scales = "free_y") +
    xlab("") +
    theme(axis.ticks = element_blank(),
          axis.text.x = element_blank())

}
```

```
confint_bar(reg_3)
```



6.8 Prévisions

Une fois que la fonction `lm()` [2] a estimé les paramètres du modèle, il est possible de réutiliser les estimations pour effectuer des prévisions, avec des nouvelles données. On considère un nouvel enregistrement, $\boldsymbol{x}_{n+1}^\top = \begin{bmatrix} x_{n+1,1} & x_{n+1,2} & \dots & x_{n+1,m} \end{bmatrix}$, et l'objectif est de prévoir la valeur de y_{n+1}, en utilisant la relation initiale :

$$y_{n+1} = \beta_0 + \beta_1 \boldsymbol{x}_{n+1,1} + \beta_2 \boldsymbol{x}_{n+1,2} + \cdots + \beta_m \boldsymbol{x}_{n+1,m} + \boldsymbol{\varepsilon}_{n+1}., \tag{6.8.1}$$

où $\mathbb{E}[\varepsilon_{n+1}] = 0$, $\mathbb{V}(\varepsilon_{n+1}) = \sigma^2$ et $\mathbb{C}ov(\varepsilon_{n+1}, \varepsilon_i) = 0$, $i = 1, 2, \dots, n$.

La valeur prévue, $\hat{y}_{n+1}^p$ s'appuie sur les coefficients estimés par le modèle :

$$\hat{y}_{n+1}^p = \hat{\beta}_0 + \hat{\beta}_1 \boldsymbol{x}_{n+1,1} + \hat{\beta}_2 \boldsymbol{x}_{n+1,2} + \cdots + \hat{\beta}_m \boldsymbol{x}_{n+1,m}. \tag{6.8.2}$$

On note $z_{n+1} = y_{n+1} - \hat{y}_{n+1}^p$ l'erreur de prévision. On a :

$$\begin{cases} \mathbb{E}[z_{n+1}] = 0 \\ \mathbb{V}(z_{n+1}) = \sigma^2 \times \left(1 + \boldsymbol{x}_{n+1}^\top (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{x}_{n+1}\right) \end{cases} \tag{6.8.3}$$

Comme on émet l'hypothèse que la distribution des ε_i est normale, la distribution des y_i et $\hat{y}_i^p$ l'est aussi. De fait, on a :

$$z_i^p \sim \mathcal{N}\left(0, \sqrt{\mathbb{V}(z_i^p)}\right). \tag{6.8.4}$$

On peut estimer la variance inconnue σ_ε^2 par son estimation $\hat{\sigma}_\varepsilon^2$.
Dès lors, on a :

$$\frac{z_i^p - \mathbb{E}(z_i^p)}{\hat{\sigma}_\varepsilon} \sim \mathcal{S}t(n-2). \tag{6.8.5}$$

2. voir l'aide de la fonction `?predict.lm`

Il est alors possible de construire un intervalle de confiance au seuil de α pour y_i^p, soit :

$$\text{I.C.}_{\widehat{y_{n+1}}}(1-\alpha) = \left[\hat{y}_{n+1}^p \pm t_{1-\alpha/2} \cdot \hat{\sigma}_{z_{n+1}^p}\right], \tag{6.8.6}$$

où $t_{1-\alpha/2}$ est la valeur du fractile lue dans la table pour α et $\gamma = n-2$ degrés de liberté.

R propose la fonction `predict()` pour calculer cet intervalle de prévision. L'objet retourné par la fonction `lm()` est passé en paramètre à la fonction `predict()`. Si aucun autre paramètre n'est fourni, l'évaluation retourne les valeurs estimées pour la variable à expliquer.

```
all.equal(predict(reg_3), fitted(reg_3))

## [1] TRUE
```

Si en revanche, on ajoute de nouvelles données, en les passant au paramètre `newdata`, alors le modèle estimé est utilisé à partir de ces nouvelles données pour fournir des prévisions. Il faut toutefois faire attention à ce que les noms des variables du nouveau *data frame* soient identiques à celui passé dans la fonction `lm()`.

```
donnees_supl <- data.frame(black = factor(c(TRUE, FALSE),
                                          levels = c(TRUE, FALSE),
                                          labels = c("Black", "Not Black")),
                           educ = c(10,5),
                           smoke = factor(c(FALSE, FALSE)),
                           gestate = c(39, 43))
predict(reg_3, newdata = donnees_supl)

##        1        2
## 3218.171 4018.853
```

Par défaut, les intervalles de prévision de sont pas donnés, il faut forcer leur calcul en donnant la valeur `"prediction"` au paramètre `interval`. L'intervalle de confiance pour la valeur prévue est donné pour un risque de première espèce de 5%. Pour un risque de première espèce différent, il faut changer la valeur du paramètre `level`.

```
# I.C. à 95% pour la prévision
predict(reg_3, newdata = donnees_supl, interval = "prediction")

##        fit      lwr      upr
## 1 3218.171 2361.229 4075.113
## 2 4018.853 3161.006 4876.700

# I.C. à 90% pour la prévision
predict(reg_3, newdata = donnees_supl, interval = "prediction", level = 0.9)

##        fit      lwr      upr
## 1 3218.171 2499.187 3937.155
## 2 4018.853 3299.109 4738.597
```

On peut demander d'afficher les valeurs des écarts-types, avec le paramètre `se.fit`.

```r
predict(reg_3, newdata = donnees_supl, interval = "prediction", se.fit = TRUE)
```

```
## $fit
##        fit      lwr      upr
## 1 3218.171 2361.229 4075.113
## 2 4018.853 3161.006 4876.700
##
## $se.fit
##        1        2
## 18.79725 27.50835
##
## $df
## [1] 1111
##
## $residual.scale
## [1] 436.3423
```

6.9 Exportation des résultats

Bien qu'il soit pratique d'avoir des sorties de régression lisibles sous `R`, un copier/coller de celles-ci dans un document externe n'est pas spécialement esthétique. Le *package* `texreg` (Leifeld, 2013) propose d'excellentes fonctions pour exporter les résultats en `HTML` ou en LaTeX, à savoir `htmlreg()` et `texreg()` respectivement. Voici les paramètres principaux :

- `l` : un modèle statistique, ou une liste de modèles ;
- `file` : le résultat est exporté dans un fichier dont le chemin et le nom sont donnés à ce paramètre ;
- `single.row` : par défaut, deux lignes sont réservées par coefficient de la régression, avec le coefficient sur la première ligne, et l'écart-type sur la seconde. Si la valeur vaut `TRUE`, le coefficient et son écart-type sont placés dans une celle cellule, sur la même ligne ;
- `custom.model.names` : un vecteur de caractères avec les étiquettes pour chaque modèle (au lieu de "Model 1", "Model 2", etc.) ;
- `custom.coef.names` : un vecteur de caractères avec les étiquettes pour chaque variable du modèle ;
- `digits` : nombre de décimales ;
- `caption` : titre pour le tableau ;

Quand on travaille avec un traitement de texte comme LibreOffice, ou Word©, il est pratique d'exporter les résultats (à l'aide du paramètre `file`) en `HTML`, et d'importer ensuite le document créé dans le fichier `odt` ou `docx` (ou en faisant un copier / coller). Dans le cas où on utilise LaTeX, il en va de même.

Dans l'exemple qui suit, les sorties sont volumineuses. Le code est donc commenté, mais le lecteur est encouragé à l'évaluer.

```r
library(texreg)
# htmlreg(reg_3, digits = 2, caption = "Résultats de la régression")
# texreg(reg_3, digits = 2, caption = "Résultats de la régression")

# En enregistrant
htmlreg(reg_3, file = "reg_3.html")
texreg(reg_3, file = "reg_3.tex")
```

Ces sorties ont pour but d'alimenter des fichiers de rapport. Elles ne sont pas très lisibles dans la console. Cependant, Philip Leifeld, le créateur du *package* `texreg` propose une troisième fonction : `screenreg()` qui tente de reproduire au mieux dans la console, ce à quoi ressemblera la sortie !

```
screenreg(list(reg, reg_3), digits = 2,
          custom.model.names = c("Rég. Lin. Simple", "Reg. Lin. Mult."))

##
## ================================================
##                  Rég. Lin. Simple  Reg. Lin. Mult.
## ------------------------------------------------
## (Intercept)      -3245.45 ***      -3073.07 ***
##                   (197.01)          (191.84)
## gestate            166.45 ***        156.57 ***
##                     (5.06)            (5.02)
## smokeFALSE                           185.02 ***
##                                       (30.88)
## blackNot Black                       174.40 ***
##                                       (27.03)
## ------------------------------------------------
## R^2                  0.49              0.53
## Adj. R^2             0.49              0.53
## Num. obs.         1115              1115
## RMSE               451.33            436.34
## ================================================
## *** p < 0.001, ** p < 0.01, * p < 0.05
```

6.10 Exercices

Exercice 6.1 : exploration rapide des données

Cet exercice s'appuie sur un jeu de données de consommation de carburant de 392 véhicules. Il provient de la bibliothèque StatLib, maintenue à la Carnegie Mellon University [3].

6.1.1 Charger le jeu de données `Auto` contenu dans le *package* `ISLR`, puis regarder sa page d'aide ;

6.1.2 Afficher un résumé des différentes variables ;

6.1.3 En utilisant la fonction `stargazer()` contenue dans le *package* du même nom, aficher dans la console un tableau de statistiques descriptives en sortie texte ASCII. Prendre soin de limiter à deux le nombre de chiffres des décimales ;

6.1.4 Exporter ce tableau dans un fichier `HTML`, en prenant soin d'ajouter le titre suivant : "*Statistiques descriptives*". De plus, changer le séparateur des décimales en une virgule au lieu d'un point ;

6.1.5 Représenter par un nuage de points la relation entre les variables de puissance (`horsepower`) et de consommation (`mpg`), puis sur un autre graphique, la relation entre la masse du véhicule (`weight`) et sa consommation ;

6.1.6 Reprendre le code du graphique représentant la consommation en fonction de la masse du véhicule, et faire dépendre la couleur des points du nombre de cylindres (le nombre

de cylindres sera considéré comme une variable catégorielle). Puis, ajouter des courbes de tendance pour chaque catégorie de cylindres à l'aide de la fonction `stat_smooth()`. Ces courbes de tendance devront être estimées à l'aide d'une régression linéaire.

6.1.7 Afficher un tableau des corrélations entre chaque variables numériques ;

6.1.8 En utilisant la fonction `corrplot.mixed()` du *package* `corrplot`, réaliser une visualitation graphique de la matrice de corrélation.

Exercice 6.2 : régression linéaire

Cet exercice s'appuie sur le même jeu de données que le précédent.

6.2.1 Préparer deux tableaux de données : l'un comprenant 80% des observations, et le second les 20% restantes. Les observations à conserver dans le tableau contenant 80% des observations doivent être tirées au hasard ;

6.2.2 En prenant comme jeu de données la base avec 80% des observations, régresser la consommation (`mpg`) sur la puissance (`horsepower`), la masse (`weight`) et l'année de mise en circulation (`year`), en faisant appel à la fonction `lm()` ;

6.2.3 Afficher un résumé de l'estimation à l'aide de la fonction `summary`, puis extraire uniquement le tableau des coefficients ;

6.2.4 Observer les graphiques retournés lorsque la fonction `plot()` est appliquée au résultat de l'estimation ;

6.2.5 Créer un tableau de données contenant les résidus de la régression, ainsi qu'une colonne indiquant le numéro des lignes de chaque observation (que l'on peut appeler `index` par exemple) ;

6.2.6 Tracer les résidus à l'aide d'un nuage de points (les valeurs de la variable `index` seront représentées en abscisses). Puis, changer la représentation géométrique pour afficher un histogramme des résidus ;

6.2.7 Construire un intervalle de confiance à 95% pour chacun des coefficients de la régression. Pour un paramètre α, l'intervalle de confiance est donné par :

$$\widehat{\text{I.C.}_\alpha(1-p)} = \left[\hat{\alpha} \pm t_{p/2,n-m-1} \times \hat{\sigma}_{\hat{\alpha}}\right],$$

avec p le risque associé au test, n le nombre d'observations, m le nombre de variables explicatives et $t_{p/2,n-m-1}$ le quantile d'ordre $p/2$ de la Student à $n-m-1$ degrés de liberté.

Pour réaliser les intervalles de confiance, procéder comme suit :
- récupérer le tableau de coefficients issu du résumé de l'estimation, et le stocker dans un objet de type `data.frame` que l'on appellera `coeffs` ;
- récupérer ensuite le nombre de degrés de libertés associés au test de nullité d'un coefficient ;
- ajouter dans le tableau `coeffs` les variables `b_inf` et `b_sup`, qui correspondent respectivement aux bornes inférieures et supérieures de chaque intervalle.

Enfin, comparer les résultats obtenus avec ceux issus de l'application de la fonction `confint()` à l'objet de la régression ;

6.2.8 Exporter les résultats de la régression dans un fichier `html`, en s'appuyant sur la fonction `stargazer()` ;

6.2.9 En utilisant le modèle estimé et les données contenues dans la base contenant uniquement 20% des observations, effectuer des prévisions sur la consommation des véhicules et les comparer aux valeurs réelles ;

7

Expressions régulières

La section 2.4.3.5 montre des exemples simples de recherches de chaînes de caractères. Celle-ci s'attarde à présenter un moyen de faire des recherches beaucoup plus avancées, à l'aide de ce que l'on appelle les **expressions régulières** (ou *regular expressions*, abrégé par *regex*), qui sont des séquences de caractères formant un motif de recherche (ou *search pattern*).

Avant d'aller plus loin, il convient de rappeler au lecteur l'existence de l'aide sous `R`. La page réservée aux *regex*, accessible en évaluant la commande `?regex`, est un bon aide mémoire. Par ailleurs, le cours intitulé "Les expressions régulières" sur `OpenClassrooms` est un bon support [1]. Cependant, `R` possède quelques terminologiques qui lui sont propres, et qui diffèrent légèrement du cours proposé sur `OpenClassrooms`.

7.1 Les recherches basiques

Les fonctions qui emploient les *regex* en `R` sont constuites de la manière suivante :

```
fonction(pattern, text)
```

Le paramètre `pattern` contient l'expression régulière, et le paramètre `text` doit être un vecteur de caractères dans lequel on souhaite chercher du texte. Lorsque l'expression est trouvée dans une chaîne de caractère, on dit qu'il y a *match*. Le tableau ci-après présente les différentes fonctions du package *base* pour rechercher du texte. Nous ne nous attarderons pas spécialement sur ces fonctions, dans la mesure où celles du *package* `stringr`, plus performantes, seront présentées en section 7.6.

1. http://fr.openclassrooms.com/informatique/cours/concevez-votre-site-web-avec-php-et-mysql/les-expressions-regulieres-partie-1-2

Fonction	Description
`grep()`	Retourne les indices des éléments du vecteur de texte pour lequel il y a un *match*.
`grepl()`	Retourne un vecteur de valeurs logiques de la même longueur que le vecteur `text`, qui indique, pour chaque élément, s'il y a eu *match*.
`regexpr()`	Retourne un vecteur d'entiers de la même taille que `text` donnant la position du premier *match* (ou `-1` s'il n'y a pas eu de *match*), avec l'attribut `match.length`, un vecteur d'entiers donnant la longueur du text *matché* (ou `-1` si pas de *match*).
`gregexpr()`	Même chose que `regexpr()`, mais le résultat est sous forme de liste, dont chaque élément correspond à la recherche du motif dans l'élément de `text` correspondant. Par ailleurs, les positions de départ de chaque *match* sont données, et non pas seulement le premier rencontré. Il en est de même pour l'attribut `match.length`.
`regexec()`	Retourne une liste de la même taille que `text` dont chaque élément est soit `-1` s'il n'y a pas eu de *match*, ou une séquence d'entiers avec les positions de début de *match* de toutes les sous-chaînes correspondant aux sous-expressions parenthésées du paramètre `pattern`, avec l'attribut `match.length`, qui est un vecteur donnant la longueu de chaque *match*, ou `-1` s'il n'y en a pas eu.

Pour se fixer les idées, voici un exemple.

```
texte <- c("Je m'évanouis avant d'avoir pu vous révéler",
           "le nom du kidnappeur", "et/ou de la kidnappeuse",
           "du petit Doug Doug !")
pattern <- "ou"

grep(pattern, texte)

## [1] 1 3 4

grepl(pattern, texte)

## [1]  TRUE FALSE  TRUE  TRUE

regexpr(pattern, texte)

## [1] 10 -1  4 11
## attr(,"match.length")
## [1]  2 -1  2  2

gregexpr(pattern, texte)
```

```
## [[1]]
## [1] 10 33
## attr(,"match.length")
## [1] 2 2
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
##
## [[3]]
## [1] 4
## attr(,"match.length")
## [1] 2
##
## [[4]]
## [1] 11 16
## attr(,"match.length")
## [1] 2 2
```

```
regexec(pattern, texte)
```

```
## [[1]]
## [1] 10
## attr(,"match.length")
## [1] 2
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
##
## [[3]]
## [1] 4
## attr(,"match.length")
## [1] 2
##
## [[4]]
## [1] 11
## attr(,"match.length")
## [1] 2
```

Si on désire rechercher la présence d'un motif ou d'un autre dans un texte, on peut effectuer à la main des appels multiples à une fonction de recherche, un appel par texte à chercher, ou on peut plus simplement utiliser l'opérateur logique "Ou", qui s'écrit `|` en R.

```
texte  <- c("J'accepte votre bonjour, Ashley.",
            "Et je vous l'échange contre un coucou.",
            "Coucou !")
pattern <- "coucou|Ashley"
grepl(pattern, texte)
```

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

Le premier élément de `text` contient le mot `"Ashley"` mais pas `"coucou"`. Il y a bien au moins une des deux sous-chaînes qui a été trouvée. Le troisième élément, en revanche, ne contient ni `"coucou"`, ni `"Ashley"` (il ne faut pas oublier que les fonctions traitant les expressions régulières sont par défaut sensibles à la casse en `R`).

Pour chercher si un motif est présent en début de texte, on fait débuter la sous-chaîne à *matcher* par un accent circonflexe. Pour chercher si un motif est en fin de texte, on termine la sous-chaîne à *matcher* par un symbole dollar.

```
texte <- c("Mais je ne mange pas, voyons !",
           "Mais je ne mange pas", "voyons ! Mais")
grepl(pattern = "^Mais", texte)

## [1]  TRUE  TRUE FALSE

grepl(pattern = "je ne mange pas$", texte)

## [1] FALSE  TRUE FALSE
```

7.2 Les classes de caractères

Les classes de caractères sont des listes de caractères appartenant à un ensemble, comme par exemple les caractètres alphabétiques, numériques, alphanumériques, etc. Il est possible de les construire soi-même, ou bien d'utiliser des classes prédéfinies. Elles sont écrites en les plaçant entre des crochets `[classe]`.
Par exemple, si on désire chercher s'il y a des occurrences du caractère `o` ou `i` entre les sous-chaînes `Cr` et `q`, on définit la classe de caractères `[oi]` :

```
grepl(pattern = "Cr[oi]q", c("Criquette", "Craquer", "Croquette"))

## [1]  TRUE FALSE  TRUE
```

On a cherché dans chaque élément de `text`, s'il y avait la chaîne `Croq` ou `Criq`.

La construction d'une classe de caractère comme `[oi]` ne représente peut-être pas un attrait énorme, mais il est possible d'en créer d'autres plus alléchantes. En effet, en utilisant le tiret (`-`), on peut définir une séquence de caractère. Ainsi, la classe de caractères `[A-Z]` permet de *matcher* les lettres de l'ensemble `ABCDEFGHIJKLMNOPQRSTUVWXYZ`, tandis que `[0-9]` permet de *matcher* les caractères de l'ensemble `0123456789`.

```
# Recherche une voyelle minuscule
grepl("[aeiou]", c("_!", "ALLO", "Allo 9-1-1", "9-1-1"))

## [1] FALSE FALSE  TRUE FALSE

# Recherche une voyelle majuscule
grepl("[AEIOU]", c("_!", "ALLO", "Allo 9-1-1", "9-1-1"))

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

```r
# Recherche un cacractère numérique
grepl("[0-9]", c("_!", "Allo", "Allo 9-1-1", "9-1-1"))

## [1] FALSE FALSE  TRUE  TRUE

# Recherche un cacractère alphabétique ASCII majuscule
grepl("[A-Z]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE FALSE  TRUE FALSE

# Recherche un cacractère alphabétique ASCII minuscule
grepl("[a-z]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE  TRUE FALSE FALSE

# Recherche un cacractère alphabétique (majuscule ou minuscule)
grepl("[A-Za-z]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE  TRUE  TRUE FALSE

# Recherche un cacractère alphanumérique
grepl("[A-Za-z0-9]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

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

Si l'utilisateur désire rechercher un caractère qui n'appartient pas à une classe de caractères, il suffit de rajouter un accent circonflèxe (^) juste après le crochet ouvrant. Si on désire rechercher la présence de l'accent circonflèxe dans un texte, en utilisant une classe de caractère, il faut placer le symbole autre part que juste après le crochet. Pour chercher un crochet fermant (ouvrant) il faut le placer directement après (avant) le crochet ouvrant (fermant) servant à définir la classe de caractères. Enfin, pour chercher un tiret, il suffit de la placer en premier ou en dernier de la définition de la classe.

```r
# Y a-t-t-il autre chose que des chiffres ?
grepl("[^0-9]", c("_!", "Allo", "Allo 9-1-1", "911"))

## [1]  TRUE  TRUE  TRUE FALSE

# Recherche de l'accent circonflèxe ou d'un chiffre
grepl("[0-9^]", c("_!", "Allo", "Allo ^ accent", "Allo 9-1-1", "911"))

## [1] FALSE FALSE  TRUE  TRUE  TRUE

# Rechercher un crochet
grepl("[][]", c("_!", "All[o", "All]o ^ accent", "Allo 9-1-1", "911"))

## [1] FALSE  TRUE  TRUE FALSE FALSE

# Rechercher un tiret ou un i
grepl("[-i]", c("_!", "All[o", "All-] ^ accent", "Allo 9-1-1", "9i11"))

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

Remarque 7.2.1

Il ne faut pas confondre l'accent circonflèxe inséré entre les crochets et celui placé au début d'une chaîne pour indiquer que le motif doit commencer par cette chaîne.

```
# Le texte commence-t-il par autre chose qu'un chiffre ?
grepl("^[^0-9]", c("_!", "Allo", "9-1-1", "911"))

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

`R` propose la bibliothèque de classes appelée *POSIX*. Ces classes s'utilisent en faisant appel à leur nom, de la même manière que les classes définies par l'utilisateur, ou les séquences : `[classe]`. Il faut toutefois noter que les noms des classes de caractères *POSIX* sont légèrement différentes en `R` que dans d'autres langages, comme `PHP` par exemple. En effet, il faut les entourer de crochets et de deux-points. Ainsi, la classe des alphanumériques sera accessible par son nom `[:alnum:]` et utilisée de la sorte : `[[:alnum:]]`. Le fichier d'aide des *regex* de `R` met en garde l'utilisateur que ces classes de caractères prédéfinies peuvent varier selon les configurations des machines, notamment les configurations linguisitiques. Voici quelques exemples.

```
# Recherche un cacractère numérique
grepl("[[:digit:]]", c("_!", "Allo", "Allo 9-1-1", "9-1-1"))

## [1] FALSE FALSE  TRUE  TRUE

# Rechercher un caractère non-numérique
grepl("[^[:digit:]]", c("_!", "Allo", "Allo 9-1-1", "911"))

## [1]  TRUE  TRUE  TRUE FALSE

# Recherche un cacractère alphabétique majuscule
grepl("[[:upper:]]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE FALSE  TRUE FALSE

# Recherche un cacractère alphabétique minuscule
grepl("[[:lower:]]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE  TRUE FALSE FALSE

# Recherche un cacractère alphabétique (majuscule ou minuscule)
grepl("[[:alpha:]]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE  TRUE  TRUE FALSE

# Recherche un cacractère alphanumérique
grepl("[[:alnum:]]", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] FALSE  TRUE  TRUE  TRUE  TRUE

# Recherche une espace ou une tabulation
grepl("[[:blank:]]", c("_!", "Allo\t", "Allo 9-1-1"))

## [1] FALSE  TRUE  TRUE

# Recherche d'un caractère de ponctuation
# (! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~)
grepl("[[:punct:]]", c("_!", "Allo", "Allo 9-1-1", "9-1-1"))

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

R propose également quelques abréviations pour accéder à certaines classes. On y accède en écrivant une lettre minuscule précédée de deux barre obliques inversées. Si la lettre est en majuscule, il s'agit de la recherche de non appartenance à la classe. Par exemple, `\\d` recherche un caractère numérique (*digital*), et `\\D` recherche un caractère n'étant pas numérique. Voici quelques exemples.

```
# Recherche un cacractère numérique
grepl("\\d", c("_!", "Allo", "Allo 9-1-1", "911"))

## [1] FALSE FALSE  TRUE  TRUE

# Recherche un caractère non-numérique
grepl("\\D", c("_!", "Allo", "Allo 9-1-1", "911"))

## [1]  TRUE  TRUE  TRUE FALSE

# Recherche un cacractère alphanumérique
# Attention à cette abréviation qui match le trait de soulignement...
grepl("\\w", c("_!", "Allo", "allo", "ALLO", "9-1-1"))

## [1] TRUE TRUE TRUE TRUE TRUE
```

Le tableau ci-après donne un récapitulatif des classes de caractères prédéfinies.

ASCII	POSIX	Raccourcis	Description
`[[:lower:]]`	`[a-z]`		Lettre minuscule
`[[:upper:]]`	`[A-Z]`		Lettre majuscule
`[[:alpha:]]`	`[a-zA-Z]`		Lettre minuscule et majuscule
`[[:digit:]]`	`[0-9]`	`\\d`	Chiffres de 0 à 9
`[[:alnum:]]`	`[a-zA-Z0-9]`		Caractère alphanumérique
`[[:blank:]]`	`[ \t]`		Espace et tabulation
`[[:cntrl:]]`			Caractère de contrôle
`[[:punct:]]`	`[]!\"#$\\%&\' \\(\\)*+,-\\. /:;<=>\\?@^_ `{|}~[]`		Ponctuation et symbole
`[[:space:]]`	`[ \t\r\n\v\f]`	`\\s`	Espace blanc ou séparateur de ligne ou de paragraphe
`[[:xdigit:]]`	`[A-Fa-f0-9]`		Chiffre hexadécimal
`[[:print:]]`	`[\x20-\x7E]`		Caractère visible et espace (tout sauf caractère de contrôle)
`[[:graph:]]`	`[\x21-\x7E]`		Caractère graphique visible (tout sauf espace et caractère de contrôle)
	`[A-Za-z0-9_]`	`\\w`	Lettre, chiffre et trait de soulignement

7.3 Les quantificateurs

Afin de pouvoir chercher des motifs qui se répètent, dans un nombre défini *a priori* ou non, on peut utiliser ce que l'on appelle des quantificateurs.

Les principaux quantificateurs sont les suivants :

- `?` : le précédent item est factultatif, et sera donc *matché* 0 ou 1 fois ;
- `*` : le précédent item sera *matché* 0 fois ou plus ;
- `+` : le précédent item sera *matché* 1 fois ou plus ;
- `n` : le précédent item sera *matché* exactement n fois ;
- `n,` : le précédent item sera *matché* n fois ou plus ;
- `n,m` : le précédent item sera *matché* au mois n fois, mais pas plus de m fois.

Quelques exemple pour se fixer les idées.

```
# La chaîne "travaille " peut être présente 0 ou 1 fois
texte <- c("Lolo travaille au skateshop", "Lolo au skateshop")
grepl("Lolo (travaille )?au skateshop", texte)

## [1] TRUE TRUE

# Cherche "ah Sonia", ou "ahah Sonia", ou "ahahah Sonia", etc.
texte <- c("ahahah Sonia", "ahah Sonia", "Sonia", "Amandine")
grepl("(ah)* Sonia", texte)

## [1]  TRUE  TRUE FALSE FALSE

# On souhaite s'assurer qu'il y a au moins un "ah" suivi de " Stéphanie"
# Mais on peut aussi avoir "ahah Stéphanie" ou "ahahah Stéphanie"
texte <- c("Bonjour Stéphanie", "ah Stéphanie", "ahah Stéphanie")
grepl("(ah)+", texte)

## [1] FALSE  TRUE  TRUE

texte <- c("a#maisoui#maisoui !", "a#maisoui !", "a#maisoui#maisoui#maisoui !",
           "a#maisoui#maisoui#maisoui#maisoui !")

# Y a-t-il une sous chaîne "a#maisoui#maisoui !" dans le texte ?
grepl("a(#maisoui){2} !", texte)

## [1]  TRUE FALSE FALSE FALSE

# Y a-t-il une sous chaîne "a#maisoui#maisoui !", ou
# "a#maisoui#maisoui#maisoui !", etc. dans le texte ?
grepl("a(#maisoui){2,} !", texte)

## [1]  TRUE FALSE  TRUE  TRUE

# La chaîne "skate" sera présente entre deux et trois fois
grepl("a(#maisoui){2,3} !", texte)

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

7.4 Les métacaractères

Les *regex* possèdent quelques caractères spéciaux, qui ont un statut réservé, et qui sont appelés des *métacaractères*. L'accent circonflèxe en fait partie. En effet, l'expression régulière `^Bonjour` ne recherchera pas la chaîne "*^Bonjour*", comme vu plus haut.

La liste des métacaractères est la suivante :

```
. \| ( ) [ { $ * +?
```

En `R` pour pouvoir rechercher ces caractères dans des chaînes, il faut les échapper, à l'aide de deux barres obliques inversées `\\`.

```
texte <- c("Clément et Marianne vont chez Thibaud.",
           "Clément (et Marianne) vont chez Thibaud !")

# Recherche la présence de "et Marianne"
grepl("(et Marianne)", texte)

## [1] TRUE TRUE

# Recherche la présence de "(et Marianne)"
grepl("\\(et Marianne\\)", texte)

## [1] FALSE  TRUE

# Le dernier caractère est-il n'importe quel caract\`ere ?
grepl(".$", texte)

## [1] TRUE TRUE

# Le texte se termine-t-il par un point ?
grepl("\\.$", texte)

## [1]  TRUE FALSE
```

7.5 Fonctions d'extraction

La fonction `regmatches()` est un complément aux fonction `regexpr()`, `gregexpr()` et `regexec()`. Elle permet d'extraire ou de remplacer, en se basant sur les résultats obtenus avec les fonctions qui viennent d'être citées, les sous-chaînes *matchées*.

7.6 Quelques fonctions du *package* `stringr`

Les fonctions proposées par `R` dans le *package* `base` sont quelque peu restreintes. Le *package* `stringr` propose quelques fonctionnalités supplémentaires. Le nom de ces fonctions possède le préfixe `str_`. La structure de ces fonctions est similaire à celle du *package* `base` qui emploient les *regex* :

```
str_function(string, pattern)
```

avec `string` contenant le texte et `pattern` le motif.

7.6.1 La fonction `str_detect()`

Pour détecter la présence ou l'absence d'un motif dans une chaîne, on peut utiliser la fonction `str_detect()`, qui s'appuie sur la fonction `grepl()`.

```
texte <- c("J'ai pas le temps, j'ai matériellement pas le temps de faire ça",
           "Y a pas assez d'chaises ici ?",
           "J'ai")
str_detect(texte, "j'ai")

## [1]  TRUE FALSE FALSE

# En ignorant la casse
str_detect(string = texte, pattern = ignore.case("j'ai"))

## Please use (fixed|coll|regexp)(x, ignore_case = TRUE) instead of ignore.case(x)

## [1]  TRUE FALSE  TRUE
```

7.6.2 La fonction `str_extract()`

Pour extraire une chaîne qui contient un motif, on peut utiliser la fonction `str_extract()`. Le résultat est un vecteur de caractères de la même longueur que celui fourni au paramètre `string`. Pour les éméments pour lesquels il n'y a pas eu de *match*, la valeur `NA` est retournée.

```
adresses <- c("criquette.rockwell@gmail.com", "brett.montgomery@yahoo.fr",
              "madge_laposte.net")
motif <- "^[[:alnum:].-]+@[[:alnum:].-]+$"

# Extraire les adresses e-mail valides
str_extract(string = adresses, pattern = motif)

## [1] "criquette.rockwell@gmail.com" "brett.montgomery@yahoo.fr"
## [3] NA
```

7.6.3 La fonction `str_extract_all()`

Alors que la fonction`str_extract()` ne permet d'extraire que la première chaîne qui *match* le motif, `str_extract_all()` retourne toutes les occurrences trouvées. Le résultat est une liste de la même longueur que celle du vecteur fourni au paramètre `string`. Les éléments de cette liste sont des vecteurs de la taille du nombre d'occurrences du motif trouvé. En cas d'absence de *match*, la valeur `character(0)` est retournée.

```
tweets <- c("@kevin_ushey yes sure, it was just an example",
            "@hspter @kwbroman next is to get into #github streak art.")
motif <- c("@[[:alpha:]]{1,}")

# Extraire la première mention trouvée
str_extract(tweets, motif)

## [1] "@kevin"  "@hspter"
```

```r
# Extraire toutes les mentions trouvées
str_extract_all(tweets, motif)

## [[1]]
## [1] "@kevin"
##
## [[2]]
## [1] "@hspter"   "@kwbroman"
```

7.6.4 La fonction `str_match()`

La fonction `str_match()` extrait le premier **groupe** trouvé dans une chaîne. Il est donc nécessaire que le paramètre `pattern` contienne un ou des groupes (définis par `()`). Le résultat est une matrice dont la première colonne contient le *match* complêt, les suivantes sont les *match* pour chaque groupe.

```r
telephones <- c("02 23 23 35 45", "02-23-23-35-45", "Madrid",
                "02.23.23.35.45", "0223233545", "Milan",
                "02 23 23 35 45  ", " 02 23 23 35 45",
                "Maison : 02 23 23 35 45")

motif_tel <- str_c(str_dup("([0-9]{2})[- \\.]", 4),  "([0-9]{2})")

# Extrait les numéros de téléphone
str_extract(telephones, motif_tel)

## [1] "02 23 23 35 45" "02-23-23-35-45" NA               "02.23.23.35.45"
## [5] NA               NA               "02 23 23 35 45" "02 23 23 35 45"
## [9] "02 23 23 35 45"

str_match(telephones, motif_tel)

##       [,1]             [,2] [,3] [,4] [,5] [,6]
##  [1,] "02 23 23 35 45" "02" "23" "23" "35" "45"
##  [2,] "02-23-23-35-45" "02" "23" "23" "35" "45"
##  [3,] NA               NA   NA   NA   NA   NA
##  [4,] "02.23.23.35.45" "02" "23" "23" "35" "45"
##  [5,] NA               NA   NA   NA   NA   NA
##  [6,] NA               NA   NA   NA   NA   NA
##  [7,] "02 23 23 35 45" "02" "23" "23" "35" "45"
##  [8,] "02 23 23 35 45" "02" "23" "23" "35" "45"
##  [9,] "02 23 23 35 45" "02" "23" "23" "35" "45"
```

7.6.5 La fonction `str_match_all()`

Si la chaîne de caractère contient plusieurs *matchs*, la fonction `str_match()` ne retournera que le premier. Pour les avoir tous, il faut utiliser la fonction `str_match_all()`. Le résultat est une liste de la même longueur que le vecteur fourni à `string`. Chaque élément de la liste, lorsqu'il y a eu *match* est une matrice comme celle retournée par `str_match()`. S'il n'y a pas eu *match*, l'élément de la liste retournée est `character(0)`.

```
telephones <- c("02 23 23 35 45", "02-23-23-35-45 / 02 23 23 35 35",
                "Madrid", "Maison : 02.23.23.35.45, Travail : 02 23 23 35 35")


motif_tel <- str_c(str_dup("([0-9]{2})[- \\.]", 4),  "([0-9]{2})")

# Extrait les numéros de téléphone
str_extract_all(telephones, motif_tel)

## [[1]]
## [1] "02 23 23 35 45"
##
## [[2]]
## [1] "02-23-23-35-45" "02 23 23 35 35"
##
## [[3]]
## character(0)
##
## [[4]]
## [1] "02.23.23.35.45" "02 23 23 35 35"

str_match_all(telephones, motif_tel)[[3]]

##      [,1] [,2] [,3] [,4] [,5] [,6]
```

7.6.6 La fonction `str_locate()`

Pour localiser la position de la première occurrence d'un motif dans une chaîne, on peut utiliser la fonction `str_locate()`. Le résultat est une matrice d'entiers dont la première colonne indique la position de départ du *match* et la seconde indique la position de fin du *match*. En cas d'absence de *match*, les deux valeurs valent `NA`.

```
fruits <- c("pomme", "banane", "poire", "ananas")
str_locate(fruits, "a")

##      start end
## [1,]    NA  NA
## [2,]     2   2
## [3,]    NA  NA
## [4,]     1   1

# On peut définir le motif pour chaque élément du texte
str_locate(fruits, c("p", "o", "p", "a"))

##      start end
## [1,]     1   1
## [2,]    NA  NA
## [3,]     1   1
## [4,]     1   1
```

7.6.7 La fonction `str_locate_all()`

Si on désire obtenir la position de toutes les occurrences d'un motif dans une chaîne, on peut utiliser la fonction `str_locate_all()`. Le résultat est une liste de la même longueur que le vecteur fourni au paramètre `string`. Chaque élément de la liste contient une matrice d'entiers dont la première colonne contient les positions de départ des *matchs* et la seconde les positions de fin. Chaque ligne de la matrice correspond à un *match*. En cas d'absence de *match* dans la chaîne, la matrice retournée est `matrix(0, nrow = 0, ncol = 2)`.

```
fruits <- c("pomme", "banane", "poire", "ananas")
str_locate_all(fruits, "a")
```

```
## [[1]]
##      start end
##
## [[2]]
##      start end
## [1,]     2   2
## [2,]     4   4
##
## [[3]]
##      start end
##
## [[4]]
##      start end
## [1,]     1   1
## [2,]     3   3
## [3,]     5   5
```

```
# On peut définir le motif pour chaque élément du texte
str_locate_all(fruits, c("p", "o", "p", "a"))
```

```
## [[1]]
##      start end
## [1,]     1   1
##
## [[2]]
##      start end
##
## [[3]]
##      start end
## [1,]     1   1
##
## [[4]]
##      start end
## [1,]     1   1
## [2,]     3   3
## [3,]     5   5
```

7.6.8 La fonction str_replace()

La fonction `str_replace()` remplace la première occurrence d'un motif *matché* par la chaîne indiquée au paramètre `replacement`.

```
telephones <- c("02/23/23/35/45", "02.23.23.35.45", "02 23 23 35 45")

# Remplacer la première occurrence du motif trouvée par un tiret
str_replace(telephones, "[/ \\.]", "-")

## [1] "02-23/23/35/45" "02-23.23.35.45" "02-23 23 35 45"
```

Lorsqu'il y a des groupes (définis par des parenthèses), on peut faire référence aux groupes *matchés* à l'aide de deux barres obliques inversées suivies du rang du groupe. Ainsi `\\1` fera référence au premier groupe matché.

```
fruits <- c("01. pomme", "02. pêches", "Maison", "03. bananes")
str_replace(fruits, "([0-9]{2}\\. )([a-z])", "Debut\\1Bonjour\\2Fin")

## [1] "Debut01. BonjourpFinomme"   "Debut02. BonjourpFinêches"
## [3] "Maison"                     "Debut03. BonjourbFinananes"
```

7.6.9 La fonction str_replace_all()

La fonction `str_replace_all()` permet de remplacer toutes les occurrences du motif trouvé par une chaîne de remplacement.

```
telephones <- c("02/23/23/35/45", "02.23.23.35.45", "02 23 23 35 45")

# Remplacer toute les occurrences du motif trouvées par un tiret
str_replace_all(telephones, "[/ \\.]", "-")

## [1] "02-23-23-35-45" "02-23-23-35-45" "02-23-23-35-45"
```

7.6.10 La fonction str_split()

La fonction `str_split()` permet de séparer une chaîne de caractères en plusieurs morceaux, suivant un motif donné. On peut préciser le nombre maximum de morceaux à retourner à l'aide du paramètre `n`. Ce paramètre vaut `Inf` par défaut, et retourne donc tous les morceaux qu'il est possible de trouver. Si la valeur donnée à `n` est plus petite que lon nombre de morceaux qu'il est possible de faire, le dernier élément du vecteur de caractères retourné contient encore des occurrences du motif. Si par contre la valeur fournie à `n` est plus grande, il n'y a pas de rajouts effectués par `R`.
Le résultat est une liste de la même longueur que le vecteur passé au paramètre `string`. Chaque élément de la liste contient les morceaux. Si le paramètre `pattern` vaut `NA`, le résultat est le texte original, si `pattern` vaut `""`, le texte original est séparé par caractères.

```
textes <- c("manger des chips", "José",
            "Considère qu'on n'est plus amis, Abitbol !")

str_split(textes, " ")
```

```
## [[1]]
## [1] "manger" "des"    "chips"
##
## [[2]]
## [1] "José"
##
## [[3]]
## [1] "Considère" "qu'on"     "n'est"     "plus"      "amis,"     "Abitbol"
## [7] "!"

# Avec n plus grand
str_split(textes, " ", n = 4)

## [[1]]
## [1] "manger" "des"    "chips"
##
## [[2]]
## [1] "José"
##
## [[3]]
## [1] "Considère"            "qu'on"                "n'est"
## [4] "plus amis, Abitbol !"
```

7.6.11 La fonction `str_string_fixed()`

Pour briser une chaîne de caractères en un nombre `n` de morceaux fixe en fonction d'un motif `pattern`, on peut utiliser la fonction `str_split_fixed()`. Le résultat est une matrice à `n`, colonnes. Chaque colonne de la matrice retournée correspond à un morceau. Si `n` est supérieur au nombre de morceaux qu'il est possible de créer, les colonnes de la matrice retournée sont complétées par la chaine vide `""`. En revanche, si `n` est inférieur au nombre de morceaux que l'on peut créer, la dernière colonne contiendra le reste de la chaîne n'ayant pas encore été découpée.

```
textes <- c("manger des chips", "José",
            "Considère qu'on n'est plus amis, Abitbol !")

# Avec 4 morceaux
str_split_fixed(textes, " ", n = 4)

##      [,1]        [,2]    [,3]    [,4]
## [1,] "manger"    "des"   "chips" ""
## [2,] "José"      ""      ""      ""
## [3,] "Considère" "qu'on" "n'est" "plus amis, Abitbol !"
```

Références

Anderson, S. (2012). *A quick introduction to plyr.* Consulté sur http://seananderson.ca/courses/12-plyr/plyr_2012.pdf

Bivand, R. S., Pebesma, E. J., & Gómez-Rubio, V. (s. d.). *Applied spatial data analysis with r* (Vol. 747248717). Springer.

Burns, P. (2011). *The r inferno.* Consulté sur http://www.burns-stat.com/pages/Tutor/R_inferno.pdf

Chang, W. (2013). *R graphics cookbook.* O'Reilly Media, Incorporated.

Charpentier, A. (2014). *Computational actuarial science with R.* Chapman and Hall.

Dalgaard, P. (2008). *Introductory statistics with r.* Springer. Consulté sur http://books.google.fr/books?id=YI0kT8cuiVUC

Elo, I., Rodgriguez, G., & Lee, H. (2001). Racial and neighborhood disparities in birth weight in philadelphia. In *Annual meeting of the populations association of america, washington dc. paper presented, under revision for publication.*

Farnsworth, G. V. (2008). *Econometrics in R.* Consulté sur http://cran.r-project.org/doc/contrib/Farnsworth-EconometricsInR.pdf

Goulet, V. (2014). *Introduction à la programmation en R.* Consulté sur http://cran.r-project.org/doc/contrib/Goulet_introduction_programmation_R.pdf

Lafaye de Micheaux, P., Drouilhet, R., & Liquet, B. (2011). *Le logiciel R : Maîtriser le langage - effectuer des analyses statistiques.* Springer.

Leifeld, P. (2013). texreg : Conversion of statistical model output in R to LaTeX and HTML tables. *Journal of Statistical Software*, *55*(8), 1–24.

Lovelace, R. (2014). *Consistent naming conventions in r.* Consulté sur http://robinlovelace.net/r/2014/07/15/naming-conventions-r.html

Matzner-Løber, É. (2007). *Régression : Théorie et applications.* Physica-Verlag. Consulté sur http://books.google.fr/books?id=Kz-cgsn634kC

Paradis, E. (2002). *R pour les débutants.* Consulté sur http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf

Ross, N. (2014). *Vectorization in r : Why ?* Consulté sur http://www.noamross.net/blog/2014/4/16/vectorization-in-r--why.html

Sanchez, G. (2013). *Handling and processing strings in R*. Berkeley : Trowchez Editions. Consulté sur http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf

Wickham, H. (2009). *ggplot2 : Elegant graphics for data analysis*. Springer.

Wickham, H. (2011). The split-apply-combine strategy for data analysis.

Wickham, H. (2014). Functionals. In *Advanced R* (chap. 3). Consulté sur http://adv-r.had.co.nz/Functionals.html

Xie, Y. (2014). *library() vs require() in r*. Consulté sur http://www.r-bloggers.com/library-vs-require-in-r/

Zuur, A., Ieno, E. N., & Meesters, E. (2009). *A beginner's guide to R*. Springer.

Index des expressions R

Symbols

#....2
*....60, 61
+....59
-....35, 59
...107
/....60
:()....26, 27
<-....2
«-....109
=....3
[()....33, 36, 38, 96
[.data.frame()....90, 91
[.data.table()....84, 88, 91–93
[[()....37, 38, 62, 96
$....39
%*%....60
\%-\%....53
%/%....93
\%>\%....63
%%....93
\%within\%....53
&....31
&&....31

A

aaply()....118
abs()....52, 93
acos()....95
acosh()....95
adply()....118
aes()....137, 139, 157
aggregate()....78
all()....32
all.equal()....31
alply()....118
annotate()....158, 160, 161
anti_join()....73
any()....32, 95
apply()....124, 128
apropos()....5
arrange()....71
arrow()....159, 160
as.data.frame()....18
as.Date()....12
as.POSIXct()....14
as.POSIXlt()....14
asin()....95
asinh()....95
assign....109
atan()....95
atanh()....95
attach....70
attach()....70

B

beep()....116
body()....103
break....115

C

c()....10, 96
casefold()....43
cat()....41, 42
cbind()....72, 96
ceiling()....52, 93
choose()....95
class()....10
coef()....193
coefficients()....193
colnames()....39, 65, 96

confint() 196
Conj() 60
coord_map() 174
coord_cartesian() 167, 169
coord_fixed() 167
coord_flip() 167
coord_map() 168
coord_polar() 168
coord_trans() 168
copy() 89
cor() 94
cos() 95
cosh() 95
cov() 95
crossprod() 61
cummax() 94
cummin() 94
cumprod() 94
cumsum() 94

D

daply() 118, 120
data.frame() 18, 84
data.table() 83
day() 50
dbeta() 29
dbinom() 29
dcast() 82
dchisq() 29
ddply() 118, 120
desc() 71
det() 62
detach() 70
dexp() 29
df() 29
dgamma() 29
dgeom() 29
dhyper() 29
diag() 62
difftime() 51
dim() 16, 18, 96
dimnames() 39, 96
dlnorm() 29
dlogis() 29
dlply() 118, 120
dmy() 15
dnbinom() 29
dnorm() 29
do.call() 126
dpois() 29
dt() 29
dunif() 29
duplicated() 95
dweibull() 29

E

element_blank() 170
element_line() 170
element_rect() 170
element_text() 170
environment() 103
exp() 94
expand.grid() 28

F

facet_grid() 162, 163, 165, 166
facet_wrap() 162, 165, 166
factor() 11, 194
factorial() 95
file.choose() 21
filter() 64
fitted() 193
floor() 52, 93
for() 112, 113
force_tz() 57
formals() 103
fortify() 173

G

gather() 79, 82
geom_bar() 139, 161
geom_boxplot() 139, 142, 143
geom_density() 139, 146
geom_histogram() 139, 145, 147
geom_jitter() 139
geom_line() 139, 141, 148, 157, 159
geom_path() 139, 159
geom_point() 136, 139, 140
geom_polygon() 139, 142
geom_rect() 161
geom_smooth() 139
geom_step() 139
geom_text() 158
get() 43
getMap() 173
getwd() 20, 177
ggplot() ... 135, 136, 139, 140, 144, 148, 160, 172, 173, 175
ggsave() 172
ggtitle() 169
gl() 28
gregexpr() 203, 210
grep() 203
grepl() 48, 203, 211

grid.arrange() 187
group_by() 76

H

head() 95
hour() 50
htmlreg() 199

I

I() 191
identical() 31
ignore.case() 47
Im() 8
inner_join() 73
install.packages() 4
int_aligns() 54
int_end() 54
int_flip() 54
int_length() 53
int_overlaps() 53
int_shift() 54
int_start() 54
intersect() 53
invisible() 104, 118
is.integer() 8
is.vector() 10
isoweek() 50

J

J() 87

L

laply() 118
lapply() 124–127
last_plot() 172
ldply() 118
leap_year() 55
left_join() 73
length() 9, 11
levels() 11
library() 4
list.files() 20
llply() 118
lm() 190, 193, 196–198
load() 25
loess() 144
log() 94, 117
ls() 3

M

map_data() 175
maply() 118
mapply() 124, 130
match() 75
matrix() 15, 16
max() 94
mday() 50
mdply() 118
mdy() 15
mean() 94
median() 94
melt() 82
merge() 75, 90, 91, 96
min() 94
minute() 50
mlply() 118
mode() 7
month() 50
month.abb 57
mutate() 66, 67

N

NA 9
names() 10, 96
nchar() 44
ncol() 16, 18, 96
new_duration() 54
new_interval() 53
next 115
nrow() 16, 18, 96
NULL 9

O

OlsonNames() 15
order() 70, 71, 95
ordered() 11

P

paste() 42
paste0() 42
pbeta() 29
pbinom() 29
pchisq() 29
pexp() 29
pf() 29
pgamma() 29
pgeom() 29
phyper() 29
plnorm() 29
plogis() 29
plot3d() 180
pnbinom() 29
pnorm() 29
position_dodge() 161
position_fill() 161

position_identity() 161
position_jitter() 161
position_stack() 161
ppois() 29
predict() 198
prod() 94
pt() 29
punif() 29
pweibull() 29

Q

qbeta() 29
qbinom() 29
qchisq() 29
qexp() 29
qf() 29
qgamma() 29
qgeom() 29
qhyper() 29
qlnorm() 29
qlogis() 29
qnbinom() 29
qnorm() 29
qplot() 162, 172
qpois() 29
qt() 29
quantile() 94
qunif() 29
qweibull() 29

R

range() 94
rbeta() 29
rbind() 72, 96
rbinom() 29
rchisq() 29
Re() 8
read.csv() 22
read.csv2() 22
read.delim() 22
read.delim2() 22
read.fwf() 23, 24
read.sas7bdat() 25
read.table() 20–23
read.xls() 24
read_csv() 22
read_csv2() 22
read_delim() 22
read_excel() 24
read_fwf() 24
read_tsv() 22
read_xl() 25
regexec() 203, 210
regexpr() 203, 210
registerDoMC() 124
regmatches() 210
relevel() 11, 194
rename() 66
rep() 27
repeat 115
require() 5
resid() 193
residuals() 193
return() 103
rev() 95
rexp() 29
rf() 29
rgamma() 29
rgeom() 29
rhyper() 29
right_join() 73
rlnorm() 29
rlogis() 29
rm() 4
rnbinom() 29
rnorm() 29
round() 52, 93
rownames() 39, 96
rpois() 29
rt() 29
runif() 29
rweibull() 29

S

sample() 95
sapply() 124, 126–128, 130, 131
save() 25
save.image() 26
scale_alpha_identity() 151
scale_colour_identity() 151
scale_fill_identity() 151
scale_linetype_identity() 151
scale_shape_identity() 152
scale_size_identity() 152
scale_alpha_continuous() 151
scale_alpha_discrete() 151
scale_alpha_manual() 151
scale_colour_brewer() 151
scale_colour_continuous() 151
scale_colour_discrete() 151
scale_colour_distiller() 151
scale_colour_gradient() 150, 151
scale_colour_gradient2() 151
scale_colour_gradientn() 151

scale_colour_grey() 151
scale_colour_hue() 151
scale_colour_manual() 151
scale_fill_brewer() 151
scale_fill_continuous() 151
scale_fill_discrete() 151
scale_fill_distiller() 151
scale_fill_gradient() 151
scale_fill_gradient2() 151
scale_fill_gradientn() 151
scale_fill_grey() 151
scale_fill_hue() 151
scale_fill_manual() 151
scale_linetype_continuous() 151
scale_linetype_discrete() 151
scale_linetype_manual() 151
scale_shape_continuous() 152
scale_shape_discrete() 152
scale_shape_manual() 152
scale_size_area() 152
scale_size_continuous() 152
scale_size_discrete() 152
scale_size_manual() 152
scale_x_continuous() 152
scale_x_date() 152
scale_x_datetime() 152
scale_x_discrete() 152
scale_x_log10() 152, 168
scale_x_reverse() 152
scale_x_sqrt() 152
scale_y_continuous() 152
scale_y_date() 152
scale_y_datetime() 152
scale_y_discrete() 152
scale_y_log10() 152
scale_y_reverse() 152
scale_y_sqrt() 152
scan() 20, 22, 23
screenreg() 200
sd() 94
second() 50
select() 62, 64, 66
semi_join() 73
seq() 26, 27, 56
seq_along() 189
seq_len() 27
sequence() 27
setdiff() 53
setequal() 53
setkey() 85
setwd() 20
sign() 52, 94
signif() 52, 93
sin() 95
sinh() 95
slice() 64
solve() 61
sort() 71, 95
spread() 80, 82
sqrt() 94
stack() 79
stat_bin() 147, 149
stat_contour() 149
stat_density() 148, 150
stat_density2d() 150
stat_identity() 150
stat_qq() 150
stat_quantile() 150
stat_smooth() 150
stat_sum() 150
stat_summary() 150
stat_unique() 150
str_detect() 211
str_extract() 211
str_extract_all() 211
str_locate() 213
str_locate_all() 214
str_match() 212
str_match_all() 212
str_replace() 215
str_replace_all() 215
str_split() 215
str_split_fixed() 216
str_c() 42, 43
str_detect() 46, 48
str_dup() 48
str_length() 43, 44
str_pad() 48, 49
str_replace() 47, 48
str_replace_all() 47, 48
str_split() 47, 48
str_sub() 44–46
str_trim() 49
strptime() 12, 14
strsplit() 48
sub() 48
substr() 46
sum() 94, 117
summarise() 76
summary() 95, 191, 196
switch() 114

T

t() 60
table() 95
tables() 84
tail() 95
tan() 95
tanh() 95
tapply() 124, 129
texreg() 199
theme() 169
theme_grey() 172
theme_bw() 172
tolower() 43
toupper() 43
transform() 69
transmute() 67
trunc() 52, 93
txtProgressBar() 116
typeof() 7
tz() 50

U

unclass() 12
union() 53
unique() 65, 85, 96
unlist() 17, 126
unstack() 79

V

vapply() 124, 128
var() 95
Vectorize() 131

W

wday() 50
week() 50
which() 34, 36
which.max() 34, 96
which.min() 34, 96
while() 112
with() 69
with_tz() 57
within() 69
word() 47
write.csv() 25
write.table() 25
write_csv() 25

X

xlab() 169
xlim() 169

Y

yday() 50
year() 50
ylab() 169
ylim() 169
ymd() 15